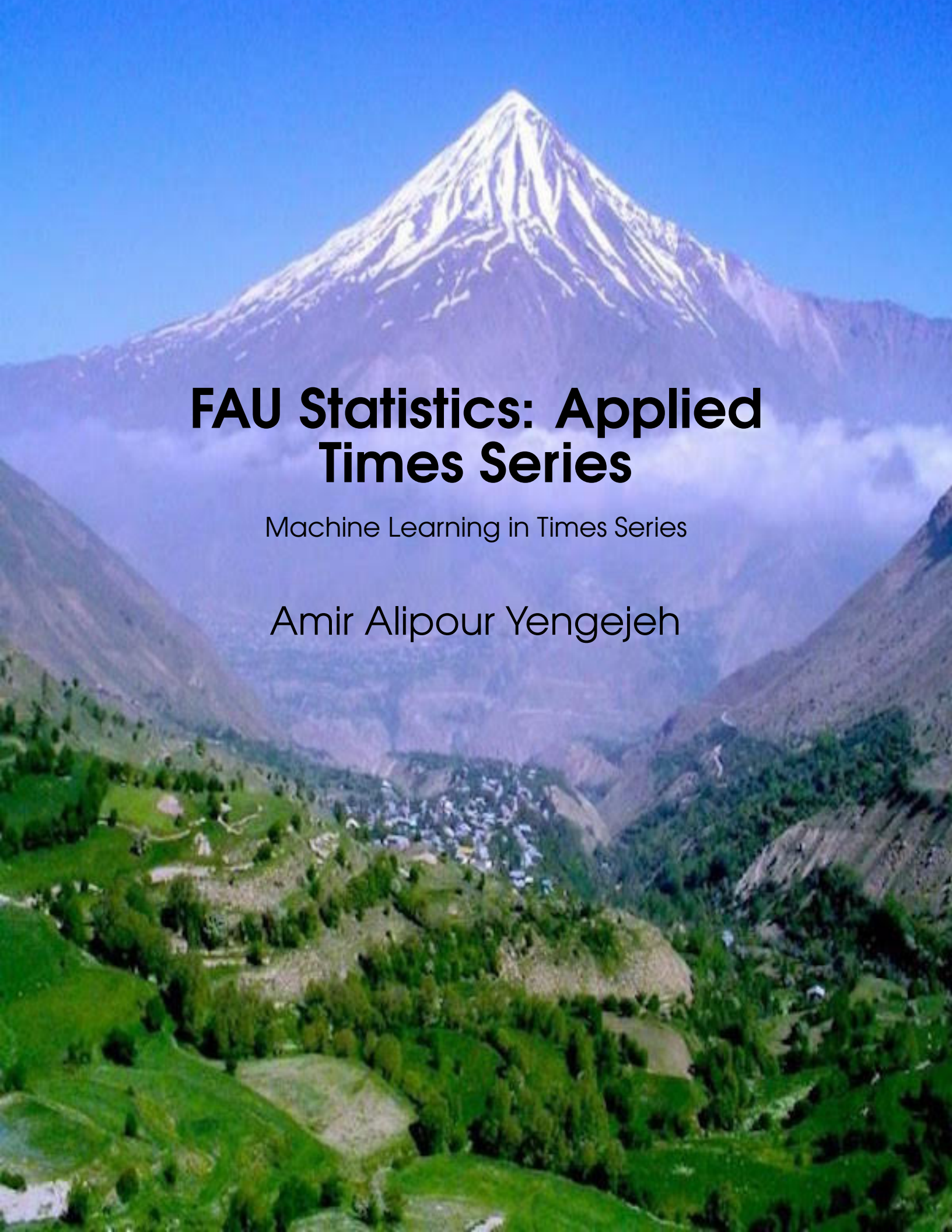


FAU Statistics: Applied Times Series

Machine Learning in Times Series

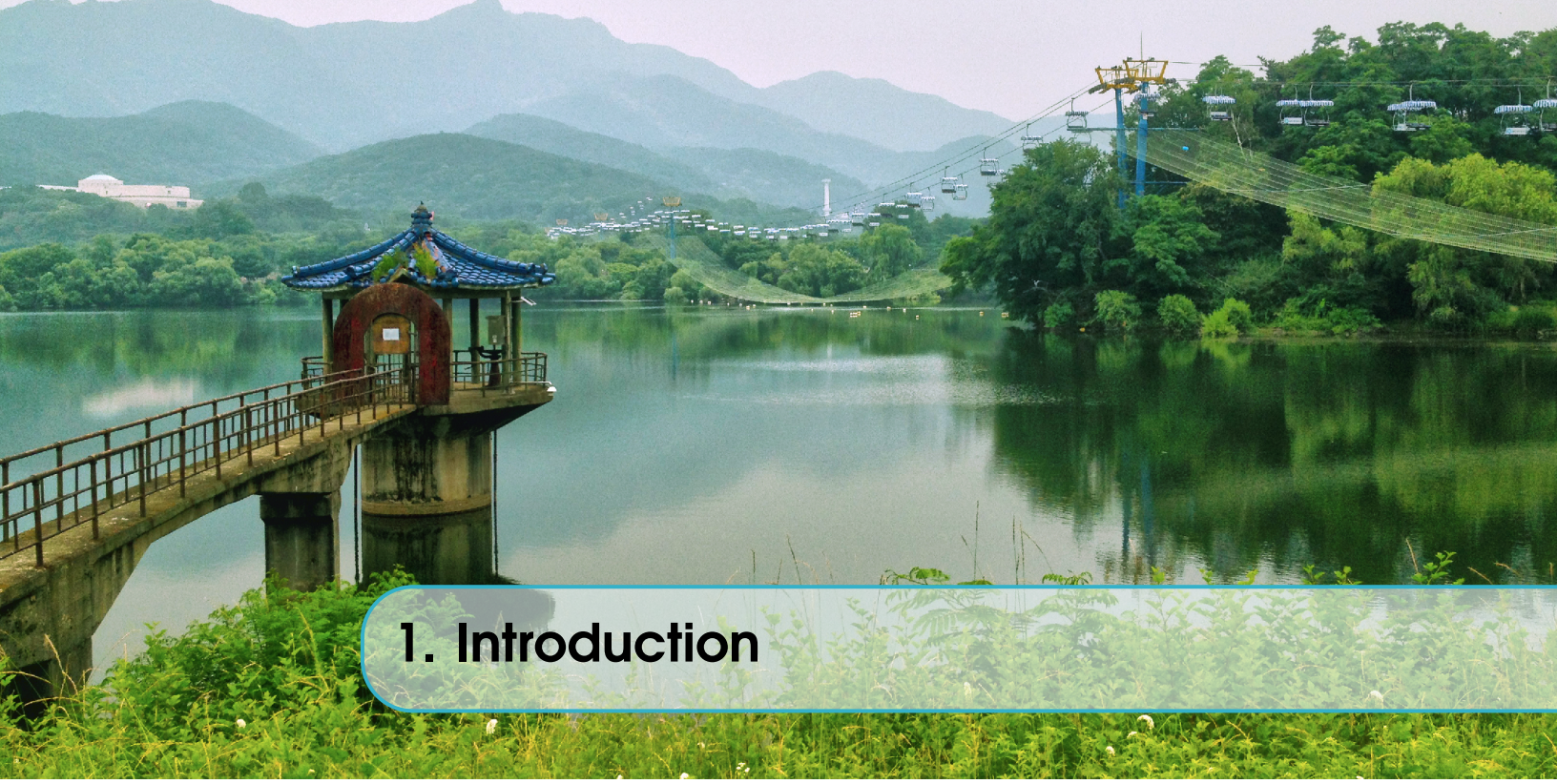
Amir Alipour Yengejeh

FAU STATISTICS: APPLIED TIMES SERIES

Professor: Dr. William Hahn

First release, April 2020

Contents



1. Introduction

The objective of this document is to introduce machine learning concept for analyzing of data are correlated to each other based on the time, known as Times Series, via Python environment.

