1 Preparation and Test Drive

Objective: To be familiar with `Anaconda` package and environment manager and `Spyder` IDE that will be used for Python programming throughout this semester.

ATTN: This practical is not assessable and hence attendance is not enforced. However, you are best recommended to attend this practical class to get familiar with the development environment and get ready for the next few weeks practicals.

2 Familiarise with the `Anaconda` package and environment manager.

We use `Anaconda` , a package and environment manager for Python programming throughout this semester. Another popular python IDE/distribution is called Enthought Canopy development environment. You are more than welcome to try it out. To start Anaconda, from the Start menu, click the Anaconda Navigator desktop app once you have logged into the lab computer.

If the `Anaconda` is not already available on your lab computer follow the instructions provided here (https://docs.anaconda.com/anaconda/install/windows/) in order to install it.

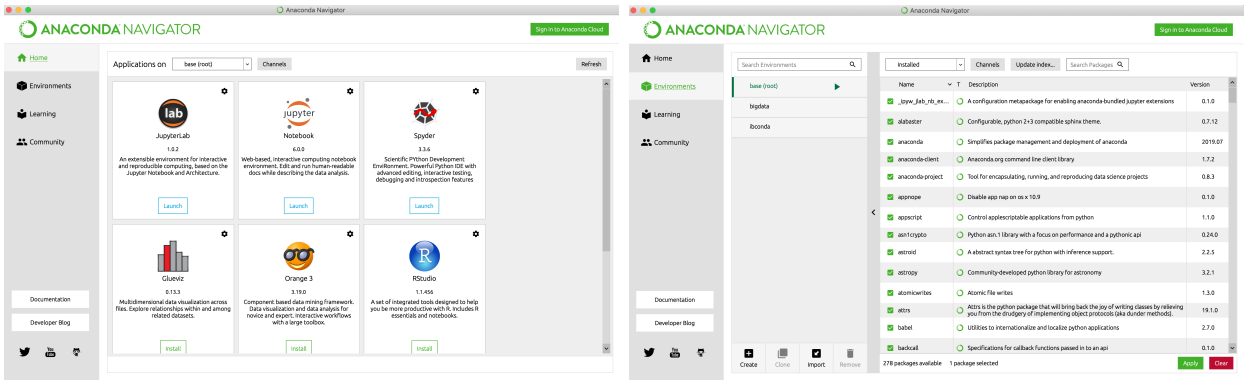


Figure 1: `Anaconda` Navigator (left) and Package Manager (right).

The main (`Home`) interface of `Anaconda` Navigator is shown in the left panel in Fig. 1. There are several applications you can access trough the Navigator. Some of the applications are already available by default and some you'll need to install (by clicking on the **Install** button) if you want to use them.

Click the `Enviroments` tab. It opens environments and package manager shown in the right panel in Fig. 1. Package manager is an important tool for managing third-party packages for `Anaconda` Python distribution. A package is a software library written by other developers and provides some specific functionalities to facilitate code reuse and avoid reinventing the

wheels. You can check the list of installed packages by selecting `Installed` item from the drop-down many in the right panel of the `Enviroments` manager. Here, you can check which packages are already installed, which are available for installation, and look for a specific package and install it.

Using `Enviroments` manager you can create separate environments containing files, packages, and their dependencies that will not interact with other environments. However, we will use the default **root** environment for the rest of the tutorial.

For this tutorial we will use `Spyder` IDE (Interactive Development Environment) provided with the `Anaconda` distribution. Go back to the `Home` tab and start the `Spyder` (Fig. 2). The window is divided into three panels: 1. an editor on the left; 2. an IPython (Interactive Python) Shell on the bottom right panel where you can type Python commands directly to run Python programs interactively and 3. the Variable Explorer on the top right, showing the namespace contents (all global object references, such as variables, functions, modules, etc.)

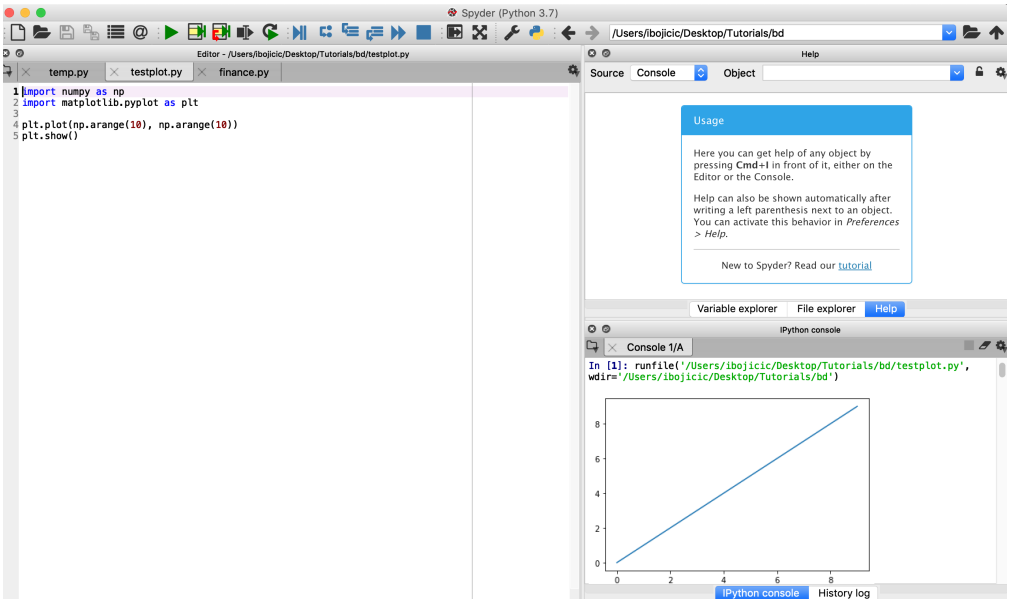


Figure 2: `Spyder` IDE

3 Run Testing Programs

Now you will test the `Anaconda` environment by running some testing Python programs in the `Spyder` IDE. Firstly, create a new file named HelloWorld.py (or any other name with the extension **.py**) that contains the single sentence in the following:

```
print "Hello, world!"
```

Note that python is case sensitive (print must be in lower case letters). Then run it by clicking the **Run File** button on the toolbar (the green triangle). The "Hello, World" message should be printed in IPython Shell embedded in the bottom right panel.

Alternatively, you can type the above command directly in the IPython Shell and it will show the same message. In this way, the program is running in interactive mode. Try to run the program from file and in interactive mode to familiarise yourselves with `Spyder` IDE.

4 Install Required Packages

As mentioned earlier, you need many extra python packages (e.g. pymongo and scikit_learn) for this unit, which may not come with the default installation.

Now choose Package Manager in `Anaconda` (`Envoronments` tab) and search for pymongo package (now search in the **All** packages), check the checkbox next to the package and click to **Apply** to install the package. See Fig. 3 for your reference. Repeat the same process (search and install) to install the following packages: scikit_learn, pandas_datareader, pip and pandas.

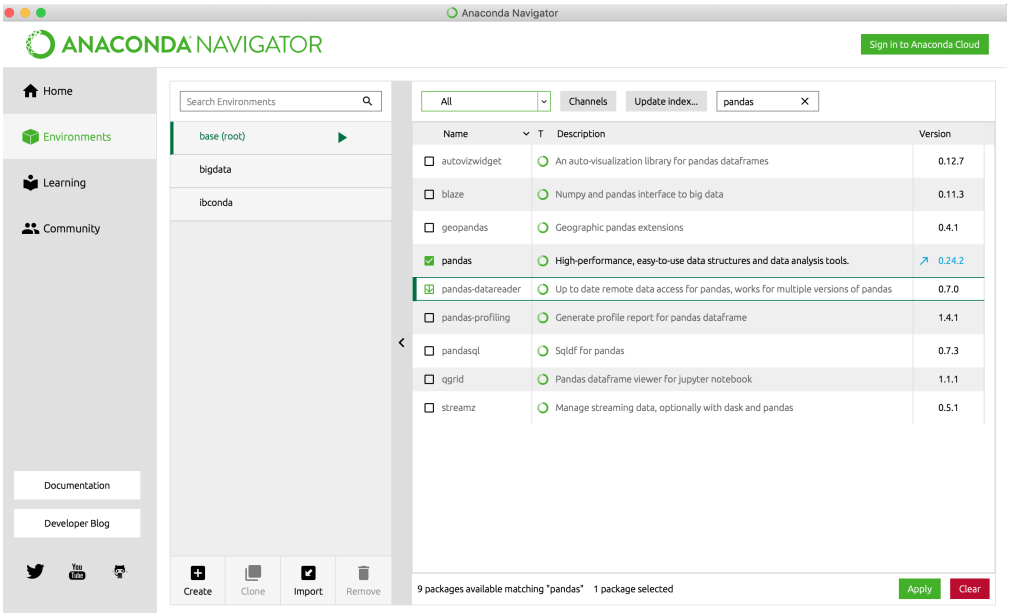


Figure 3: Installing new Python package using `Anaconda` Package Manager

After completing the installation of these two packages, type the following command in IPython Shell

```
import pymongo
```

Again, import should be entered all in lower case. If no error occurs, you then have installed pymongo package successfully. Do the same for other packages you installed. Check with your tutor should you run into any problem.

Sometimes we also need packages that are not included in `Anaconda` distribution. In this case, we need to use pip to install them. pip is a python package manager. After you installed pip package, it should be ready to use. To use pip, you have to go to IPython Shell.

Then type in

```
pip install pandas_datareader
```

in the terminal window to install package pandas datareader. Some messages will show up in the terminal window. It will show whether the package is installed successfully. Install another package called **yfinance** in this way. Use import command in IPython shell to see if you install them successfully.

5 Run More Testing Programs

Once finish installing all the packages, download the following programs from unit vUWS website

testplot.py

finance.py

The first program tests the plotting function. It plots a straight line in a new plot window from bottom-left to top-right corner. The second program tests data acquisition and plotting function of a Python package called Pandas. It plots the stock prices for Apple, Amazon, Google, and Microsoft over the past twelve months in a single plot window with legends and different colour plots for different stocks.

Make sure you run all the programs successfully with the plot similar to the Figures 2 and 4. Show them to the tutor.