Let make clear the concept through the following question: What do these applications have in common: predicting the electricity consumption of a household for the next three months, estimating traffic on roads at certain periods, and predicting the price at which a stock will trade on the New York Stock Exchange?

They all fall under the concept of time series data! You cannot accurately predict any of these results without the 'time' component. And as more and more data is generated in the world around us, time series forecasting keeps becoming an ever more critical technique for a data scientist to master.

Therefore, we start with understanding the internal structures of data that makes a time series, then we get familiar with a major concept of this literature called stationarity. It would proceed to predict or forecast the future values of a series based on the history of that series and, possibly, other related series or factors. In this regard, we will use the standard or classic forecasting times methods namely AR, MA, ARIMA, and the novel approaches such as advanced deep learning models such as Long Short Term Memory Networks (LSTM) by Pytorch as well as Keras. The dataset we are supposed to apply these method is Airline Passengers that we would like not only to predict the number of passengers for future but also learn how to prepare the data for obtaining the reliable results. Additionally, we will learn some programming techniques regarding Python such as loading, explanatory, and modifying methods .

At the book progresses, we will introduce some worthwhile and advanced applications of time series in deep learning. We would learn how to analysis of Audio data, Convolution Neural Network for time series, and using fast fourier transformation.

However, before diving into the time series analysis it is worthwhile to start introduction of python to let any reader to find a basic idea about this programming environment.

2. Getting ready for Python

2.1 What is Python and why should learn it?

Python is a widely used high-level programming language created by Guido van Rossum in the late 1980s. The language places strong emphasis on code readability and simplicity, making it possible for programmers to develop applications rapidly. Like all high level programming languages, Python code resembles the English language which computers are unable to understand. Codes that we write in Python have to be interpreted by a special program known as the Python interpreter, which we'll have to install before we can code, test and execute our Python programs. We'll look at how to install the Python interpreter

On the other hand, the main reason why we should learn Python is that there are a large number of high level programming languages available, such as C, C++, and Java. The good news is all high level programming languages are very similar to one another.

What differs is mainly the syntax, the libraries available and the way we access those libraries. If you learn one language well, you can easily learn a new language in a fraction of the time it took you to learn the first language. If you are new to programming, Python is a great place to start. One of the key features of Python is its simplicity, making it the ideal language for beginners to learn.

Installing the Interpreter Before moving ahead, I would recommend using Google Colab for doing everything related to Neural networks because it is free and provides GPUs and TPUs as runtime environments. Before we can write our first Python program, we can use free Google colab online environment instead of downloading and installing any interpreter on our computers . We'll be using Python 3 in this document because as stated on the official Python site "Python 2.x is legacy, Python 3.x is the present and future of the language". In addition, "Python 3 eliminates many quirks that can unnecessarily trip up beginning programmers".

2.2 Data Type

We have to skip introducing some basic parts such as variables and operators; however, we should look at other important basic data types in Python, specifically the integer, float and string. Next,

we'll explore the concept of type casting. Finally, we'll discuss three more advanced data types in Python: the list, tuple and dictionary.

Integer are numbers with no decimal parts, such as -5, -4, -3, 0, 5, 7 etc. To declare an integer in Python, simply write variableName = initial value

Float refers to numbers that have decimal parts, such as 1.234, -0.023, 12.01.

To declare a float in Python, we write variableName = initial value

Example: userHeight = 1.82, userWeight = 67.2

String refers to text.

To declare a string, you can either use variableName = 'initial value' (single quotes) or variableName = "initial value" (double quotes)

Example: userName = 'Peter', userSpouseName = "Janet", userAge = '30'

In the last example, because we wrote userAge = '30', userAge is a string. In contrast, if you wrote userAge = 30 (without quotes), userAge is an integer. We can combine multiple substrings by using the concatenate sign (+). For instance, "Peter" + "Lee" is equivalent to the string "PeterLee".

Type Casting In Python

Sometimes in our program, it is necessary for us to convert from one data type to another, such as from an integer to a string. This is known as type casting. There are three built-in functions in Python that allow us to do type casting. These are the int(), float(), and str() functions.

The int() function in Python takes in a float or an appropriate string and converts it to an integer. To change a float to an integer, we can type int(5.712987). We'll get 5 as the result (anything after the decimal point is removed). To change a string to an integer, we can type int ("4") and we'll get 4. However, we cannot type int ("Hello") or int ("4.22321"). We'll get an error in both cases.

The float() function takes in an integer or an appropriate string and changes it to a float. For instance, if we type float(2) or float("2"), we'll get 2.0. If we type float("2.09109"), we'll get 2.09109 which is a float and not a string since the quotation marks are removed. The str() function on the other hand converts an integer or a float to a string. For instance, if we type str(2.1), we'll get "2.1". Now that we've covered the three basic data types in Python and their casting, let's move on to the more advanced data types:

List refers to a collection of data which are normally related. Instead of storing these data as separate variables, we can store them as a list. For instance, suppose our program needs to store the age of 5 users. Instead of storing them as user1Age, user2Age, user3Age, user4Age and user5Age, it makes more sense to store them as a list. To declare a list, you write listName = [initial values]. Note that we use square brackets [ ] when declaring a list. Multiple values are separated by a comma.
Example: userAge = [21, 22, 23, 24, 25]

Tuple are just like lists, but you cannot modify their values. The initial values are the values that will stay for the rest of the program. An example where tuples are useful is when your program needs to store the names of the months of the year. To declare a tuple, you write tupleName = (initial values). Notice that we use round brackets ( ) when declaring a tuple. Multiple values are separated by a comma.

Example:

monthsOfYear = ("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec") You access the individual values of a tuple using their indexes, just like with a list. Hence, monthsOfYear[0] = "Jan", monthsOfYear[-1] = "Dec".

Dictionary is a collection of related data PAIRS. For instance, if we want to store the username and age of 5 users, we can store them in a dictionary. To declare a dictionary, you write dictionaryName = dictionary key : data, with the requirement that dictionary keys must be unique (within one

dictionary). That is, you cannot declare a dictionary like this myDictionary = “Peter”:38, “John”:51, “Peter”:13.

This is because “Peter” is used as the dictionary key twice. Note that we use curly brackets when declaring a dictionary. Multiple pairs are separated by a comma.

Example: userNameAndAge = “Peter”:38, “John”:51, “Alex”:13, “Alvin”:“Not Available”

2.3 Making Choices and Decisions

We will look at how to make your program smarter, capable of making choices and decisions. Specifically, we’ll be looking at the **if** statement, for **loop**, and **while loop**. These are known as control flow tools; they control the flow of the program.

However, before we go into these control flow tools, we have to first look at condition statements.

Condition Statements

All control flow tools involve evaluating a condition statement. The program will proceed differently depending on whether the condition is met. The most common condition statement is the comparison statement. If we want to compare whether two variables are the same, we use the == sign (double =). For instance, if you write x== y, you are asking the program to check if the value of x is equals to the value of y. If they are equal, the condition is met, and the statement will evaluate to **True**. Else, the statement will evaluate to **False**.

Other comparison signs include != (not equals), < (smaller than), > (greater than), <= (smaller than or equals to), and >= (greater than or equals to). The list below shows how these signs can be used and gives examples of statements that will evaluate to **True**.

Not equals: 5 != 2 Greater than: 5>2 Smaller than: 2<5 Greater than or equals to: 5>=2 5>=5 Smaller than or equals to: 2 <= 5 2 <= 2 We also have three logical operators, and, or, not that are useful if we want to combine multiple conditions.

If Statement

The *if* statement is one of the most commonly used control flow statements. It allows the program to evaluate if a certain condition is met, and to perform the appropriate action based on the result of the evaluation. The structure of an *if* statement is as follows: if condition 1 is met: do A elif condition 2 is met: do B elif condition 3 is met: do C elif condition 4 is met: do D else: do E elif stands for “else if” and you can have as many elif statements as you like.

To fully understand how the if statement works, fire up IDLE and key in the following code.

```

userInput = input('Enter 1 or 2: ')
if userInput == "1":
                print ("Hello World")
                print ("How are you?")
elif userInput == "2":
                print ("Python Rocks!")
                print ("I love Python")
else:
                print ("You did not enter a valid number")
```

Run the program three times, enter 1, 2 and 3 respectively for each run. You’ll get the following:

```

output:
Enter 1 or 2: 1
Hello World
```

```
How are you?
Enter 1 or 2: 2
Python Rocks!
I love Python
Enter 1 or 2: 3
You did not enter a valid number
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

```
x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
    if x == 10:
        print("x=10")
    else:
        print("invalid")
else:
    print("x=y")
```

```
x<y
x=10
```

An inline

if statement is a simpler form of an if statement and is more convenient if you only need to perform a simple task. The syntax is:

do Task A if condition is true else do Task B For instance,

num1 = 12 if myInt==10 else 13

This statement assigns 12 to num1 (Task A) if myInt equals to 10. Else it assigns 13 to num1(Task B).

Another example is

```
myInt = 10
print ("This is task A" if myInt == 10 else "This is task B")
```

This statement prints “This is task A” (Task A) if myInt equals to 10. Else it prints “This is task B” (Task B).

For Loop

Next, let us look at the for loop. The for loop executes a block of code repeatedly until the condition in the for statement is no longer valid.

Looping through an iterable

In Python, an iterable refers to anything that can be looped over, such as a string, list or tuple.

The syntax for looping through an iterable is as follows:

for a in iterable: print (a)

Example:

```

pets = ['cats', 'dogs', 'rabbits', 'hamsters']
for myPets in pets:
    print (myPets)
```

In the program above, we first declare the list pets and give it the members 'cats','dogs', 'rabbits' and 'hamsters'. Next the statement for myPets in pets: loops through the pets list and assigns each member in the list to the variable myPets.

The first time the program runs through the for loop, it assigns 'cats' to the variable myPets. The statement print (myPets) then prints the value 'cats'. The second time the programs loops through the for statement, it assigns the value 'dogs' to myPets and prints the value 'dogs'. The program continues looping through the list until the end of the list is reached.

If you run the program, you'll get cats dogs rabbits hamsters

We can also display the index of the members in the list. To do that, we use the enumerate() function.

```
for index, myPets in enumerate(pets):
      print (index, myPets)
```

This will give us the output 0 cats 1 dogs 2 rabbits 3 hamster It is also possible to iterate over a nested list illustrated below.

```
 list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list1 in list_of_lists:
        print(list1)

```

The output is:

```

[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

A use case of a nested for loop in this case would be,

```

list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list1 in list_of_lists:
    for x in list1:
        print(x)
```

It gives us:

```
1
2
3
4
5
6
7
8
9
```

While Loop

The next control flow statement we are going to look at is the while loop. Like the name suggests, a *while loop* repeatedly executes instructions inside the loop while a certain condition remains valid. The structure of a while statement is as follows:

while condition is true: do A

```
counter = 5
while counter > 0:
```

```
        print ("Counter =" , counter)
        counter = counter - 1
```

```
Counter = 5
Counter = 4
Counter = 3
Counter = 2
Counter = 1
```

At first look, a while statement seems to have the simplest syntax and should be the easiest to use. However, one has to be careful when using while loops due to the danger of infinite loops. Notice that in the program above, we have the line counter = counter - 1? This line is crucial. It decreases the value of counter by 1 and assigns this new value back to counter, overwriting the original value.

We need to decrease the value of counter by 1 so that the loop condition while counter > 0 will eventually evaluate to False. If we forget to do that, the loop will keep running endlessly resulting in an infinite loop. If you want to experience this first hand, just delete the line *counter = counter - 1* and try running the program again. The program will keep printing counter = 5 until you somehow kill the program. Not a pleasant experience especially if you have a large program and you have no idea which code segment is causing the infinite loop.

Another example is

```
i = 1
while i < 3:
    print(i ** 2)
    i = i + 1
print('Bye')
##output
1
4
Bye
```

Break

When working with loops, sometimes you may want to exit the entire loop when a certain condition is met. To do that, we use the break keyword. Run the following program to see how it works.

```
j = 0
for i in range(5):
    j = j + 2
    print ('i = ', i, ', j = ', j)
if j == 6:
    break
```

You should get the following output. i = 0 , j = 2 i = 1 , j = 4 i = 2 , j = 6

Without the break keyword, the program should loop from i = 0 to i = 4 because we used the function range(5). However with the break keyword, the program ends prematurely at i = 2.

This is because when i = 2, j reaches the value of 6 and the break keyword causes the loop to end.

In the example above, notice that we used an if statement within a for loop. It is very common for us to 'mix-and-match' various control tools in programming, such as using a while loop inside an if statement or using a for loop inside a while loop. This is known as a nested control statement.

Continue

Another useful keyword for loops is the continue keyword. When we use continue, the rest of the loop after the keyword is skipped for that iteration. An example will make it clearer.

```
j = 0
for i in range(5):
    j = j + 2
    print ('\ni = ', i, ', j = ', j)
    if j == 6:
      continue
    print ('I will be skipped over if j=6')

## Output:
i =  0 , j =  2
I will be skipped over if j=6

i =  1 , j =  4
I will be skipped over if j=6

i =  2 , j =  6

i =  3 , j =  8
I will be skipped over if j=6

i =  4 , j =  10
I will be skipped over if j=6
```

When j = 6, the line after the continue keyword is not printed. Other than that, everything runs as per normal.

2.4 Functions

Functions are simply pre-written codes that perform a certain task. For an analogy, think of the mathematical functions available in MS Excel. To add numbers, we can use the sum() function and type sum(A1:A5) instead of typing A1+A2+A3+A4+A5.

Depending on how the function is written, whether it is part of a class (a class is a concept in object-oriented programming) and how you import it, we can call a function simply by typing the name of the function or by using the dot notation. Some functions require us to pass data in for them to perform their tasks. These data are known as parameters and we pass them to the function by enclosing their values in parenthesis ( ) separated by commas.

For instance, to use the print() function for displaying text on the screen, we call it by typing print("Hello World") where print is the name of the function and "Hello World" is the parameter.

On the other hand, to use the replace() function for manipulating text strings, we have to type "Hello World".replace("World", "Universe") where replace is the name of the function and "World" and "Universe" are the parameters. The string before the dot (i.e. "Hello World") is the string that will be affected. Hence, "Hello World" will be changed to "Hello Universe".

Defining Your Own Functions

We can define our own functions in Python and reuse them throughout the program. The syntax for defining a function is as follows:

```
def functionName(parameters):
    code detailing what the function should do
    return [expression]
```

There are two keywords here, def and return.

def tells the program that the indented code from the next line onwards is part of the function. return is the keyword that we use to return an answer from the function. There can be more than one return statements in a function.

However, once the function executes a return statement, the function will exit. If your function does not need to return any value, you can omit the return statement. Alternatively, you can write return or return None.

Let us now define our first function. Suppose we want to determine if a given number is a prime number. Here's how we can define the function using the modulus (%) and the for loop and if statement we learned above.

```
1 def checkIfPrime (numberToCheck):
2     for x in range(2, numberToCheck):
3         if (numberToCheck%x == 0):
4                 return False
5     return True
```

In the function above, lines 2 and 3 uses a for loop to divide the given parameter numberToCheck by all numbers from 2 to numberToCheck - 1 to determine if the remainder is zero. If the remainder is zero, numberToCheck is not a prime number. Line 4 will return False and the function will exit.

If by last iteration of the for loop, none of the division gives a remainder of zero, the function will reach Line 5, and return True. The function will then exit. To use this function, we type checkIfPrime(13) and assign it to a variable like this

answer = checkIfPrime(13)

Here we are passing in 13 as the parameter. We can then print the answer by typing print(answer). We'll get the output: True.

2.5 Loading and reading files

The prerequisite for doing any data-related operations in Python, such as data cleansing, data aggregation, data transformation, and data visualisation, is to load data into Python. Depends on the types of data files (e.g. .csv, .txt, .tsv, .html, .json, Excel spreadsheets, relational databases etc.) and their size, different methods should be applied to deal with this initial operation accordingly. In this section, I will list some common methods for importing data in Python.