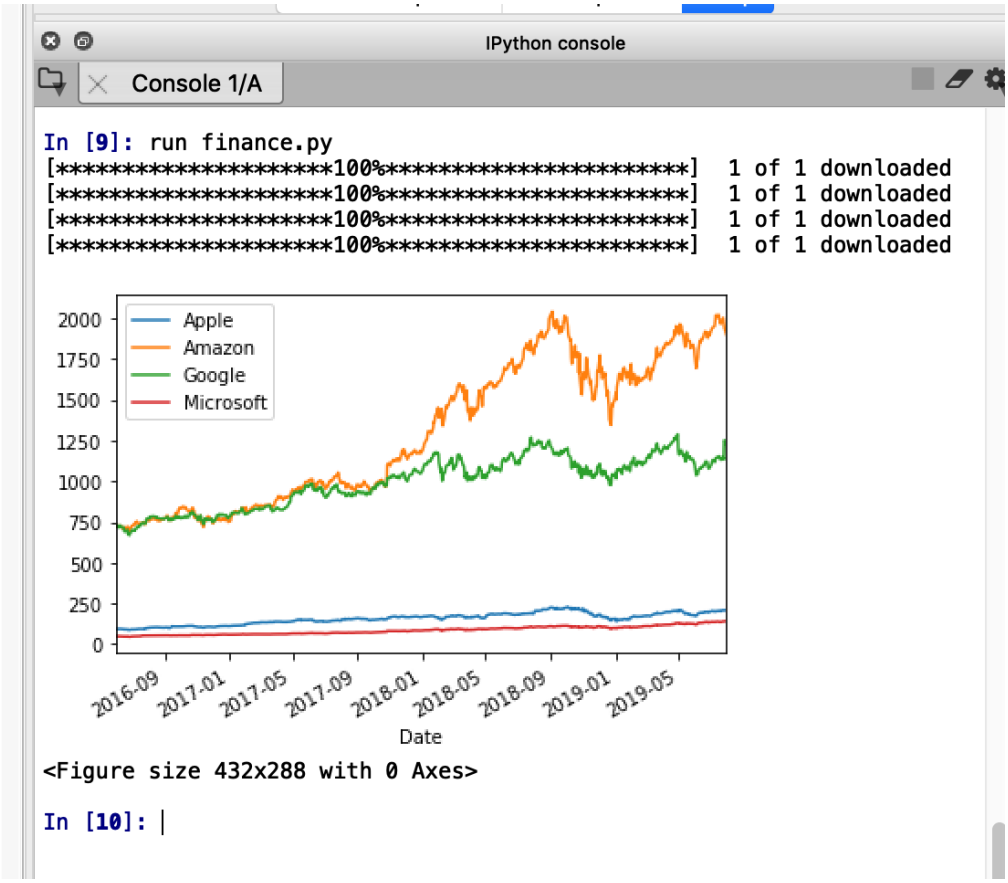


Figure 4: Output of the **finance.py** Python script.

Practical 2: Python Basics II

301110 Applications of Big Data

Week 03, Spring 2019

Objective: Understand and use different Python programming techniques discussed in Week 3 lecture: functions, file input/output, string manipulation, lists, tuples and dictionaries.

Instructions: This practical contains 8 questions covering different aspects of Python programming such as functions, string manipulation, file input/output, lists, tuples and dictionary operations.

You should solve 2 questions for presentation in class. The first question should be selected from Q1–Q4, and the second question should be selected from Q5–Q8.

You should show your solution to the tutor, and discuss your approach.

This practical is not an assessed item, so there are no marks. In order to get the most benefit from this practical, you should not only solve 2 problems, but attempt the others as well, since this will give you a better grounding in Python.

1 Calculate data statistics

Write a program to calculate the sum, maximum and minimum values of a list. You need to define your own functions for calculating the above values instead of calling system functions. A skeleton of your program is provided below and you need to complete the empty functions

```
def mysum(x):
    # Your code goes here to calculate and return the sum of list x
def mymax(x):
    # Your code goes here to calculate and return the maximum value
def mymin(x):
    # Your code goes here to calculate and return the minimum value
# x = [0,1,2,3,4,5,6,7,8,9]
x = range(10)
# expected result: sum= 45 max= 9 min= 0
print "sum=",mysum(x), " max=", mymax(x), " min=",mymin(x)
# x = [1,3,5,7,9]
x = range(1,10,2)
```

```
# expected result: sum= 25 max= 9 min= 1
print "sum=",mysum(x), " max=", mymax(x), " min=",mymin(x)
# x = [1.0,2.0,4.0,8.0,...,512.0]
x = [2.0**i for i in range(10)]
# expected result: sum= 1023.0 max= 512.0 min= 1.0
print "sum=",mysum(x), " max=", mymax(x), " min=",mymin(x)
```

2 Vector norm, inner product and distance

Write a program to calculate the norm of a vector, as well as the inner product and distance between two vectors.

A vector is a mathematical term that can be represented as a list of numbers. Let $x = [x_1, x_2, \ldots, x_n]$, and $y = [y_1, y_2, \ldots, y_n]$ be two vectors of size n. The norm of vector x is given by

$$norm(x) = \sqrt{\sum_{i=1}^{n}} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

The inner product between vectors x and y is given by

$$innerprod(x, y) = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

The distance between vectors x and y is given by

$$dist(x, y) = \sqrt{norm(x)^2 + norm(y)^2 - 2 * innerprod(x, y)}$$

A skeleton program is provided below and you need to complete the empty functions (Hint: square root can be calculated by using `math` library's `sqrt` function)

```
def getNorm(x):
    # Your code goes here to calculate and return the normal
def getInnerProd(x, y):
    # Your code goes here to calculate and return the inner product
def getDist(x, y):
    # Your code goes here to calculate and return the distance
# xVec = [0,1,2,3,4]
xVec = range(5)
# yVec = [4,3,2,1,0]
yVec = range(4,-1,-1)
# expected result: norm(x) = 5.4772 norm(y) = 5.4772
print "norm(x) =", getNorm(xVec)," norm(y) =",getNorm(yVec)
# expected result: innerprod(x,y) = 10
print "innerprod(x,y) =", getInnerProd(xVec, yVec)
# expected result: distance(x,y) = 6.3246
print "distance(x,y) =", getDist(xVec,yVec)
```

3 Vote Counting

You will write a function to count the votes from an input string. The votes are represented by 'Y's and 'N's in both upper and lower cases, and separated by ',' in between. Moreover, there might be extra spaces in the front or/and end of each vote. Your function should display a message for the voting results. The skeleton program is provided below and you need to complete the function `countVotes`

```
def countVotes(votes):
    # Your code goes here to count the input votes and output voting result

votes = "N , Y, Y,N,n , N , N , N ,n ,y, n,N,Y, y,Y,N , N , n ,y, N"
# the function call below should display the message
# Reject: 20 votes in total, 7 accept and 13 reject
countVotes(votes)
votes = " y,n,Y ,y,y ,n, N, Y,N , N,y , n, Y,Y,y, N ,y, n, n , Y "
# the function call below should display the message
# Accept: 20 votes in total, 11 accept and 9 reject
countVotes(votes)
```

4 Word capitaliser

Write a program to capitalise each word in a phrase unless the word is an escaping word like a, an, the, am, is, are, and, of, in, on, with, from, to. For example, for the input message "I am a good player", your program should return "I am a Good Player" at output. A skeleton program is provided in the following and you need to complete the empty function for word capitalisation.

(Hint: you can create a list of escape words above and decide if a new word is an escape word by searching the list)

```
def capitalise(phrase):
    # write your code here to perform word capitalisation

# Output: "I am an Educator and a Researcher"
capitalise("I am an educator and a researcher")
# Output: "Big Data is the Future of Information Technology"
capitalise("big data is the future of information technology")
# Output: "He Wants to Have Breakfast with Her in the Hotel"
capitalise("He wants to have breakfast with her in the hotel")
```

5 Parse File

You will be given a text file (`scemunits.txt`) that contains information about some units offered in SCEM. The text file contains three columns separated by ',',

that correspond to the unit ID, name and the course to which it belongs. The first line of the text file contains the header information that can be skipped for file processing. The text file is on vuws. Write a program that reads the text file and saves all units from the MICT course in a separate text file. A skeleton program is provided below and you need to complete the `readWriteFile()` function.

(Hint: you can use a for loop to read the lines from `infile` *and check whether that line corresponds to a record for an MICT unit. Save that line to* `outfile` *if this is the case and discard the line otherwise. You can use* `readline()` *to skip the first line of* `infile`*.)*

```
def readWriteFile(infile, outfile):
    # Your code goes here to read the content of infile, pick up records of MICT units,
and save the results to outfile

# mictunits.txt should list two MICT units, namely Big Data and Data Science
readWriteFile("scemunits.txt", "mictunits.txt")
```

6 Parse Dictionary

This exercise is similar to the above one except that the unit information is provided in a dictionary. Write a function that accepts the unit list in a dictionary variable, and a keyword string that saves the course information. The function should display all units in a course that matches the keyword.

```
def displayUnits(units, keyword):
    # Your code goes here to pick up all records from the list of units that belong to
the course specified by the keyword, and display the result on screen

units = {('301046','Big Data'): 'MICT', ('300581', 'Programming Techniques'): 'BICT',
('300144', 'OOA'): 'BICT', ('300103', 'Data Structures'): 'BCS', ('300147',
'OOP'):'BCS', ('300569', 'Computer Security'): 'BIS', ('301044', 'Data Science'):
'MICT', ('300582', 'TWA'): 'BICT'}

# the function below should display all MICT units
# 301046 Big Data
# 301044 Data Science
displayUnits(units, 'MICT')
# the function below should display all BCS units
# 300103 Data Structure
# 300147 OOP
displayUnits(units, 'BCS')
```

7 grep

grep is an important utility program in the Unix (MacOS/Linux) operating system that shows all lines in a file that contain certain words or expressions. In this program, you will write a simple grep program. You need to define a grep function in the following, where filename is the name of input file and expr is the expression to be searched for in each line of the input file. The defined grep function can then be used to search for all occurrences of the given expression in a file. The example test file bigdata.txt is on vuws.

```
def grep(filename, expr):
    # your code goes here to complete the grep functionality

# display all lines from bigdata.txt containing Big data
grep("bigdata.txt", "Big data")
# display all lines from bigdata.txt containing technology
grep("bigdata.txt", "technology")
```

8 Top 10 words

For this exercise, you will complete a program that displays the top 10 most frequently occurring words in an input file. We will use the same test file bigdata.txt here that can be found on vuws. A skeleton program is provided in the following, where the sorting and displaying functionality has already been implemented. You need to complete the createDict() function which creates a dictionary of key-value pairs given input file name passed to the function. Each pair in the dictionary has a key, which is a word occurring in the input file, and a value, which is the number of occurrences of that word in the input file. Your program should return this dictionary as output. Any return value of your function other than a dictionary will cause the rest of the program to fail.

```
# itemgetter is used by the sorted() system function
from operator import itemgetter

def createDict(filename):
    # Your code goes here to create and return a dictionary

myDict = createDict("bigdata.txt")
# sort the items of the dictionary by descending order of the second entry of each
pair (i.e. value in the key-value pair)
# note that the return value is a list which is assigned to sortedList
sortedList = sorted(myDict.items(), key=itemgetter(1), reverse = True)
# print the top 10 entries in the sorted list
for key, value in sortedList[:10]:
```

```
print key, value
```

Practical 4: MongoDB

301110 Applications of Big Data

Week 05, Spring 2019

Objective: In this practical you will gain some experience of accessing data stored in a MongoDB database, one of the most popular NoSQL databases used in industry.

Instructions: MongoDB can be installed as a server on a local machine, or in the cloud. In this practical, we give instructions on running the MongoDB server on the SCEM Lab computers Windows 10 build. The MacOS build seems to have some installation problems. You may wish to install MongoDB on your own machine, using either the methods given here or similar instructions available on the web.

Pre-requisites: Install MongoDB on Windows 10

1. Download the MongoDB Windows installer from
   `https://www.mongodb.com/download-center/community`
   use the MSI package for v.4.2.0.

2. Install MongoDB by running the MSI file, using the installation instructions at
   `https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/`

3. During installation, select the option to run MongoDB as a Windows service. (Once installed, it should automatically start.) Include installation of the MongoDB Compass GUI.

4. The MongoDB server should be running. You can talk to it by running the mongo shell. Run the mongo shell by pasting the '`mongo.exe`' line from the installation instructions into a Windows command prompt.

5. In the mongo shell:
   `use local`
   will switch the database to `local`. Then a command like `db.product.find()` will execute on a *Collection* called `product` in the `local` database.

1 Basic MongoDB operations

Try out the tutorials on CRUD (*create, read, update, delete* operations at:

1. `https://docs.mongodb.com/manual/tutorial/insert-documents/`
2. `https://docs.mongodb.com/manual/tutorial/query-documents/`
3. `https://docs.mongodb.com/manual/tutorial/update-documents/`
4. `https://docs.mongodb.com/manual/tutorial/remove-documents/`

2 Using the `product` database

Download the `product.json` file from `vuws` (under Learning Materials → week05 → practical)

Import the JSON file into MongoDB to create the `product` collection:

1. In Compass sidebar, place your mouse over the `local` database and click on the '+' to add the `product` collection.
2. In Compass, Collection menu:
3. Import
4. Select JSON
5. browse to `product.json` on your system and select

You can then use the mongo shell to manipulate the database, e.g.:

1. `use local`
2. `dp.product.find()`

2.1 Q2.1

List all movies in which Morgan Freeman is a cast member. A query that will return all the movies in the product collection is already given below to get you started. You must modify this query to answer the question correctly.

```
db.product.find('Type': 'Movie')
```

2.2 Q2.2

Show only the titles, artists, and the album names of all songs from the Rock genre (including Rock, Progressive Rock, Alternative Rock, Hard Rock, etc) in reverse chronological order (most recent songs first). Again, a query has been provided below to get you started.
Additionally, to query if a value is like a given string (i.e. the equivalent to using the LIKE keyword in SQL) you must use the $regex predicate. Encompassing the string with forward- slashes as shown in the lecture is not supported by pymongo. An example is given below:

```
cursor = db.collection.find('Title': /Programming/)
db.product.find('Type': 'Song')
```

2.3 Q2.3

Using a single query, calculate the average price of books that have more than 500 pages. Be sure to pass the $match and $group functions as strings (i.e. in quotes) as well as all keys and values. e.g.

```
db.product.aggregate([ "$match": ..., "$group": ... ])
```

u301110_prac6a

August 29, 2019

1 Practical 6 - Web Services and APIs

In this practical, you will practice how to extract data from the web using APIs provided by the developers of commercial websites.

This practical is also provided in a form of a **jupyter notebook** document. You can run scripts and create new scripts within this document using **jupyter notebook** from the Anaconda Navigator or you can copy/paste code from this document into a python script or directly into the python shell.

You may find it useful to refer to the examples given in the lecture notes.

1.1 Question 1 - Google Geocode API

1.1.1 Preparation (OPTIONAL TASK)

Since the use of the Google Geocode (GC) API is behind a paywall you are NOT required to query the GC API. If you still want to test access to GC API please follow the instructions below to create an API key (please note that you will need credit card information).

1. Go to https://developers.google.com/maps/documentation/geocoding/get-api-key
2. Click the "GET A KEY" button. You will need to sign in with a Google account if you are not signed in already. Create a Google account if you do not have one.
3. Google will create a new development project for you and begin the process to create an API key for you.
4. Once you have your API key, paste it in the cell below to store it as the Python variable `google_api_key`.

```
In [ ]: google_api_key = "paste your key here"
```

Follow the Google Geocode example discussed in the lecture and create a program to automatically find out the country, latitude and longitude of the following cities using the Geocode API:

Almaty, Beijing, Busan, Chiang Mai, Delhi, Hanoi, Isfahan, Karachi, Kathmandu, Lhasa, Madras, Norilsk, Phnom Penh, Pyongyang, Tashkent, Thimpu, Ulannbaatar, Urumqi, Vientiane, Yokohama

Your program should produce a **JSON file** with the following format:

```
[{"City":"Almaty", "Country": "Kazakhstan", "Latitude":43.2, "Longitude":76.9},
{"City":"Beijing", "Country":"China", "Latitude": 39.9, "Longitude":116.4},
...]
```

Begin coding your solution below, using as many cells as necessary.

1.1.2 Part 1.A

In this task you will insert the JSON data from GC API into a MongoDB collection and write queries to answer some questions. You may find it helpful to refer to Lecture 6 and Practical 5.

First, if you didn't create your own Google Geocode file download *cities.json* file from vuws.

In a case that you importing json file in the mongo shell run the following command from the shell (before login to mongo db):

```
In [ ]: mongoimport --db googlegeocode --collection cities --file cities.json --jsonArray
```

If you're using **jupyter-notebook** enviroment the code to read the JSON file and insert it into a MongoDB collection is already written for you below.

```
In [58]: my_student_id = '123'

         from pymongo import MongoClient
         client = MongoClient()
         db = client[my_student_id]

         def mongo_print(cursor):
             for doc in cursor:
                 print(doc)
                 print()
```

```
In [60]: # change the filename if you named your output file differently in Part A
         with open('cities.json', 'r') as input_file:
             data = json.load(input_file)
             result = db.cities.insert_many(data)
```

Question: Write some MongoDB queries to answer the following questions:

1. What is the eastern-most city (the city with the largest longitude)?
2. What is the western-most city?
3. What is the southern-most city?
4. What is the northern-most city?

1.2 Question 2 - Twitter Streaming API

1.3 Collecting tweets

Following the Twitter live streaming example (discussed in the lecture) a function *collect_tweets* is created for you. This function will initialise tweeter streaming and collect all tweets (for 300sec) to file "tweet_stream.txt". If you not working in the **jupyter-notebook** enviroment copy/paste the following code into your script. Run the cell below to load *collect_tweets* function. The rest of the practical excercises will not work unless you run the cell below first!

```
In [28]: import tweepy
         import time
         from IPython.display import clear_output

         class RawStreamListener(tweepy.StreamListener):
             # note on_data, not on_status
             def on_data(self, data):
                 with open("tweet_stream.txt", 'a') as output:
                     output.write(data)
                 return True

             def on_error(self, status_code):
                 print(status_code)
                 return False

         def collect_tweets(myapi):
             ## initialise stream
             tstream = tweepy.Stream(auth=myapi.auth, listener=RawStreamListener())

             ## start collecting tweets
             tstream.sample(async=True, languages=['en'])

             ## counting down
             for i in range(1,301):
                 time.sleep(1); clear_output(wait = True)
                 print("Collecting tweets, wait for {} sec".format(300-i))

             ## disconnect when finished
             tstream.disconnect()
             print("Done collecting tweets!")
```

To access the live stream, you will need to authenticate yourself. You need to create your access token for authentication purposes by following the steps below:

1. Create a Twitter account if you do not have one yet.
2. Go to the Twitter Apps website (https://dev.twitter.com/apps) and sign in with your Twitter account.
3. Click "Create New App" button on the next page.
4. Complete the application form and click "Create your Twitter application" on the next page. You can enter whatever you like for most fields. You can provide a temporary homepage such as the University homepage (http://www.westernsydney.edu.au).
5. Switch to the "Keys and Access Tokens" tab on the App main page. You will find your Consumer Key (API Key) and Consumer Secret (API Secret) at the top. You will also need access tokens (Access Key and Access secret) to use Twitter APIs for your application. These can be created by scrolling to the bottom of the page and clicking "Create my access token".
6. Cahnge your API details for four access keys and run each line in the ipython shell

```
In [29]: access_token = "YOUR ACCESS TOKEN"
         access_secret = "YOUR ACCESS SECRET"
```

```
consumer_key = "YOUR CONSUMER KEY"
consumer_secret = "YOUR CONSUMER SECRET"

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)
api = tweepy.API(auth)
```

Then, run the following line in the ipython shell

```
In [3]: collect_tweets(api)
```

You should now have a file "tweet_stream.txt" in your folder. You should examine several tweets for the structure. Twitter objects are explained here in more details.

1.3.1 Part 2.A

For this part, you will create a function to extract important informations from each tweet:

1. (**tweettext**) tweeted text
2. (**name**) user's real name
3. (**username**) user's screen name
4. (**self_loc**) user's self-determined location (location entered by the user)
5. (**real_loc**) user's real location (location determined by the twiter, be aware that this field is usually empty)
6. (**verified**) is the user verified or not

You are provided with a skeleton which accepts one argument *tweetline* which is a line (string) in your "tweet_stream.txt" file, and returns all required fields. *Hint: Use one line from your "tweet_stream.txt" file to test your function.*

```
In [42]: def extract_from_tweet(tweetline):
             ## your code goes here



             return tweettext,name,username,self_loc,real_loc,verified
```

07-Practical

July 18, 2020

1 Practical 7 - Map Reduce

Objective: In this practical, you will: - get hands on experience with HDFS and Hadoop systems, and - do a simple word count with map-reduce using Hadoop streaming.

To access the SCEM big data machine, you need your SCEM account.

Go to the following website: https://hadoop-01.scem.westernsydney.edu.au to register an account using your SCEM user name. Once this is done, you should also have a MySQL server database with the name like `db_scemaccountname`, and a user directory on the Hadoop HDFS.

Once you've finished the registration, you're ready to use the big data machine!

1.1 HDFS and Hadoop

Once you've logged into the big data machine and have a terminal available to you (using ssh or putty), you can act on the Hadoop **HDFS** file system, by issuing commands starting with `hadoop fs` on the terminal.

Try **HDFS** commands for the following tasks, which are commonly used when you are dealing with Hadoop and map reduce:
- list all files under */users/ugbigdata/examples*. - list all files under */users/yourscemaccount/*. - create a directory in */users/yourscemaccount/* called *firstmapreducedata*. - copy the files under */users/ugbigdata/examples/moviedata/* to the directory you just created. - display the content of the first file in your directory. - copy one of the files to your local directory (not on HDFS). - remove the entire directory you created in */users/yourscemaccount/* and copy the whole directory */users/ugbigdata/examples/moviedata* to */users/yourscemaccount/*.

Refer to the lecture notes when needed or use `hdfs dfs` to get help as necessary.

Record the commands you used below.

[ ]:
```
%%bash
hdfs dfs -put /localfs/source/path /hdfs/destination/path
...
```

1.2 First Map Reduce Using Hadoop Streaming

Now we move on to do the first map-reduce job…

We start from the example used in the lecture, counting the number of words from all movie reviews. But we begin with a small set of review documents, which is available in

`/users/ugbigdata/examples/smallmoviedata/`. Copy this directory including all files in it to your own directory on HDFS. Make sure you get this done correctly by using

```
hdfs dfs -ls /hdfs/path
```

where `/hdfs/path` is the path to the folder you just copied.

Now run the map reduce job using Hadoop Streaming

[ ]:
```
%%bash
export HLIB=/local/apps/hadoop/share/hadoop/tools/lib
hadoop jar ${HLIB}/hadoop-streaming-2.7.3.jar \
  -input /users/yourscemaccount/smallmoviedata/ \
  -mapper /usr/bin/cat \
  -reducer /usr/bin/wc
  -output /users/yourscemaccount/movieoutput/
```

You need to copy and paste the above linux commands (without `%%bash`) to your terminal and enter to make it run. You may have to insert the HLIB path by into the command line by hand. (Ask the tutor about this…)

The output will be generated into `/users/yourscemaccount/movieoutput/` directory. Copy and paste what you see in it in the next cell (double click the next markdown cell to add):

What is the output of the above map reduce job? Please add your output here.

There should be at least two files if the job is successful. One is simply `_SUCCESS_`. The other is a file named `part-00000`, which is the output generated by the first reducer. If you have multiple reducers, then you should see multiple of `part-*****` files. Display the content of file `part-00000`, copy and paste the text in the following markdown cell (double click the next markdown cell to add):

What is in file `part-00000`? Please add your output here.