Python build-in functions (read(), readline(), and readlines())

In general, a text file (.txt) is the most common file we will deal with. Text files are structured as a sequence of lines, where each line includes a sequence of characters. Let's assume we need to import in Python the following text file (sampletext.txt).

```
1
2 Country/Region
3 Mainland China
4 Japan
5 Singapore
6 Hong Kong
7 Japan
8 Thailand
9 South Korea
10 Malaysia
11 Taiwan
12 Germany
13 Vietnam
```

```
France
Macau
UK
United Arab Emirates
US
Australia
```

Since here we are using Google colab, we should upload it and then follow the below commands:

```
with open("sampletext.txt","r") as reader:
  #read and print the entire file
  print(reader.read())
```

```
Country/Region
Mainland China
Japan
Singapore
Hong Kong
Japan
Thailand
South Korea
Malaysia
Taiwan
Germany
Vietnam
France
Macau
UK
United Arab Emirates
US
Australia
```

Figure 2.1: Reading a text file

2.3. Import data using Pandas

Another very popular option in importing data in Python must go to Pandas, especially when the data size is big (like several hundred MBs). We won't delve into the specifics of how pandas works or how to use it. There are many excellent tutorials and books (e.g. Python for Data Analysis, by Wes McKinney, creator of pandas). Here I just show some of the power of pandas in reading csv and excel files.

- *pd.read_csv()*: this reads a csv file into DataFrame object. An important point here is that pandas is smart enough to automatically tell the header row and data type of each field, which make the downstream analyse more efficient.

```
    import pandas as pd
    df = pd.read_csv('sample.csv')
    df
```

	Province/State	Country/Region	Last Update	Confirmed	Deaths	Recovered	City	Date_last_updated_AEDT	lat	lon
0	Hubei	China	02/15/2020 23:00	56249	1596	5623	NaN	2020-02-16 15:00:00	31.151725	112.878322
1	Guangdong	China	02/15/2020 23:00	1316	2	442	NaN	2020-02-16 15:00:00	23.135769	113.198269
2	Henan	China	02/15/2020 23:00	1231	13	415	NaN	2020-02-16 15:00:00	34.000000	114.000000

Figure 2.2: Reading a CSV file

- *pd.read_excel()*: this reads an excel file (.xls, .xlsx, .xlsm, .xlsb, and .odf file extensions) into a pandas DataFrame. By default, it only import the first data sheet within the excel file (sample.xlsx has multiple sheets), as shown in Figure below.

```
import pandas as pd
df = pd.read_excel('sample.xlsx')
df
```

	Province/State	Country/Region	Last Update	Confirmed	Deaths	Recovered	City
0	Hubei	Mainland China	2/16/2020 18:30	58182	1696	6639	NaN
1	Guangdong	Mainland China	2/16/2020 18:30	1316	2	465	NaN
2	Henan	Mainland China	2/16/2020 18:30	1231	13	440	NaN
3	Zhejiang	Mainland China	2/16/2020 18:30	1167	0	456	NaN
4	Hunan	Mainland China	2/16/2020 18:30	1004	3	464	NaN
...	...	...	...	...	...	...	...
65	Arizona	US	2/16/2020 18:30	1	0	0	Tempe
66	Brussels	Belgium	2/16/2020 18:30	1	0	0	NaN
67	WI	US	2/16/2020 18:30	1	0	0	Madison
68	Texas	US	2/16/2020 18:30	1	0	0	NaN
69	NaN	Egypt	2/16/2020 18:30	1	0	0	NaN

70 rows × 7 columns

Figure 2.3: Reading a Excel file

- read the url data: this reads any type of files has an online linke into a pandas DataFrame. It will illustrated in the following for a csv file:

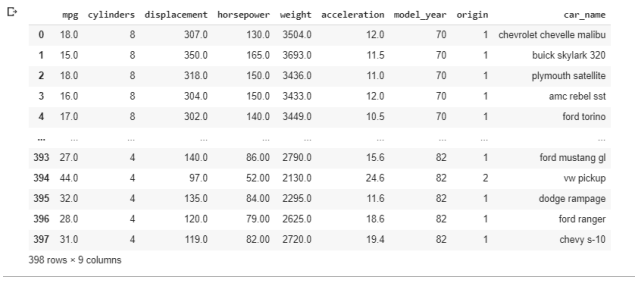
```
import pandas as pd
import urllib
import requests

web_url='http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data'

urllib.request.urlretrieve(web_url, "auto-mpg.data")

MPG=pd.read_csv('auto-mpg.data' ,  header=None, delim_whitespace=True, dtype={'3':float})

MPG.columns=['mpg', 'cylinders', 'displacement','horsepower', 'weight', 'acceleration', 'model_year', 'origin', 'car_name']
```



	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86.00	2790.0	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52.00	2130.0	24.6	82	2	vw pickup
395	32.0	4	135.0	84.00	2295.0	11.6	82	1	dodge rampage
396	28.0	4	120.0	79.00	2625.0	18.6	82	1	ford ranger
397	31.0	4	119.0	82.00	2720.0	19.4	82	1	chevy s-10

398 rows × 9 columns

Figure 2.4: Reading an Url file

4. Options for importing large size data

In the age of big data, sometimes, we need to import files from a client or colleague, which may be too large (gigabytes or terabytes) to load into memory. So what should we do to tackle this bottleneck? Fortunately, Pandas provides *chunksize* option to work this around. Essentially, we are not importing the whole file in one go instead of importing partial contents.

In addition, I found a very useful post by Mihail Yanchev, where he provided multiple methods and compared their efficiency in handling this situation. Here I just list those methods mentioned in his post and you can read his post if that is what you are looking for.

- **dask.dataframe**(): a large parallel DataFrame composed of many smaller Pandas DataFrames, split along the index. A good thing is that most functions used with pandas can also be used with dask.
- **datatable**: a Python package for manipulating big 2-dimensional tabular data structures (aka data frames, up to 100GB).

Before leaving this section, however, uploading and reading data from google drive when we are on Google Colab is worthy of notice. Since we can save our data on Google drive, we can mout drive on colab and read them:

```
1
2 from google.colab import drive
3 drive.mount('/content/drive')
4
5 ##Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?
    client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.
    googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&
    response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth
    %2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3
    a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2
    fwww.googleapis.com%2fauth%2fpeopleapi.readonly
6
7 Enter your authorization code:
8
9 Mounted at /content/drive
10
11 ## read the data
12
13 kc_house=pd.read_csv('/content/drive/My Drive/Data_storage/Regression Final
    Project Data/kc_house_data.csv')
```

2.6 Plotting and Visualization

Making informative visualizations (sometimes called plots) is one of the most important tasks in data analysis. It may be a part of the exploratory process—for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for making static or dynamic visualizations, but we'll be mainly focused on matplotlib and libraries that build on top of it.

matplotlib supports various GUI backends on all operating systems and additionally can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.)

Since there is not enough room in the cookbook to give a comprehensive treatment to the breadth and depth of functionality in matplotlib, we will give the ropes to get up and running. In addition,

the matplotlib gallery and documentation are the best resource for learning advanced features.

Let start how to use matplotlib for plotting simple data:

```
1
2  import matplotlib.pyplot as plt
3  import numpy as np
4  data = np.arange(10)
5  data
6  ##output
7  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
8
9  ## plot the data
10 plt.plot
```

```
[<matplotlib.lines.Line2D at 0x7f574bc48550>]
```

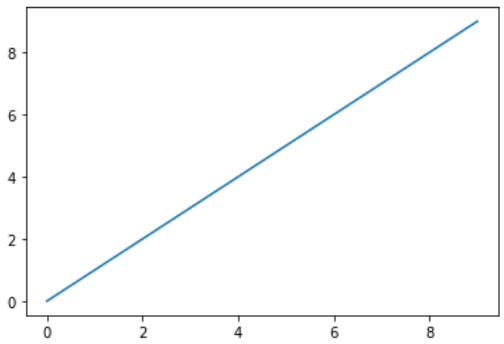


Figure 2.5: A simple line plot

In following three examples, however, we will focus on some advanced plots and how we can more arguments.