Now copy the whole `smallmoviedata` to your local directory (not on HDFS), e.g. into `~/examples/smallmoviedata`. Check to ensure all files in `smallmoviedata` have been successfully copied through from HDFS. The purpose is to run the same code **locally** instead on *Hadoop* to check if the mapper and reducer are doing the right thing. Run the following `bash` command to see if the results agree with what you just did with Hadoop Streaming.

[ ]:
```
%%bash
ls *.txt | xargs -I % bash -c "cat % <(echo)" | sort | wc
```

To understand the "little bars", which is called pipelines formally in Linux system, in above bash command, use the following command to invoke the help page:

[ ]:
```
%%bash
man -P 'less -p Pipelines' bash
```

The local version is basically mimicking Hadoop streaming. The details have been fully covered in lecture 7.

Now we move on to full moview review data set.

1.2.1 Run the word counting on full movie review data set

The full movie review data is available in `/users/ugbigdata/examples/moviedata/`. Copy this directory including all files in it to your own directory on HDFS. Make sure you get this done correctly by using `hdfs dfs -ls /hdfs/path` where `/hdfs/path` is the path to the folder you just copied. Once setup correctly, you can run the map reduce job now:

[ ]:
```
%%bash
export HLIB=/local/apps/hadoop/share/hadoop/tools/lib
hadoop jar ${HLIB}/hadoop-streaming-2.7.3.jar \
  -input /users/yourscemaccount/moviedata/ \
  -mapper /usr/bin/cat \
  -reducer /usr/bin/wc
  -output /users/yourscemaccount/fullmovieoutput/
```

This is going to take some time. While you are waiting, you can check the progress from the link it provides after you run the above command or simply go to http://hadoop-01.scem.westernsydney.edu.au:8088. In `Hadoop-01`, the available browser is `firefox`. Use the following command to start it and connect to Hadoop job monitor

```
firefox http://hadoop-01.scem.westernsydney.edu.au:8088 &
```

Once in the Hadoop job monitor, look for `running job` for detail. Click through all tabs on the left hand side to see what information it provides.

When the job has finished, record the output in the following cell.

What is the final word counts of all movie reviews? Please add your output here.

Check the correctness of the above MR (map reduce) job by running similar commands locally (as for the smaller job above). Finally answer the following questions:

1. What is the *mapper* used in Hadoop streaming?
2. What is the *reducer* used in Hadoop streaming?
3. How many reducers used in the above Hadoop streaming MR jobs?

See http://hadoop.apache.org/docs/r2.7.3/hadoop-streaming/HadoopStreaming.html for details about Hadoop streaming.

[ ]:
```

```

Practical 8 - MapReduce Part II

Objective: In this practical, you will get some more practice with the map reduce computing model (MR) using Hadoop streaming, and apply it to more advanced examples.

In this practical, we assume that you can handle files stored in HDFS using hdfs commands, as in the previous practical. If you are not confident about this, make sure you have completed practical 7. You can also use the following resources:

- lecture 7 and 8
- HDFS or Hadoop documentation on http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html (http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html) You will need access to the SCEM big data facility (hadoop-01.scem.westernsydney.edu.au). If you have not already got this running, please refer to practical 7 for the procedure.

There are 5 main tasks and one extra task in total in this prac.

The minimum requirement is to carry out Tasks 1, 2, 5 and one of 3 or 4 - the choice is yours.

You are advised to also do the other one of tasks 3 and 4 to build up your experience with HDFS, Hadoop, Hadoop streaming and - most importantly - the map reduce computing model.

*If you finish all 5 tasks, you may wish to attempt the extra task. But you are advised **not** to attempt the extra task unless you have done the rest.*

You should save your notebook or notebooks for future reference.

Task 1: Hadoop streaming with python for TF

In prac 7, we did a simple MR for word counting using Hadoop streaming, in which the mapper and reducer were system commands, `cat` and `wc` respectively. It worked well enough for counting how many words *only*. However, we may need a more detailed analysis, in which we count the occurrences of each word, i.e. term frequency (TF). We use python for this.

The python code may be found in the lecture notes for week 8: `hadoopmapper_wcnt.py` and `hadoopreducer_wcnt.py` .

Following the steps below to make this MR job run on Hadoop.

1. copy `smallmoviedata` and `moviedata` to your HDFS path
2. create `hadoopmapper_wcnt.py` and `hadoopreducer_wcnt.py` on your local directory
3. run Hadoop streaming command shown in the next cell (without `%%bash` part)
4. monitor the MR job
5. fetch the MR results to your local directory

N.B. The following Hadoop streaming command needs some modification, since it is specifically for `smallmoviedata` . Modify it for use with `moviedata` .

In [ ]:
```
%%bash
hadoop jar /local/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.
7.3.jar \
  -mapper hadoopmapper_wcnt.py \
  -reducer hadoopreducer_wcnt.py \
  -input /hdfs/path/to/smallmoviedata/ \
  -output /hdfs/path/to/outputdirectory \
  -file local/path/to/hadoopmapper_wcnt.py \
  -file local/path/to/hadoopreducer_wcnt.py
```

Display your results here for both data sets.

In [ ]:
```
%%bash
# Assume you get the results on smallmoviedata and put them into ~/TFsma
llmoviedataresults/
# Run this cell
cat ~/TFsmallmoviedataresults/*
```

In [ ]:
```
%%bash
# Assume you get the results on smallmoviedata and put them into ~/TFall
moviedataresults/
cat ~/TFallmoviedataresults/*
```

Task 2: Hadoop streaming with a single python file for TF

Again the python code may be found in the lecture notes for week 8: `mr_one.py` .

Following the steps below to make this MR job run on Hadoop (step 1 can be skipped if you've done task 1).

1. copy `smallmoviedata` and `moviedata` to your HDFS path
2. create `mr_one.py` on your local directory
3. run Hadoop streaming command shown in the next cell (without `%%bash` part)
4. monitor the MR job
5. fetch the MR results to your local directory

N.B. The following Hadoop streaming command needs some modification, and it is for `smallmoviedata` . Modify it for `moviedata` . Also the option `numReduceTasks` is on, which is set to 5, i.e. there will be 5 reducers running in parallel.

In [ ]:
```
%%bash
hadoop jar /local/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.
7.3.jar \
 -files local/path/to/mr_one.py \
 -mapper 'python mr_one.py map' \
 -reducer 'python mr_one.py reduce' \
 -input /hdfs/path/to/smallmoviedata/ \
 -output /hdfs/path/to/outputdirectory \
 -numReduceTasks 5
```

You should have exactly the same results as those from task 1. Check this. But there is a complication, i.e. there are 5 outputs from running above MR job. But the counts should be the same.

Task 3: Inverse documents counts (IDC or ITF)

In this task, you are to do an IDC analysis on movie review data. The requirements are:

1. use mr_one.py as skeleton (see week 8 lecture notes). So revise `mr_one.py` to suit the needs.
2. use only 1 reducer in MR.
3. try MR on `smallmoviedata` first and then apply to `moviedata` once you know your code works.
4. copy and paste your code in the following code cell.
5. copy and paste top 10 words that appeared in most reviews.

In [2]:
```
# Copy and paste your finished mr_one.py for task 3 here
```

Top 10 words that appeared in most review documents:

Task 4: find Co-occurring products using MR

Find co-occurring products in supermarket basket data. The data should be in HDFS `/users/ugbigdata/examples/supermarket/purchaseHist.csv` . If not, you can get it from vuws. Every single line is a list of purchased items:

icecream,chocolate,bread,coffee,tea,milk,apple,chicken

milk,bread,icecream,chocolate,coffee,tea,apple,chicken

milk,bread,chocolate,coffee,tea

chocolate,icecream,milk

tea,coffee,chicken

Requirements:

1. Use `mr_one.py` as a skeleton for your code, and apply it to the data using Hadoop streaming.
2. copy and paste your code in the following code cell.
3. copy and paste top 10 co-occured item pairs.

In [4]:
```
# Copy and paste your finished mr_one.py for task 4 here
```

Top 10 co-occured item pairs:

Task 5: compute simple statistics using MR - mean and covariance matrix

Compute simple statistics using MR model. In this case, we consider only mean and covariance matrix. The data are in HDFS `/users/ugbigdata/examples/test/multivariatedata.txt` (or on vuws). Every single line contains some number of observations separated by ":" as shown below:

```
0.59,1.64,2.5,0.93,0.58:-0.09,1.13,1.68,0.76,1.31:1.43,1.14,-0.03,0.88,2.87
2.1,0.99,0.61,1.65,1.95:1.46,1.81,2.26,1.96,3.68
```

So the first line has 3 observations, each of which is a vector of length 5, in which elements are separated by commas. The second line has 2 observations. (The data in `/users/ugbigdata/examples/test/multivariatedata.txt` are not necessarily the same, for example, the number of decimal places may be different.)

The covariance matrix of multivariate observations, written as C is defined as

$$C = \sum_{i=1}^{N} \tilde{\mathbf{x}}_i \tilde{\mathbf{x}}_i^\top$$

where

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \mathbf{m}$$

and $\mathbf{x}_i$ is the ith observation and $\mathbf{m}$ is the mean of all observations, i.e.

$$\mathbf{m} = \frac{\sum_{i=1}^{N} \mathbf{x}_i}{N}$$

In the case above, $\mathbf{x}_i$ is a vector of length 5.

See also https://datascienceplus.com/understanding-the-covariance-matrix/ (https://datascienceplus.com/understanding-the-covariance-matrix/) for an explanation of how to build a covariance matrix. (the code on this website uses some python - e.g. numpy - that you may not be familiar with, so it's not advised to simply copy-paste!)

Requirements:

1. Use `mr_one.py` as a skeleton for your code, and apply it to the data using Hadoop streaming.
2. copy and paste your code in the following code cell.
3. copy and paste the mean and covariance matrix you have found.

In [5]:
```
# Copy and paste your finished mr_one.py for task 5 here
```

Mean and covariance matrix computed for the data:

Extra task: TF-IDF

This problem is an advanced task. If you don't finish it in the lab, feel free to work on it at home. (Access to the SCEM big data machine from outside the university may require use of the SCEM VPN service.)

For this problem, you are required to implement the TF-IDF metric as outlined in the lecture and restated in the following:

Term Frequency

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).

Inverse Document Frequency

IDF(t) = log_e(Total number of documents / Number of documents with term t in it).

TF-IDF

TF(t) * IDF(t)

A helpful website for a more complete overview is: http://www.tfidf.com/ (http://www.tfidf.com/). Note that there are many versions of TF-IDF available each of which claims to be good at some aspects. We have no preference for which version to use. The one given above is a simple version that is easily understood.

You will have to construct the required mapper and reducer functions in python and apply them to the whole movie review data set.

TF is associated with one document, i.e. the same word/term has different TF in different documents. For this reason, you may use a single document of your choice to calculate TF.

However, for IDF, you have to go through all documents as IDF is counts how many documents contain a given word.

```
In [ ]:
```

prac09

July 18, 2020

1 Practical 9 - Big Data Analytics (Part I)

Objective: Understand some basic concepts and techniques for classification by predictive modelling, using a jupyter notebook. Understand linear and kernel SVMs and identify their strengths and weaknesses in application. Understand the importance of parameter selection and describe the cross validation technique for parameter selection.

For this practical, your task is to review this notebook which outlines two examples of predictive modeling being applied to two datasets. All code blocks can be executed if there is no instruction asking for your code. You do need to read the notebook carefully and test everything by yourself. Make sure you understand everything before continuing with the questions at the end and the final coding task.

N.B. all code blocks have to be excuted in sequence. If you encounter any difficulties, run the code from the very first block. In the final coding task, ***DO NOT*** *assume any preloaded packages or data set other than those appear in coding task blocks!*

1.1 Iris Classfication

In this exercise, you will work through the Iris classification example discussed in the lecture, from data loading, classifier training and prediction.

The **iris dataset** is an example dataset included in the `scikit-learn` package. But here we use `pandas` package to read it in from a `.csv` file to familiarise yourself with I/O. The **iris dataset** is widely used to demonstrate predictive modelling (classification) techniques in machine learning. The data set consists of 50 samples from each of three species of the Iris flower genus (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Four features were measured from each sample (in centimetres): 1. Sepal length 2. Sepal width 3. Petal length 4. Petal width

This results in a dataset with 150 records (50 samples * 3 species of iris) where each record contains 4 measurements. The code below imports some libraries we will use for the data analysis and plotting. A brief description of each library is given below. - `sklearn` is a library for machine learning. It also comes with the iris dataset. - `numpy` is a library for numerical calculations. - `pandas` is a library for manipulating data such as tables. - `seaborn` is a statistics and plotting library. - `mlxtend` is a library containing some additional tools for machine learning.

N.B. if you need to install any of the packages, you should be able to use conda to get them into anaconda. But try a simple import first

[ ]:
```
%matplotlib inline
from sklearn.svm import LinearSVC, SVC
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import seaborn as sns
from mlxtend.evaluate import plot_decision_regions
```

[ ]:
```
iris = pd.read_csv('iris.csv')  # load the iris dataset
iris.head(5)
```

The purpose of this dataset is to find a model that can predict which species of iris (*setosa, versicolour, virginica*) a flower belongs to given these measurements. We call the species we are attempting to classify *targets*. For ease of use in a programming environment, each target corresponds to a numerical label value:

target
0
1
2

To start, we need to create training and testing sets from the raw dataset. This can be done with the following statement that splits the initial dataset into 60% (90 examples) and 40% (60 examples) subsets for training and testing respectively.

[ ]:
```
training, testing = train_test_split(iris, train_size=0.6)
```

This produces four arrays that we have named `training_data`, `testing_data`, `training_labels` and `testing_labels`. As an example, `training_data` contains 90 samples from the original dataset (i.e. 90 records of the 4 physical iris measurements) and `training_labels` is an array of 90 integers representing which species the elements of `training_data` belong to (e.g. the species of iris that the first record in `training_data` belongs to is given as the first record in `training_labels`). Remember that the species have been given an integer identifier rather than storing the names as strings.

We now want to examine the statistical properties of training examples from different targets (species) in terms of mean values and standard deviations for each feature. This will help us identify which features differ the most between species.

1.1.1 Mean values by target

[ ]:
```
iris.groupby('target').mean()  # the groupby methods forces the mean to be␣
↪calculated per target
```

1.1.2 Standard deviation by target

[ ]:
```
iris.groupby('target').std()
```

1.1.3 Group mean standard deviation

[ ]:
```
iris.groupby('target').mean().std() # Group mean stdandard deviations
```

It seems that the two features `petal_length` and `petal_width` differ greatly between species. Their group means vary more than other two features, indicated by large group mean std devs shown above. The larger the group mean, the further apart they are.

The discriminative nature of features can be verified with correlation analysis between feature values and target values. Calculating the correlation coefficient between the target (the target is just an arbitrary integer) and the feature measurements tells us how effective that feature is at distinguishing species. If the coefficient is close to -1 or 1, then the two are strongly correlated, if the value is close to 0, then the values are very weakly correlated.

[ ]:
```
for feature in iris.columns:
    # print the correlation coefficient between the feature and the species␣
↪(target)
    if feature != 'target':
        print 'Correlation between {} and target: '.format(feature) +␣
↪str(iris[feature].corr(iris['target'])) + '\n'
```

From the results above, we can see that features `petal_length` and `petal_width` are the best candidate features for iris classification, confirming our general observation of the mean statistics earlier. This can be visualised by plotting the measurements of all samples against each other as the following.

[ ]:
```
sns.pairplot(iris, hue='target', vars=('sepal_length', 'sepal_width',␣
↪'petal_length', 'petal_width'))
```

The off-diagonal plots are pairwise scatter plots, e.g. the left bottom corner is the scatter plot of `petal_width` against `sepal_length`. The diagonal plots are histograms of the property. All of these plots are grouped according to species, i.e. each color for each species. The above plots again show that `petal_length` and `petal_width` are the most strongly correlated features against the species of iris. Notice how the points for each species in the plots containing these features form distinct groups, in contrast to `sepal_length` vs `sepal_width` where the species is not clearly distinguishable especially between *versicolour* and *virginica* (green and red).

Another way to think about this is to imagine the plot colours were removed (i.e. you did not know the species of each point) and you were to pick one point at random. How confident would you be in determining the species of iris in each of those plots?

1.1.4 Linear SVM

We will now train a linear SVM classifier on the training set and compute the accuracy on training and test datasets respectively using the `petal_length` and `petal_width` features. It is important

to set up the predictive model (i.e. the classifier) on the training set and test its performance on the test set. The purpose of predictive modelling is to create models that are able to make predictions based on future data. Hence it is important to keep training and test data separate and not to use test data for training predictive models.

[ ]:
```
clf = LinearSVC()

# Train the linear SVC using the training dataset
clf.fit(training[['petal_length','petal_width']], training['target'])

# Test the linear SVC accuracy using the testing dataset
print('Test accuracy =', clf.score(testing[['petal_length','petal_width']],␣
 ↪testing['target'])*100, '%')
```

We demonstrate the classification results visually by plotting the decision boundaries produced by the linear SVM in the following code. Different colored regions correspond to different classes.

[ ]:
```
ax = plot_decision_regions(training[['petal_length','petal_width']].values,␣
 ↪training['target'].values, clf=clf, legend=2)
ax.set_title('Linear SVM')
ax.set_xlabel('Petal Length')
ax.set_ylabel('Petal Width')
```

A linear SVM can also be trained by using all input features. In many cases, this will produce better test results than using a subset of features.

[ ]:
```
clf = LinearSVC()
clf.fit(training[['sepal_length', 'sepal_width', 'petal_length',␣
 ↪'petal_width']], training['target'])
print('Test accuracy =', clf.score(testing[['sepal_length', 'sepal_width',␣
 ↪'petal_length', 'petal_width']], testing['target'])*100, '%')
```

As can be seen, we achieve better test accuracy with linear SVM trained on all features compared to the previous result using only feature 2 and feature 3. Note that only test performances are important for prediction modelling. (Why?)

1.2 Two-Moons Dataset

SVM classifiers can also be used to deal with nonlinear data. In the following example, we will show this using a dataset in which the two classes of data form interlocking moon shapes. First run the following block of code to generate and plot the dataset.

[ ]:
```
training_moon = pd.read_csv('moon_training.csv')
testing_moon = pd.read_csv('moon_testing.csv')
print training_moon[:3]
```

[ ]:
```
sns.lmplot(x='A', y='B', data=training_moon, hue='target', fit_reg=False)
```

This is a binary classification problem, where points from each class form a moon shaped cloud

opening in opposite direction, as shown in different colours in the above figure. Obviously, points from two classes can not be separated with a linear classifier. Next we train and test this problem with both linear SVM and kernel SVM.

[ ]:
```
# give an easier to read name to the datasets
training_data = training_moon[['A','B']]
training_target = training_moon['target']
testing_data = testing_moon[['A','B']]
testing_target = testing_moon['target']


# train and test linear svm classifier
clf_linear = LinearSVC()
clf_linear.fit(training_data, training_target)
clf_linear_accuracy = clf_linear.score(testing_data, testing_target)*100
print('Accuracy of Linear SVM =', clf_linear_accuracy, '%')

# train and test kernel svm classifier
clf_kernel = SVC()
clf_kernel.fit(training_data, training_target)
clf_kernel_accuracy = clf_kernel.score(testing_data, testing_target)*100
print('Accuracy of Kernel SVM =', clf_kernel_accuracy, '%')
```

As expected, the kernel SVM achieves better performance (higher accuracy) than the linear SVM for this example. Next, we plot the decision boundary of each classifier model.

[ ]:
```
ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf_linear, legend=2)
ax.set_title('Linear SVM')
ax.set_xlabel('A')
ax.set_ylabel('B')
```

[ ]:
```
ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf_kernel, legend=2)
ax.set_title('Kernel SVM')
ax.set_xlabel('A')
ax.set_ylabel('B')
```

As you can see from the above demonstration, the nonlinear kernel SVM produces a nonlinear decision boundary (i.e. a curve) to separate points from two classes, where regions that belong to different classes are shown in different colors. In contrast, linear SVM produces a linear decision boundary (i.e. a line) to separate points from two classes, which is not good enough for this example.

Despite its effectiveness, kernel SVM is slower than linear SVM in training. We demonstrate this by training both linear and kernel SVM classifiers 3 * 100 times and measuring the average time it takes to train each model.

```
[ ]: clf_linear = LinearSVC()
print('Linear SVM: ')
%timeit -n 100 -r 3 clf_linear.fit(training_data, training_target)

clf_kernel = SVC()
print('Kernel SVM: ')
%timeit -n 100 -r 3 clf_kernel.fit(training_data, training_target)
```

So far, we have used the default parameters for training and testing the kernel SVM classifier. The performance can be further improved with parameter tuning, specifically the regularisation parameter C plays an important role in final performance. We show how training and testing accuracies are affected by kernel SVM models trained with different C values.

```
[ ]: clist = 2**np.array(range(-2, 10), dtype='float')
traccuList = []     # list of training accuracies
tsaccuList = []     # list of testing accuracies
# train and test SVM for each c value in clist
for c in clist:
    # train a kernel SVM with C parameter value equal to c
    clf = SVC(C=c)
    clf.fit(training_data, training_target)
    # calculate the training and testing accuracies with given c value
    # and add them to the corresponding lists
    traccuList.append(clf.score(training_data, training_target)*100)
    tsaccuList.append(clf.score(testing_data, testing_target)*100)
```

```
[ ]: # plot the training accuracy values for different values of parameter C
# C values are in log-scale, i.e. -1 for 0.5, 0 for 1, i for 2**i
sns.plt.plot(np.log2(clist), traccuList, 'o-', label='Training Accuracy')

# plot the testing accuracy against C values
sns.plt.plot(np.log2(clist), tsaccuList, 'o-', label='Testing Accuracy')

# change some plot parameters and display result
sns.plt.xticks(np.log2(clist), [str(x) for x in clist])
sns.plt.legend(loc='upper left')
sns.plt.show()
```

Both training and testing performances are affected by the choice of parameter C. Increasing the value of C leads to an improvement in training performance, but not necessarily in testing performance because of overfitting. In practice, we can't tune the parameters based on testing-set accuracy and have to do this on the training dataset. (Why?) To solve this problem, we need to use cross-validation, an effective approach for parameter selection on the training dataset.

Next, we use k-fold cross-validation (`k=3`) to test the cross-validation accuracy for different C values and pick the best C that achieves the highest accuracy for cross-validation. We then retrain the kernel SVM classifier on the whole training dataset using the selected C value and test it on the testing dataset.

```
[ ]: from sklearn import cross_validation

clist = 2**np.array(range(-2, 10), dtype='float')
cvscores = []
for c in clist:
    clf = SVC(C=c)
    scores = cross_validation.cross_val_score(clf, training_data,␣
 ↪training_target, cv=3)
    cvscores.append(scores.mean()*100)
bestscore, bestC = max([(val, clist[idx]) for (idx, val) in␣
 ↪enumerate(cvscores)])
print('Best CV accuracy =', bestscore, 'achieved at C =', bestC)
# retrain on whole trainning set using best C value obtained from Cross␣
 ↪validation
clf = SVC(C=bestC)
clf.fit(training_data, training_target)
accu = clf.score(testing_data, testing_target)*100
print('Test accuracy =', accu, 'achieved at C =', bestC)
```

The accuracy value achieved by the kernel SVM classifier trained with the optimal parameter is higher than that produced with the kernel SVM classifier trained using the default parameter value. This demonstrates the importance and effectiveness of parameter selection.