Example1:

Let first load and read the data

```
1 # Load a numpy record array from yahoo csv data with fields date, open,
      close,
2 # volume, adj_close from the mpl-data/example directory. The record array
3 # stores the date as an np.datetime64 with a day unit ('D') in the date
      column.
4 with cbook.get_sample_data('goog.npz') as datafile:
5     r = np.load(datafile)['price_data'].view(np.recarray)
6 r = r[-50:]  # get the last 50 day
7
8 # display the first 10 rows of dataset
9 r
```

We will get the below data

```
rec.array([('2008-10-01', 411.15, 416.98, 403.1 , 411.72,  6234800, 411.72),
           ('2008-10-02', 409.79, 409.98, 386.  , 390.49,  5984900, 390.49),
           ('2008-10-03', 397.35, 412.5 , 383.07, 386.91,  7992900, 386.91),
           ('2008-10-06', 373.98, 375.99, 357.16, 371.21, 11220600, 371.21),
           ('2008-10-07', 373.33, 374.98, 345.37, 346.01, 11054400, 346.01),
           ('2008-10-08', 330.16, 358.99, 326.11, 338.11, 11826400, 338.11),
           ('2008-10-09', 344.52, 348.57, 321.67, 328.98,  8075000, 328.98),
           ('2008-10-10', 313.16, 341.89, 310.3 , 332.  , 10597000, 332.  ),
           ('2008-10-13', 355.79, 381.95, 345.75, 381.02,  8905500, 381.02),
           ('2008-10-14', 393.53, 394.5 , 357.  , 362.71,  7784800, 362.71)],
          dtype=[('date', '<M8[D]'), ('open', '<f8'), ('high', '<f8'), ('low', '<f8'), ('close', '<f8'), ('volume', '<i8'), ('adj_close', '<f8')])
```

Figure 2.6: The yahoo file data

Now let plot the above data

```
# first we'll do it the default way, with gaps on weekends
fig, axes = plt.subplots(ncols=2, figsize=(8, 4))# Create two subplot to the current figure.
ax = axes[0] # set the fig in the fist frame
ax.plot(date, r.adj_close, 'o-') # plot the date vs values with "0" marker and link them with dash"-"
ax.set_title("Default") #set tittle for the plot
fig.autofmt_xdate() # set or arrange the x's tikers



# next we'll write a custom formatter
N = len(r) # define the length of dataset
ind = np.arange(N)  # the evenly spaced plot indices

#Define the format function for dates indices
def format_date(x, pos=None):
    thisind = np.clip(int(x + 0.5), 0, N - 1)
    return date[thisind].strftime('%Y-%m-%d')

ax = axes[1]# set on the second frame
ax.plot(ind, r.adj_close, 'o-') # Plot the indices vs values
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date)) # set the formatter of x tickers
ax.set_title("Custom tick formatter") # set title
fig.autofmt_xdate() # set the x tikers arrangment

plt.show() # show the plot (output)
```

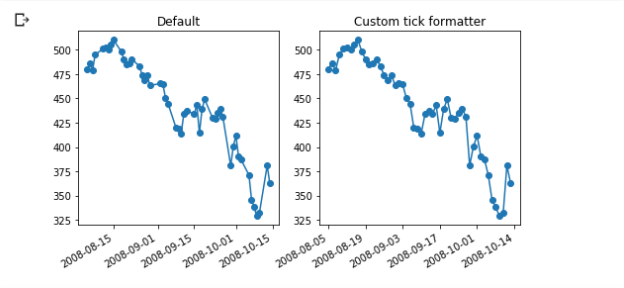


Figure 2.7: The plots

Example 2

The below figure shows a time series plot from an industrial chemical process in which a color property of consecutive batches in this process measured.

```

df2=pd.read_csv('/content/drive/My Drive/Data_Folder/Time Series/Cryer/color.csv', parse_dates=['batch'], squeeze=True)

df2.shape# (35, 2)


```

```
#plot
plt.figure(figsize=(9.5, 5.5))# set figure size
plt.plot(range(0,35), df2['color'], linestyle='-', marker='o',
color='b')# set plot data with defined arguments
plt.title('Time Series Plot of Color Property from a Chemical Process')# set
    title of plot
plt.ylabel('Color')# set label of y
plt.xlabel('Batch')# set the label of x
```

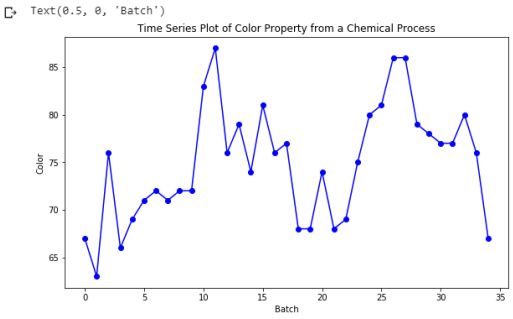


Figure 2.8: The times series plot for color properties

```
plt.figure(figsize=(9.5, 5.5))
plt.plot(range(0,35), df2['color'], linestyle='-', marker='o',
color='b')
plt.title('Time Series Plot of Color Property from a Chemical Process')
plt.ylabel('Color')
plt.xlabel('Batch')
```

Example3

The following figure displays the scatterplot of the neighboring pairs of color values.

```

series2 = pd.read_csv('/content/drive/My Drive/Data_Folder/Time Series/Cryer
    /color.csv', header=0, index_col=0, parse_dates=True, squeeze=True)#
    read the data
lag_plot(series2)# set a lag plot
pyplot.title('Scatterplot of Color Value versus Previous Color Value')# set
    a title
pyplot.xlabel('Batch')# set a label for x
pyplot.ylabel('Color') # set a label for y
pyplot.show() # show the plot
```

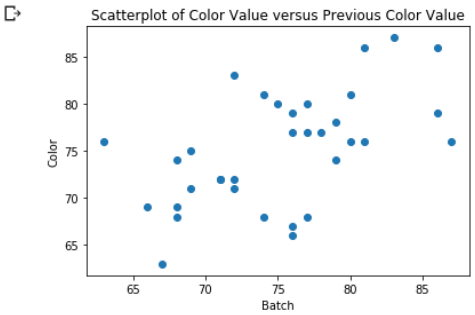


Figure 2.9: The scatter plot for color data

Example4

Here, we like to show how add legend the plot.

```
1
2 from numpy.random import randn # call random library for generating random
    data
3 fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)# set the frame of figure
    for 3 subplots
4 ax.plot(randn(1000).cumsum(), 'k', label='one',color='r')# set the the stile
     of line, label, and color for first random data
5 ax.plot(randn(1000).cumsum(), 'k--', label='two',color='b')
6 ax.plot(randn(1000).cumsum(), 'k.', label='three',color='g')
7 ax.legend(loc='best') # call to create legends
```

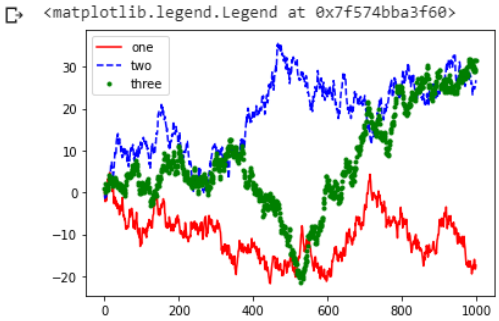


Figure 2.10: Simple plot with three lines and legend

The goal of this chapter was to get familiar with some basics of Python such as the importance of learning Python, the data types, conditional statements and functions, reading data, and matplotlib library for visualizing data.

In the next chapter, we begin our journey in times series from stationary.

3. Stationary in Times Series

Making the time series stationary is critical if we want the forecasting model to work well. Why? Because most of the data you collect will have non-stationary trends. And if the spikes are erratic how can you be sure the model will work properly?

3.1 Introduction to Stationary

'Stationarity' is one of the most important concepts you will come across when working with time series data. A stationary series is one in which the properties – mean, variance and covariance, do not vary with time.