2 Questions

Answer the questions below using the examples and code provided above. Double-click this cell and then edit the text below after each question.

1. How many features are there for the iris dataset? How many examples? How many labels?
2. Why is it important to split the dataset into training and test sets? Why does a classification model need to be trained on the training set and why does the prediction performance need to be measured on the test set?
3. How does correlation analysis can help identify the best features for the classification task? What are the best features for the iris data based on correlation analysis results?
4. Which class is easier to identify than the other two classes for the iris dataset? How can you tell?
5. Which classification model produces better test results for the iris data? Linear SVM trained on all features or linear SVM trained on the two best features? What does this tell you?
6. Why does linear SVM not produce a good result for the two-moons example?
7. Compare linear and kernel SVM in terms of predictive performance and training speed. What conclusions can you make?
8. Why do we need to perform parameter selection in training classification models for predictive modelling?

9. Why can't we choose the classifier parameter that produces the best training performance?
10. What is cross validation and why it is an effective technique for parameter selection in classifier training?

3 Coding task: Construct a classifier to predict handwritten digits using images

We use the `sklearn` package's `digits` data set. The following helps you understand the data.

[ ]:
```
from sklearn.datasets import load_digits
import numpy as np
digits = load_digits()

print('The number of images is: '+ str(digits.data.shape[0]))
print('The number of pixels in an image is: '+ str(digits.data.shape[1]))
print('The targets/labels are: '+ str(np.unique(digits.target)))

import matplotlib.pyplot as plt
plt.gray()
plt.matshow(digits.images[100]) #Show the 101st image in the data set.
plt.show()
```

[ ]:
```
# Show all 10 digits
for i in range(10):
    plt.subplot(1, 10, i + 1)
    plt.imshow(digits.images[i], cmap=plt.cm.gray, vmax=16,␣
 ↪interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
plt.show()
```

The data is in `digits.data` and the labels are in `digits.target`. Images have been "squashed" to vectors and stored in `digits.data`. Construct a classifier to able to predict digits for a given new image. Please refer to the demonstration code blocks above if you have difficulties.

3.0.1 Your code goes here:

[ ]:
```
# Python code for digits classification
```

10-Practical

July 18, 2020

1 Practical 10 - Predictive Modelling (Part II)

The objective is to practice on tree-based and ensemble models and identify their strengths and weaknesses in application in comparison to SVM-based models discussed in the lecture. You will also have some hands on experience on regression models that were covered in the lectures espcially the sparse linear regression. Finally you will experience the importance of feature pre-processing for practical classification tasks.

For this practical, again your task is to review this notebook. All code blocks can be executed if there is no instruction asking for your code. You do need to read the notebook carefully and test everything by yourself. Make sure you understand everything before continuing with the questions at the end and the final coding task.

N.B. all code blocks have to be excuted in sequel. If you encounter any difficulties, run the code from the very first block. In the final coding task, ***DO NOT*** *assume any preloaded packages or data set other than those appear in coding task blocks!*

1.1 Classification (cont.)

You will also need two moons training and test data (`moon_training.csv` and `moon_testing.csv` used last week. Please find them in last week prac class folder on vUWS.

Much like last week, you will not be required to write any code to perform the analysis. Instead, you will have to understand how it works and answer questions based on it and its outputs.

We start with loading packages for our analysis.

[8]:
```
%matplotlib inline
from sklearn.datasets import make_moons
from sklearn.svm import LinearSVC, SVC
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
#from mlxtend.evaluate import plot_decision_regions
#from mlxtend.plotting import plot_decision_regions
import mlxtend.plotting.plot_decision_regions as plot_decision_regions
```

```
        File "/home/nfht/anaconda2-new/lib/python2.7/site-packages/mlxtend/
↪plotting/heatmap.py", line 74
```

```
        raise AssertionError(f'len(row_names) (got {len(row_names)})'
                                                                    ^
    SyntaxError: invalid syntax
```

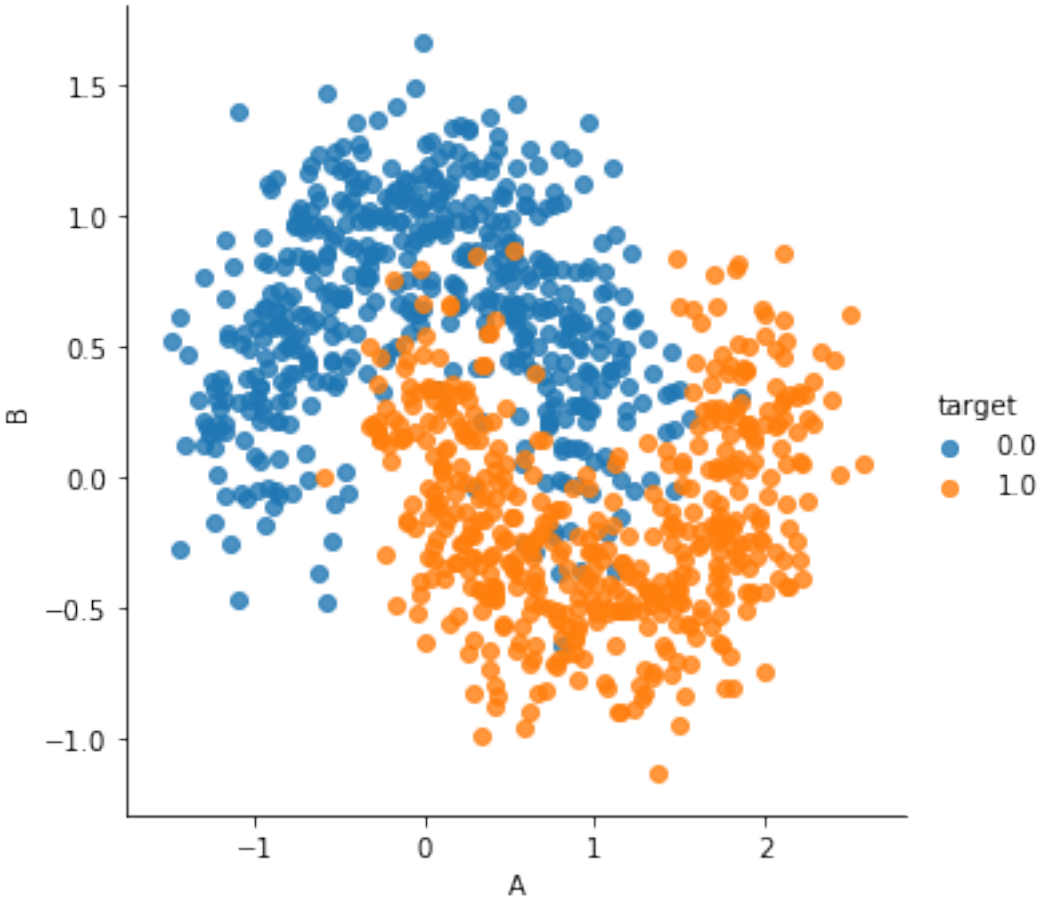
1.2 Two Moons Dataset

Here we are using the same two moons example from last week. Ensure that `moon_training.csv` and `moon_testing.csv` are in the same path as this notebook, or adjust the file paths in the cell below as necessary.

[5]:
```
training_moon = pd.read_csv('moon_training.csv')
testing_moon = pd.read_csv('moon_testing.csv')
```

[6]:
```
sns.lmplot(x='A', y='B', data=training_moon, hue='target', fit_reg=False)
```

[6]:
```
<seaborn.axisgrid.FacetGrid at 0x7f239bea3110>
```



For the sake of convenience, we only showed the training dataset above, but testing dataset can be plotted similarly .

In the following code block, we train a kernel SVM model to separate examples from two classes with different colors and also plot the decision boundary.

[9]:
```
# give an easier to read name to the datasets
training_data = training_moon[['A','B']]
training_target = training_moon['target']
testing_data = testing_moon[['A','B']]
testing_target = testing_moon['target']

# train and test kernel svm classifier
clf_kernel = SVC()
clf_kernel.fit(training_data, training_target)
clf_kernel_accuracy = clf_kernel.score(testing_data, testing_target)*100
print('Accuracy of Kernel SVM =', clf_kernel_accuracy, '%')

ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf_kernel, legend=2)
ax.set_title('Kernel SVM')
ax.set_xlabel('x')
ax.set_ylabel('y')
```

```
('Accuracy of Kernel SVM =', 93.4, '%')
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-9-f13402b725bb> in <module>()
        11 print('Accuracy of Kernel SVM =', clf_kernel_accuracy, '%')
        12
   ---> 13 ax = plot_decision_regions(training_data.values, training_target.
↪values.astype(int), clf=clf_kernel, legend=2)
        14 ax.set_title('Kernel SVM')
        15 ax.set_xlabel('x')


       NameError: name 'plot_decision_regions' is not defined
```

Next, we experiment with the decision tree classifier, which recursively partitions the space such that the samples with the same labels are grouped together. In this case, it recursively splits the space into two regions along the x and y axis. The depth of the tree, i.e. the maximum number of splits allowed, controls the complexity of the tree classifier. The deeper the tree, the more levels

and splits, the more complex the tree classifier, but it is also more likely to overfit the training data. On the other hand, a shallow tree with small depth may not be complex enough to deal with the non-linearity in the data.

First, we test a decision tree with depth 1 (just one cut along x or y direction). This is expected to produce poor result. Try executing the following code block **multiple times** and observe the results.

[ ]:
```
from sklearn.tree import DecisionTreeClassifier

# create a decision tree with depth 1, max_depth=1
print('Training decision tree with depth 1')
clf_tree = DecisionTreeClassifier(max_features=1, max_depth=1)
clf_tree.fit(training_data, training_target)
print('Accuracy = %.2f%%'  % (clf_tree.score(testing_data, testing_target)*100))

ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf_tree, legend=2)
ax.set_title('Decision Tree (d=1)')
ax.set_xlabel('x')
ax.set_ylabel('y')
```

Unlike the SVM classification model, which obtains quite stable results over multiple runs, *the decision tree has a random nature and may obtain different results for different runs.* For above example, we have set the maximum number of features to best tested during the split process to 1 (max_features=1). This means, for different runs, a different direction might be chosen for spliting the feature space. Hence, you can see that sometimes the space is splitted vertically (along x direction), and sometimes is splitted horizontally. You can simple use the default value for max_features, i.e `max_features=None`, which basically takes all features available and this problem will disappear.

By increasing the depth of the tree classifier, we can produce more sophisticated decision boundaries and achieve improved accuracies. The following code demonstrates this by testing different tree depths of 2, 4, 8 and 16.