Let us understand this using an intuitive example. Consider the three plots shown below:

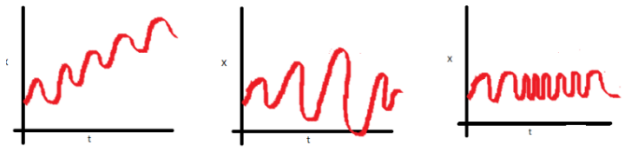


Figure 3.1: The month vs passenger

- In the first plot, we can clearly see that the mean varies (increases) with time which results in an upward trend. Thus, this is a non-stationary series. For a series to be classified as stationary, it should not exhibit a trend.
- Moving on to the second plot, we certainly do not see a trend in the series, but the variance of the series is a function of time. As mentioned previously, a stationary series must have a constant variance.
- If you look at the third plot, the spread becomes closer as the time increases, which implies that the covariance is a function of time.

The three examples shown above represent non-stationary time series. Now look at a fourth plot:

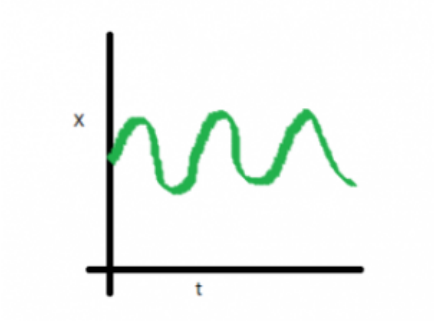


Figure 3.2: The month vs passenger

In this case, the mean, variance and covariance are constant with time. This is what a stationary time series looks like.

Think about this for a second – predicting future values using which of the above plots would be easier? The fourth plot, right? Most statistical models require the series to be stationary to make effective and precise predictions.

So to summarize, a stationary time series is the one for which the properties (namely mean, variance and covariance) do not depend on time. In the next section, we will cover various methods to check if the given series is stationary or not.

3.2 Loading the data

In this and the next few sections, methods will be introducing to check the stationarity of time series data, and the techniques required to deal with any non-stationary series. The dataset we will use: Airline Passangers.

Before we go ahead and analyze our dataset, let's load and preprocess the data first.

```
1
2 #Required Libraries
3 import seaborn as sns
4 import numpy as np
5 import pandas as pd
6 import matplotlib.pyplot as plt
7 %matplotlib inline
```

Python Sample 3.1: Mount Google Drive

```
1 # Get the data set from Seaborn library
2 sns.get_dataset_names()
3
4
5   gh_list = BeautifulSoup(http)
6 ['anscombe',
7  'attention',
8  'brain_networks',
9  'car_crashes',
10 'diamonds',
11 'dots',
12 'exercise',
13 'flights',
```

```
 'fmri',
 'gammas',
 'iris',
 'mpg',
 'planets',
 'tips',
 'titanic']
```

Python Sample 3.2: Loading Data

From above datasets, we pick the **flights**:

```
#load the "flights" dataset
df = sns.load_dataset("flights")
df.head()
```

Python Sample 3.3: Loading flights dataset

Figure 5.44 it summarizes the data.

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Figure 3.3: The head of dataset

The dataset has three columns: year, month, and passengers. The passengers column contains the total number of traveling passengers in a specified month. From above datasets, we pick the **flights**:

```
#shape of data
df.shape
```

(144,33)

The dataset contains 12 year traveling record of the passengers.

```
# Check the type of dataset:
flight_data.columns
```

Index(['year', 'month', 'passengers'], dtype='object')

```
#Convert the type of "passengers" to float:
all_data = flight_data['passengers'].values.astype(float)
all_data:
flight_data.columns
```

3.3 Methods to Check Stationarity

The next step is to determine whether a given series is stationary or not and deal with it accordingly. This section looks at some common methods which we can use to perform this check.

3.3.1 Visual test

We were able to identify the series in which mean and variance were changing with time, simply by looking at each plot. Similarly, we can plot the data and determine if the properties of the series are changing with time or not.

```
#Set the plot size
fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 15
fig_size[1] = 5
plt.rcParams["figure.figsize"] = fig_size
#plot the frequency of the passengers traveling per month
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.xlabel('Months')
plt.grid(True)
plt.autoscale(axis='x',tight=True)
plt.plot(flight_data['passengers'])
```

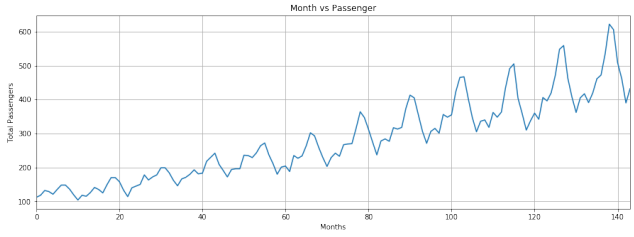


Figure 3.4: The month vs passenger

It is clearly evident that there is an overall increasing trend in the data along with some seasonal variations. However, it might not always be possible to make such visual inferences (we'll see such cases later). So, more formally, we can check stationarity using the following:

- **Plotting Rolling Statistics**: We can plot the moving average or moving variance and see if it varies with time. By moving average/variance I mean that at any instant 't', we'll take the average/variance of the last year, i.e. last 12 months. But again this is more of a visual technique.
- **Statistical Tests** There are various statistical test to check stationary that we will introduce two main of which, that is, Dickey-Fuller as well as Kwiatkowski-Phillips-Schmidt-Shin (KPSS) in the next section:

we'll be using the rolling statistics plots along with statistical tests results a lot, so I have defined a function which takes a TS as input and generated them for us. Please note that standard deviation will be used instead of variance to keep the unit similar to mean.

```
 #Determing rolling statistics
Rolling_mean=flight_data['passengers'].rolling(12).mean()
```

```
Rolling_stdv=flight_data['passengers'].rolling(12).std()

 #Plot rolling statistics:
plt.plot(flight_data['passengers'],color='blue',label='Original')
plt.plot(Rolling_mean, color='red', label='Rolling Mean')
plt.plot(Rolling_stdv,color='black', label = 'Rolling Std')
plt.legend(loc='best')
plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)

```

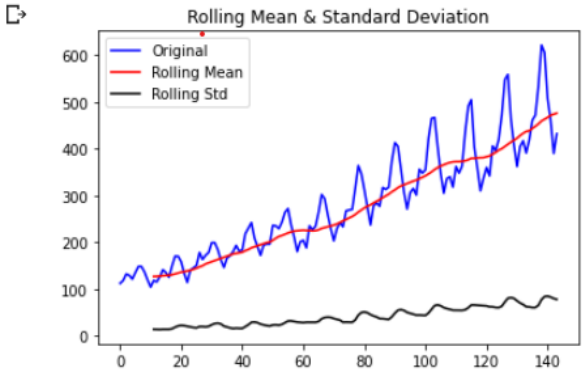


Figure 3.5: Rolling Statistics

Though the variation in standard deviation is small, mean is clearly increasing with time and this is not a stationary series.

3.3.2 Statistical test

Instead of going for the visual test, we can use statistical tests like the unit root stationary tests. Unit root indicates that the statistical properties of a given series are not constant with time, which is the condition for stationary time series. Here is the mathematics explanation of the same.

Suppose we have a time series :

$$y_t = a \times y_{t-1} + \varepsilon_t$$

where y_t is the value at the time instant t and t is the error term. In order to calculate y_t we need the value of y_{t-1}, which is :

$$y_{t-1} = a \times y_{t-2} + \varepsilon_{t-1}$$

If we do that for all observations, the value of y_t will come out to be:

$$y_t = a^n * y_{t-n} + \varepsilon_{t-i} * a^i$$

If the value of a is 1 (unit) in the above equation, then the predictions will be equal to the y_{t-n} and sum of all errors from t-n to t, which means that the variance will increase with time. This is knows as unit root in a time series. We know that for a stationary time series, the variance must not

be a function of time. The unit root tests check the presence of unit root in the series by checking if value of a=1. Below are the two of the most commonly used unit root stationary tests:

3.3.3 ADF (Augmented Dickey Fuller) Test

The Dickey Fuller test is one of the most popular statistical tests. It can be used to determine the presence of unit root in the series, and hence help us understand if the series is stationary or not. The null and alternate hypothesis of this test are:

Null Hypothesis: The series has a unit root (value of a =1)

Alternate Hypothesis: The series has no unit root.

If we fail to reject the null hypothesis, we can say that the series is non-stationary. This means that the series can be linear or difference stationary (we will understand more about difference stationary in the next section).

```
1
2  #define function for ADF test
3  from statsmodels.tsa.stattools import adfuller
4  def adf_test(timeseries):
5      #Perform Dickey-Fuller test:
6      print ('Results of Dickey-Fuller Test:')
7      dftest = adfuller(timeseries, autolag='AIC')
8      dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#
       Lags Used','Number of Observations Used'])
9      for key,value in dftest[4].items():
10        dfoutput['Critical Value (%s)'%key] = value
11     print (dfoutput)
12
13 #apply adf test on the series
14 adf_test(all_data)
```

Python Sample 3.4: Loading flights dataset

Results of ADF test: The ADF tests gives the following results – test statistic, p value and the critical value at 1%, 5% , and 10% confidence intervals. The results of our test for this particular series are:

```
1
2  Results of Dickey-Fuller Test:
3  Test Statistic                  0.815369
4  p-value                         0.991880
5  #Lags Used                     13.000000
6  Number of Observations Used    130.000000
7  Critical Value (1%)             -3.481682
8  Critical Value (5%)             -2.884042
9  Critical Value (10%)            -2.578770
10 dtype: float64
```

Python Sample 3.5: Loading flights dataset

Test for stationarity: If the test statistic is less than the critical value, we can reject the null hypothesis (aka the series is stationary). When the test statistic is greater than the critical value, we fail to reject the null hypothesis (which means the series is not stationary).

In our above example, the test statistic > critical value, which implies that the series is not stationary. This confirms our original observation which we initially saw in the visual test.

3.3.4 KPSS(Kwiatkowski-Phillips-Schmidt-Shin) Test

KPSS is another test for checking the stationarity of a time series (slightly less popular than the Dickey Fuller test). The null and alternate hypothesis for the KPSS test are opposite that of the ADF test, which often creates confusion.

The authors of the KPSS test have defined the null hypothesis as the process is trend stationary, to an alternate hypothesis of a unit root series. We will understand the trend stationarity in detail in the next section. For now, let's focus on the implementation and see the results of the KPSS test.

Null Hypothesis: The process is trend stationary.

Alternate Hypothesis: The series has a unit root (series is not stationary).

```
# KPSS test
from statsmodels.tsa.stattools import kpss
def kpss_test(all_data, **kw):
    statistic, p_value, n_lags, critical_values = kpss(all_data, **kw)
    # Format Output
    print(f'KPSS Statistic: {statistic}')
    print(f'p-value: {p_value}')
    print(f' lags Used: {n_lags}')
    print('Critial Values:')
    for key, value in critical_values.items():
        print(f'   {key} : {value}')
    print(f'Result: The series is {"not " if p_value < 0.05 else ""}
    stationary')

kpss_test(all_data)
```

Python Sample 3.6: Kpss test

Results of KPSS test: Following are the results of the KPSS test – Test statistic, p-value, and the critical value at 1%, 2.5%, 5%, and 10% confidence intervals. For the air passengers dataset, here are the results:

```
KPSS Statistic: 1.0521750110135095
p-value: 0.01
 lags Used: 14
Critial Values:
   10% : 0.347
   5% : 0.463
   2.5% : 0.574
   1% : 0.739
Result: The series is not stationary
```

Python Sample 3.7: Kpss test results

3.3.5 Test for stationarity

If the test statistic is greater than the critical value, we reject the null hypothesis (series is not stationary). If the test statistic is less than the critical value, if fail to reject the null hypothesis (series is stationary). For the air passenger data, the value of the test statistic is greater than the critical value at all confidence intervals, and hence we can say that the series is not stationary.

I usually perform both the statistical tests before I prepare a model for my time series data. It once happened that both the tests showed contradictory results. One of the tests showed that the

series is stationary while the other showed that the series is not! I got stuck at this part for hours, trying to figure out how is this possible. As it turns out, there are more than one type of stationarity.

So in summary, the ADF test has an alternate hypothesis of linear or difference stationary, while the KPSS test identifies trend-stationarity in a series.

3.4 Types of Stationarity

Let us understand the different types of stationarities and how to interpret the results of the above test.

- Strict Stationary: A strict stationary series satisfies the mathematical definition of a stationary process. For a strict stationary series, the mean, variance and covariance are not the function of time. The aim is to convert a non-stationary series into a strict stationary series for making predictions.
- Trend Stationary: A series that has no unit root but exhibits a trend is referred to as a trend stationary series. Once the trend is removed, the resulting series will be strict stationary. The KPSS test classifies a series as stationary on the absence of unit root. This means that the series can be strict stationary or trend stationary.
- Difference Stationary: A time series that can be made strict stationary by differencing falls under difference stationary. ADF test is also known as a difference stationarity test. It's always better to apply both the tests, so that we are sure that the series is truly stationary. Let us look at the possible outcomes of applying these stationary tests.

3.4.1 Making a Time Series Stationary

Now that we are familiar with the concept of stationarity and its different types, we can finally move on to actually making our series stationary. Always keep in mind that in order to use time series forecasting models, it is necessary to convert any non-stationary series to a stationary series first.

- **Differencing**
  In this method, we compute the difference of consecutive terms in the series. Differencing is typically performed to get rid of the varying mean. Mathematically, differencing can be written as:

$$y'_t = y_t - y_{t-1}$$

  where y_t is the value at a time t.
  Applying differencing on our series and plotting the results:

```
1
2 diff= flight_data['passengers']-flight_data['passengers'].shift(1)
3 diff.dropna().plot()
```

Python Sample 3.8: differencing flights dataset

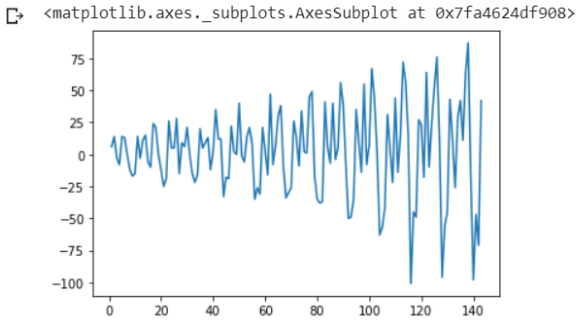


Figure 3.6: differncing flight passengers

- **Seasonal Differencing**
  In seasonal differencing, instead of calculating the difference between consecutive values, we calculate the difference between an observation and a previous observation from the same season. For example, an observation taken on a Monday will be subtracted from an observation taken on the previous Monday. Mathematically it can be written as:

$$y'_t = y_t - y_{t-1}$$

```
division=flight_data['passengers']/(flight_data['passengers'].shift(1))
flights_log.dropna().plot()
```

Python Sample 3.9: Seasonal differencing of flights dataset

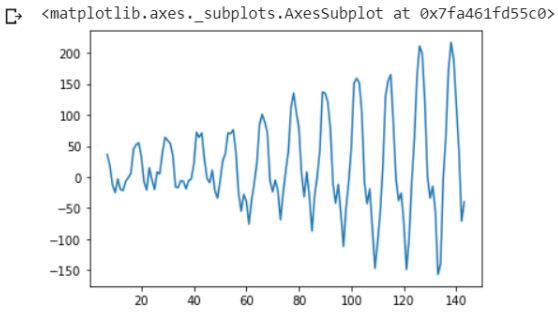


Figure 3.7: Seasonal differncing flight passengers

- **Transformation**
  Transformations are used to stabilize the non-constant variance of a series. Common transformation methods include power transform, square root, and log transform. Let's do a quick log transform and differencing on our air passenger dataset:

```
diff= flight_data['passengers']-flight_data['passengers'].shift(7)
diff.dropna().plot()
```

Python Sample 3.10: Transformation of flights dataset

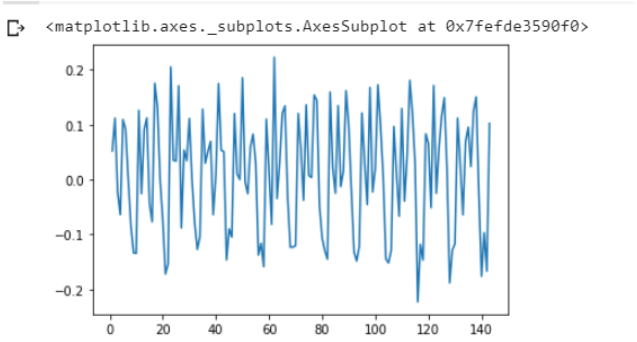


Figure 3.8: Log transformation of flights passengers

As you can see, this plot is a significant improvement over the previous plots. You can use square root or power transformation on the series and see if they come up with better results.