[ ]:
```
# test depths = 2, 4, 8, 16
for depth in [2,4,8,16]:
    print('Training decision tree with depth %d' % depth )
    clf = DecisionTreeClassifier(max_features=1, max_depth=depth,␣
 ↪min_samples_leaf=1)
    clf = clf.fit(training_data, training_target)
    print('Accuracy = %.2f%%' % (clf.score(testing_data, testing_target)*100))
    plt.clf()
    ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf, legend=2)
    ax.set_title('Decision Tree (d=%d)'%depth)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    plt.show()
```

Though decision tree classifier gets more powerful by increasing the depth, the decision boundaries generated by the classification model also becomes quite spiky as can be observed from the above demonstration. This is mainly because the tree classifier is limited to axis-parallel directions for each split, and potentially many splits are needed to model complex data distrbutions making the resulting decision boundary excessively spiky. *This usually indicates overfitting as the trees are developed to model every data sample.* To overcome this issue and produce smoother boundaries, we can use some heuristics, such as tree pruning (not implemented with scikit-learn package though) and early stopping (do not split the region if it contains less than a threhold number of examples. The threshold can be set with *min_samples_leaf* parameter). The *min_samples_leaf* parameter value is 1 by default. Try setting this parameter to larger values for the above example and check the differences in results.

Moreover, by executing the above block multiple times, you might notice that the performance of decision tree classifier does not always improve with increasing depth. Sometimes overfitting occurs with larger depth. Technically, we need to select the optimal depth for decision tree classifier using the cross validation technique discussed during the lecture and practical class last week.

A more principled approach for achieving smoother decision boundaries and further improvement in performance is to use an ensemble model - a classification model that combines individual classifiers. The **random forest classifier**, which is an ensemble model for a collection of decision trees, is based on the decision tree but more powerful due to the use of multiple trees.

We test the performance of random forest classifier on the two moons dataset in the following cell.

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

num_trees = 100
print('Training random forest classifier')
clf_forest = RandomForestClassifier(n_estimators = num_trees,␣
 ↪min_samples_leaf=1, max_features=1, max_depth=16)
clf_forest.fit(training_data, training_target)
print('Accuracy = %.2f%%' % (clf_forest.score(testing_data,␣
 ↪testing_target)*100))
# plot decision boundary
ax = plot_decision_regions(training_data.values, training_target.values.
 ↪astype(int), clf=clf_forest, legend=2)
ax.set_title('Random Forest (%d trees)'%num_trees)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

As you can see, the random forest classifier produces quite smooth decision boundary compared to decision trees. Sometimes its classification performance is even better than kernel SVM! Furthermore, it is much more efficient for training random forest classifiers compared to kernel SVM. From the previous practical, we know that kernel SVM is slower than the linear SVM due to the nonlinear decision function it solves. In the next example, we will show that random forest classifier is even faster than linear SVM for large problems, making it a very useful classification model for practical applications.

The performance of random forest classifier depends on the number of trees used in the forest. This

can be set with the *n_estimator* parameter (10 by default). We have used 100 trees in the above example. However, the results are not overly sensitive with this parameter as long as sufficient number of trees is used. Moreover, unlike decision tree classifiers which overfit with large depth value, random forest classifiers are unlikely to overfit with large number of trees due to the use of the ensemble (collection). This is also a desirable feature of random forest classifier. In practice, we can just set *n_estimator* to a large value and let the classification model do the magic for us. To show this, you can try different values of *n_estimator* for the above example and observe the results. Notice, however, that a large value for the *n_estimator* parameter does make it slower for training and prediction.

1.3 Regression

Before we move onto regression, we sort out connection to `R` first. Here we assume you have `R` installed on your computer. If you don't, please go to http://r-project.org and install a corret version of `R` for your computer platform. Or alternatively, you can simply use hadoop-01.scem.uws.edu.au, the big data machine for this practical. If you use big data machine, you can skip the following installation part and go straight to next code block for regression.

In python there is a package called `rpy2` which can connect to `R` so that you can do amazing things like using `R` commands and `R` data sets. If you are using python 3 or later version, you can simply use

```
pip install rpy2
```

to install the package. However, if you use python 2.7.x, then you need to specify the version number to suit the python version as

```
pip install rpy2==2.7
```

Canopy does not have `rpy2` package. So you need to use command window (for windows) or canopy terminal (for Mac/Linux) to install this package.

When you sort out `rpy2` package, it is time to move on to the code blocks below.

[ ]:
```
import pandas as pd
import rpy2.robjects as ro
import rpy2.robjects.conversion as conversion
from rpy2.robjects import pandas2ri
pandas2ri.activate()

R = ro.r
# We play with cars data set in R
cars = conversion.ri2py(R['cars'])
print(cars.head())
```

You should see the first a few lines of `cars` data set, which is records about speed and stop distance. Now we proceed to apply ordinary least squares to simple linear regression and predict stop distance using speed.

[ ]:
```
from sklearn import linear_model
from sklearn.model_selection import train_test_split
```

```
y = np.array(cars['dist'].astype('float'))
x = np.array(cars['speed'].astype('float'))

x_train, x_test, y_train, y_test = train_test_split(x,y,
test_size=0.4,random_state=42)

reg = linear_model.LinearRegression(fit_intercept=True)
reg.fit(x_train.reshape(-1, 1),y_train)
y_pred = reg.predict(x_test.reshape(-1, 1))

plt.figure(figsize=(8,4))
plt.scatter(x_test, y_test,  color='black')
plt.plot(x_test, y_pred, color='blue', linewidth=3)
plt.xlabel('Speed')
plt.ylabel('Stopping distance')
plt.grid(True)
plt.title('Linear Regression Ordinary Least Squares')
plt.legend(['Model prediction','Observations'])
plt.show()
```

Stopping distance is a continuous quantity, so regression is the appropriate model for this. From the plot above it shows clearly that linear regression does a pretty good job in getting the stopping distance right.

Now we look at `diabetes` data which is included in python package `sklearn`. In `diabetes` data, ten baseline variables, *age, sex, body mass index, average blood pressure, and six blood serum measurements* were obtained for each of $n = 442$ diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline. The research question is to find the relationship between baseline variables (observables) and the disease progression and in turn we can predict the disease progression. One more important question is that actually the progression is *only determined by a couple of observable variables*.

[ ]:
```
from itertools import cycle
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import lasso_path, enet_path
from sklearn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target
```

[ ]:
```
print 'The dimension of diabetes data: '+ str(X.shape) + '\nFirst 3 rows:\n'
print X[range(3),:]
```

We explore the data a little bit by pairwise scatter plot.

```
[ ]: import seaborn as sns
df = pd.DataFrame(np.column_stack((X, y)),columns=['age', 'sex', 'body mass␣
 ↪index', 'average blood pressure',␣
 ↪'s1','s2','s3','s4','s5','s6','Progression'])
sns.pairplot(df)
plt.show()
```

We apply LASSO to the data, a sparse linear regression as the following. Note that the regularisation parameter, i.e. λ in the LASSO model, is arbitarily chosen to be 1.5, it is `alpha` in function `linear_model.Lasso`. So we end up of selecting **2** varaibles.

```
[ ]: varnames = ['age', 'sex', 'body mass index', 'average blood pressure',␣
 ↪'s1','s2','s3','s4','s5','s6']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.
 ↪4,random_state=42)
reg = linear_model.Lasso(alpha = 1.5,normalize=True)
reg.fit(X_train,y_train)
print('Selected variables: ',[varnames[i] for i in np.where(reg.coef_!=0)[0]])
```

But why in particular only 2 varialbes selected in the model? Why not 3, 4 or whatever? This question is essentially a model selection question. In general there should be a true model with a number of variables in the model that dicates the response. However, we never know this true model. There are two issues in the question. The first is how many variables is actually in the model. The second is what are they. Path algorithms can answwer these two in one go. They vary the regularisation parameter(s), do the sparse regression and output the varaibles selected as regularisation progress. Of course, this is done clearly in the path algorithms rather than applying sparse regression on a grid of regularisation parameters.

The following code demonstrates two path algorithms, LASSO path algorithm and elastic net path algorithm and their corresponding nonnegative version, i.e. constraining the regression coefficients to be nonnegative. The code plots the path. The vertical lines in the plots show the places where the set of selected variables changes.

```
[ ]: X /= X.std(axis=0)  # Standardize data (easier to set the l1_ratio parameter)

# Compute paths

eps = 5e-3  # the smaller it is the longer is the path

print("Computing regularization path using the lasso...")
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, eps, fit_intercept=False)

print("Computing regularization path using the positive lasso...")
alphas_positive_lasso, coefs_positive_lasso, _ = lasso_path(
    X, y, eps, positive=True, fit_intercept=False)
print("Computing regularization path using the elastic net...")
alphas_enet, coefs_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, fit_intercept=False)
```

```
print("Computing regularization path using the positive elastic net...")
alphas_positive_enet, coefs_positive_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, positive=True, fit_intercept=False)

# Display results

plt.figure(1)
ax = plt.gca()

colors = cycle(['b', 'r', 'g', 'c', 'k'])
neg_log_alphas_lasso = -np.log10(alphas_lasso)
neg_log_alphas_enet = -np.log10(alphas_enet)
for coef_l, coef_e, c in zip(coefs_lasso, coefs_enet, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_enet, coef_e, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and Elastic-Net Paths')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'Elastic-Net'), loc='lower left')
plt.axis('tight')


plt.figure(2)
ax = plt.gca()
neg_log_alphas_positive_lasso = -np.log10(alphas_positive_lasso)
for coef_l, coef_pl, c in zip(coefs_lasso, coefs_positive_lasso, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_positive_lasso, coef_pl, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and positive Lasso')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'positive Lasso'), loc='lower left')
plt.axis('tight')


plt.figure(3)
ax = plt.gca()
neg_log_alphas_positive_enet = -np.log10(alphas_positive_enet)
for (coef_e, coef_pe, c) in zip(coefs_enet, coefs_positive_enet, colors):
    l1 = plt.plot(neg_log_alphas_enet, coef_e, c=c)
    l2 = plt.plot(neg_log_alphas_positive_enet, coef_pe, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
```

```
plt.title('Elastic-Net and positive Elastic-Net')
plt.legend((l1[-1], l2[-1]), ('Elastic-Net', 'positive Elastic-Net'),
           loc='lower left')
plt.axis('tight')
plt.show()
```

The above shows that path algorithms are capable of finding all sets of selected variables with different sizes. But which set is the truth? This remains an open problem. Current solution is cross validation.

1.4 Feature Preprocessing: Census Income Example

In this exercise, we will work on a census income example to predict wether a person makes over $50,000K a year based on census information such as age, workclass, years of education, marital status, etc. The dataset can be downloaded from the vUWS site that contains a training data file **adult.trn** and a testing data file **adult.tst**. Copy these two data files to where this notebook is being saved and load the datasets by the following statements

[ ]:
```
print('Reading datasets...')
df_trn = pd.read_csv('adult.trn', index_col=False, skipinitialspace=True)
df_tst = pd.read_csv('adult.tst', index_col=False, skipinitialspace=True)
print('Done')
```

[ ]:
```
# inspect the data
df_trn
```

The training data (*df_trn*) contains 30162 rows and 13 columns. Each row corresponds to a training example and each column maps to an input feature, except the last column (salary) which represents the target label value to be predicted. This is a typical binary classification problem. The purpose is to predict whether a person makes >50K given the census information.

Since the input features are mixed with columns of numbers and strings (the latter is known as nominal features), we first have to convert the nominal features to numbers in order to use existing classification methods. This can be done with two encoding schemes. The first one is a standard **substitution scheme** that replaces each unique string value to a different number like 0, 1, 2, etc. The second one uses the more sophistated strategy based on **one-of-K encoding** (sometimes called one-hot encoding), that uses a K-vector to represent a single nominal feature with K different values. Each location in the K-vector correponds to one of the values and is set to 1 if the feature takes that value and 0 otherwise.

To illustrate this point, we use the race feature as an example. To get a list of unique values for race feature, we can use the following command

[ ]:
```
df_trn['workclass'].unique()
```

With the first scheme, we use **0, 1, 2, 3, 4** to represent **White, Black, Asian-Pac-Islander, Amer-Indian-Eskimo** and **Other** respectively. With the second scheme, we use **[1,0,0,0,0]** to represent **White**, **[0,1,0,0,0]** to represent **Black**, etc. In the following, we will try both encoding schemes for training our classification models to identify their respective strengths and limitations.

1.4.1 Substitution Scheme

First, we will test the performances of different classification models with the substitution scheme for feature value conversion. The following code block generates two new data frames, **df_trn_subst** and **df_tst_subst**, that replace the old training data frame **df_trn** and testing data frame **df_tst** respectively, where all nominal features have been converted into numerical values. We use a class from Scikit Learn named `LabelEncoder` which accepts catagorical data and substitutes the values with integers. This class is intended to be used for labels (targets) but we use it here on features as a demonstration.

[ ]:
```
from sklearn.preprocessing import LabelEncoder
ALL_COLS = set(df_trn.columns)
NUMERIC_COLS = set(['age', 'fnlwgt', 'years-of-edu', 'capital-gain',␣
↪'capital-loss', 'hours'])  # cols with numerical data
CATAGORICAL_COLS = ALL_COLS - NUMERIC_COLS - set(['salary'])  # catagorical␣
↪columns, excluding the target (salary)
df_trn_subst = df_trn.copy()
df_tst_subst = df_tst.copy()
for column in df_trn.drop(NUMERIC_COLS, axis=1).columns:
    le = LabelEncoder()
    le.fit_transform(df_trn[column].append((df_tst[column]))) # Just to ensure␣
↪that LabelEncoder has seen all labels
    df_trn_subst[column] = le.transform(df_trn[column])
    df_tst_subst[column] = le.transform(df_tst[column])
```

We can now examine the content of the new dataframes processed by the simple substitution schem by (again, we only check the trainining dataset **df_trn_subst**, and testing dataset **df_tst_subst** can be checked in the same way)

[ ]:
```
df_trn_subst
```

It can be seen that all entries have been converted to numerical values. We can now apply the standard classification model to the above problem to fit the last column of target values to other columns of feature values.

Firstly, we try the SVM classifier discussed in last week's lecture. Due to the size of the problem (>30,000 examples), we use the linear SVM classifier for the purpose of efficiency. For output result, we show both predictive accuracy and training speed measured by the number of seconds it takes for model training.

Execute the next block multiple times to observe the results.

[ ]:
```
from time import time

print('Extracting data...')
# split the dataframes into the features and labels (i.e. the salary column)
training_features, training_labels = df_trn_subst.drop(['salary'], axis=1),␣
↪df_trn_subst['salary']
```

```
testing_features, testing_labels = df_tst_subst.drop(['salary'], axis=1),␣
↪df_tst_subst['salary']

print('Training Linear SVM classifier...')
start = time()
clf = LinearSVC()
clf.fit(training_features, training_labels)
end = time()

print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features,␣
↪testing_labels)*100, end-start))
```

As you can see, every time you execute the code for training linear SVM, the accuracy value achieved is different. Most time it produces accuracy value of >75%. Sometimes you get a very poor accuracy (<50%). Why does this happen?

This happens due to numerical instability caused by the different scales of input features. By spotting the values for df_trn_subst from the previous table, we see that some columns have large values (fnlwgt and capital-gain), whereas others have relatively smaller values. This causes problem to classifier training that involves using a numerical method. To solve this problem, we need to preprocess the data to ensure each column has the same scale of values. In the following, we applied linear SVM to pre-processed features, where each feature is scaled to have zero mean and unit standard deviation.

[ ]:
```
from sklearn.preprocessing import StandardScaler

# apply feature transformation so that values from each column are scaled to␣
↪zero mean and unit standard deviation
scaler = StandardScaler().fit(training_features)
training_features_scaled = scaler.transform(training_features)
testing_features_scaled = scaler.transform(testing_features)

print('Training Linear SVM on pre-processed features...')
start = time()
clf = LinearSVC()
clf.fit(training_features_scaled, training_labels)
end = time()
print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features_scaled,␣
↪testing_labels)*100, end-start))
```

As you can see from the above results, a linear SVM classification model trained on the pre-processed data achieves **better** results than direct application of the classifier on raw input data. The results are also quite stable. If you execute the above code block multiple times, the accuracy values obtained are all quite similar. This demonstrates the importance of feature pre-processing for both performance and stability.

Interestingly, tree-based classification models such as the decision tree classifier and the random forest classifier, which is an ensemble of decision trees, are not affected by the scale of input features. To show this, we train a random forest classifier with 100 decision trees for both raw and

pre-processed features in the following and compare their performances. We have empirically set the minimum sample size to 5 for each leave node (min_samples_leaf=5), and the maximum depth of each tree to 16 (max_depth=16). Feel free to play with these parameter values for performance comparison.

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

# create a random forest classifier
print('Training random forest classifier on raw features...')
start = time()
clf = RandomForestClassifier(n_estimators=100, min_samples_leaf=5,␣
 ↪max_features='auto', max_depth=16)

# train and test random forest classifier on raw input features
clf.fit(training_features, training_labels)
end = time()

print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features,␣
 ↪testing_labels)*100, end-start))

# train and test random forest classifier on pre-processed features
print('Training random forest classifier on pre-processed features...')
start = time()
clf.fit(training_features_scaled, training_labels)
end = time()
print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features_scaled,␣
 ↪testing_labels)*100, end-start))
```

The results with random forest classifier for different input are not much different. This indicates that tree-based classification models are not overly sensitive to different scales of input data.

1.4.2 One-Of-K Encoding Scheme

In the following, we test the one-of-K encoding scheme, which converts each column of nominal feature into K columns of numerical values representing K possible nominal feature values. The following cell uses a Pandas function named `get_dummies`, which converts catagorical values into indicator values. This is exactly what One-of-K encoding is. e.g. the *relationship* column will be converted into 6 columns: Not-in-family, Husband, Wife, Own-child, Unmarried, Other-relative with 1 stored in the column for the value that is true.

In order for `get_dummies` to work properly, we will need to temporarily combine the training and testing datasets so that no values are missed in the one-of-K encoding. This could occur if no samples in the training dataset contain a value that is present in the testing dataset e.g. if no one in the training dataset has a relationship status of *Wife*, then the one-of-K encoding will not make a column for the *Wife* value. We can do this with another Pandas function named `concat`.

[ ]:
```
# combine training and testing datasets temporarily for one-of-K encoding
# only encode the columns listed in CATAGORICAL_COLS
```

```
df_dummies = pd.get_dummies(pd.concat([df_trn, df_tst], axis=0),␣
 ↪columns=CATAGORICAL_COLS)

train_rows = df_trn.shape[0]  # number of training dataset rows

# split the dataset back into training and testing
df_train_oneofk = df_dummies.iloc[:train_rows, :]
df_test_oneofk = df_dummies.iloc[train_rows:, :]
```

Now you can view the new training data frame **df_train_oneofk** by typing it directly (**df_test_oneofk** can be shown in the same way). You can scroll the horizontal scroll bar in the bottom to view more columns.

Next, we train two classification models, the linear SVM and random forest classifier, respectively on the new data frames obtained above. From previous example, we know that numerical features must be pre-processed to have the same scale for linear SVM training, but there is no need to pre-process features for random forest.

[ ]:
```
# get training and testing examples and labels from respective data frames
training_features, training_labels = df_train_oneofk.drop(['salary'], axis=1),␣
 ↪df_train_oneofk['salary']
testing_features, testing_labels = df_test_oneofk.drop(['salary'], axis=1),␣
 ↪df_test_oneofk['salary']
```

[ ]:
```
# random forest
print('Learning random forest classifier on raw features...')
start = time()
clf = RandomForestClassifier(n_estimators = 100, min_samples_leaf=5,␣
 ↪max_features='auto', max_depth=16)
clf.fit(training_features, training_labels)
end = time()
print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features,␣
 ↪testing_labels)*100, end-start))
```

[ ]:
```
# SVM, need feature scaling
print('Learning SVM classifier on pre-processed features...')
# apply feature scaling
start = time()
scaler = StandardScaler().fit(training_features)
training_features_scaled = scaler.transform(training_features)
testing_features_scaled = scaler.transform(testing_features)
# training and testing SVM
clf = LinearSVC()
clf.fit(training_features_scaled, training_labels)
end = time()
print('Accuracy = %.2f%% in %.2f seconds' % (clf.score(testing_features_scaled,␣
 ↪testing_labels)*100, end-start)   )
```

It can be seen that classification models trained on features obtained from the One-Of-K encoding scheme achieve better performances than models trained with the substitution scheme with increasing accuracy values. This is more obvious with the linear SVM classification model, which obtains about almost 3% improvement on accuracy (84.8% vs 82.07%). Speedwise, One-Of-K encoding scheme is slower than substitution scheme in training, because more columns are generated with the one-of-K encoding which increases the number of input features. With both encoding schemes, random forest classifier is able to outperform the linear SVM model in terms of both training speed and accuracy, demonstrating the effectiveness and efficiency of tree-based classification models for this classification task.

1.5 Questions

Answer each of the questions below using the examples and code provided above. Double click this cell and then edit the text below after each question.

1. How does increasing the depth of the decision tree classifier affect its performance?
2. What is the problem with decision tree classifier when applying it applied to the two moons dataset? Does random forest classifier have the same problem? Why?
3. Try running the random forest classifier multiple times on the two moons dataset with different numbers of trees (e.g. by changing the value of the num_trees variable to 10, 20, 50, 100, 200, 500, 1000, 2000). How does increasing the number of trees affect the performances of random forest classifier?
4. How many training examples are there for the census income dataset? What are the features for the census data? What is the purpose of prediction?
5. For the substitution and one-of-K schemes for converting nominal features into numerical values, which conversion scheme is more effective for predictive modelling? Why?
6. Why we need to apply feature pre-processing to scale the input features into the same range for classification training? Which classification model is more sensitive to different feature ranges?
7. Which classification model achieves better result with the substitution scheme? How about the one-of-K scheme?
8. Select a classification model for each of the following scenario and justify your choices.
    a) The dataset contains 2,000 examples with 50 features. All features are numerical values.
    b) The dataset contains 2,000,000 examples with 50 features. Majority of them are numerical values.
    c) The dataset contains 20,000 examples with 50,000 features. All features are numerical values.
    d) The dataset contains 20,000 examples with 10 features. Majority of them are nominal features.

Please submit this Jupyter notebook to the tutor's email address.

2 Coding task: apply sparse linear regression to Boston Housing data.

Boston housing data is contained in `sklearn` package. See http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html also https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/Boston.html. The response variable is the median value of owner-occupied homes in $1000s. The following code load the data in and print the shape of the predictors, `X` and response `y`.

[ ]:
```
from sklearn.datasets import load_boston
boston = load_boston()
X = boston.data
y = boston.target
print(X.shape)
print(y.shape)
```

Use path algorithm to find all possible sets of selected variables. You can go further to do cross validation to select the best model. The normal practice is that when you selected some variables, refit those to the data with an ordinary linear regression and take performance statistics from there.

2.0.1 Your code goes from here:

[ ]:
```
# Your python code goes from here.
```

Submit your completed jupyter notebook to your tutor email address!