Now, let apply rolling statistics on transformed data. However, notice the first 11 being Nan, since we are taking average of last 12 values, rolling mean is not defined for first 11 values. This can be observed as:

```
1
2 Rolling_mean_log=flights_log.rolling(12).mean()
3 Rolling_stdv_log=flights_log.rolling(12).std()
4 rolling_mean
```

Python Sample 3.11: Rolling statistics on log transformed data

	year	passengers
0	NaN	NaN
1	NaN	NaN
2	NaN	NaN
3	NaN	NaN
4	NaN	NaN
...	...	...
139	1959.666667	463.333333
140	1959.750000	467.083333
141	1959.833333	471.583333
142	1959.916667	473.916667
143	1960.000000	476.166667

144 rows × 2 columns

Figure 3.9: Rolling Statistics log transformed data

Lets drop these NaN values and check the plots to test stationarity.

```
Rolling_mean_log.dropna(inplace=True)
Rolling_mean_log
```

```
12     0.002203
13     0.005466
14     0.005496
15     0.003789
16     0.002710
          ...
139    0.006728
140    0.007730
141    0.010382
142    0.006209
143    0.005378
Name: passengers, Length: 132, dtype: float64
```

Figure 3.10: Table Rolling Statistics on log transformed data

and their plot as:

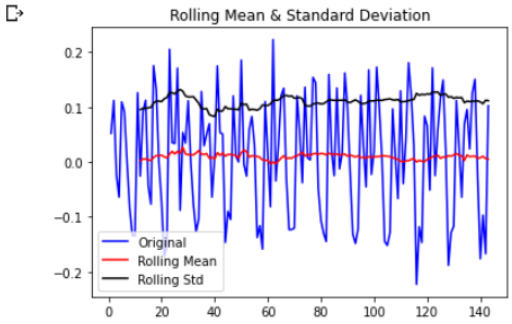


Figure 3.11: Plot Rolling Statistics on log transformed data

We can see that the mean and std variations have small variations with time.

In this chapter, we covered different methods that can be used to check the stationarity of a time series. But the buck doesn't stop here. The next step is to apply a forecasting model on the series we obtained. You can refer to the following chapter to build such a models.

.

4. Forecasting a time series

We saw different techniques and all of them worked reasonably well for making the TS stationary. Lets make model on the TS after differencing as it is a very popular techniques.

4.1 Times Series by ARIMA

It is relatively easier to add noise and seasonality back into predicted residuals in this case. Having performed the trend and seasonality estimation techniques, there can be two situations:

- **A strictly stationary series** with no dependence among the values. This is the easy case wherein we can model the residuals as white noise. But this is very rare.
- A series with significant **dependence among values**. In this case we need to use some statistical models like ARIMA to forecast the data.

Let me give you a brief introduction to ARIMA. I won't go into the technical details but you should understand these concepts in detail if you wish to apply them more effectively. ARIMA stands for Auto-Regressive Integrated Moving Averages. The ARIMA forecasting for a stationary time series is nothing but a linear (like a linear regression) equation. The predictors depend on the parameters (p,d,q) of the ARIMA model:

1.**Number of AR (Auto-Regressive) terms (p)**: AR terms are just lags of dependent variable. For instance if p is 5, the predictors for x(t) will be x(t-1)....x(t-5).

2.**Number of MA (Moving Average) terms (q)**: MA terms are lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)....e(t-5) where e(i) is the difference between the moving average at ith instant and actual value.

3.**Number of Differences (d)**: These are the number of nonseasonal differences, i.e. in this case we took the first order difference. So either we can pass that variable and put d=0 or pass the original variable and put d=1. Both will generate same results. An importance concern here is how to determine the value of 'p' and 'q'. We use two plots to determine these numbers. Lets discuss them first.

1.**Autocorrelation Function (ACF)**: It is a measure of the correlation between the the TS with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant 't1'...'t2'

with series at instant 't1-5'...'t2-5' (t1-5 and t2 being end points).

2.**Partial Autocorrelation Function (PACF)**: This measures the correlation between the TS with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. Eg at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4. The ACF and PACF plots for the TS after differencing can be plotted as:

The ACF and PACF plots for the TS after differencing can be plotted as:

```
#Plot ACF:
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(flights_log_diff)),linestyle='--',color='
    gray')
plt.axhline(y=1.96/np.sqrt(len(flights_log_diff)),linestyle='--',color='gray
    ')
plt.title('Autocorrelation Function')
#Plot PACF:
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(flights_log_diff)),linestyle='--',color='
    gray')
plt.axhline(y=1.96/np.sqrt(len(flights_log_diff)),linestyle='--',color='gray
    ')
plt.title('Partial Autocorrelation Function')
plt.tight_layout()
```

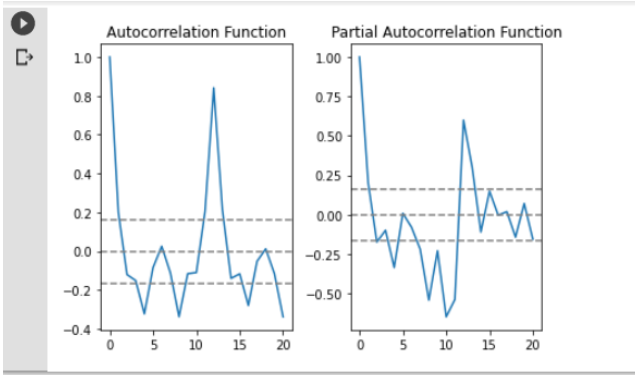


Figure 4.1: ACF and PACF plots for data

In this plot, the two dotted lines on either sides of 0 are the confidence intervals. These can be used to determine the 'p' and 'q' values as:

1.**p** – The lag value where the **PACF**PACF chart crosses the upper confidence interval for the first time. If you notice closely, in this case p=2.

2.**q** – The lag value where the **ACF** chart crosses the upper confidence interval for the first time. If you notice closely, in this case q=2.

Now, lets make 3 different ARIMA models considering individual as well as combined effects. I will also print the RSS for each. Please note that here RSS is for the values of residuals and not actual series.

We need to load the ARIMA model first:

AR Model

```
# fitting AR
from statsmodels.tsa.arima_model import ARIMA
model = ARIMA(flights_log, order=(2, 1, 0))
results_AR = model.fit(disp=-1)
plt.plot(flights_log_diff)
plt.plot(results_AR.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_AR.fittedvalues-flights_log_diff)**2)))
```

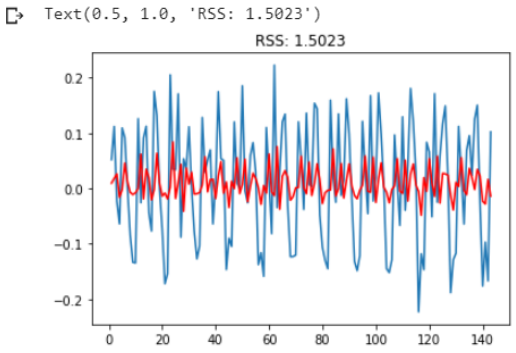


Figure 4.2: Fitting AR on the data

MA model

```
#fitting MA model
model = ARIMA(flights_log, order=(0, 1, 2))
results_MA = model.fit(disp=-1)
plt.plot(flights_log_diff)
plt.plot(results_MA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_MA.fittedvalues-flights_log_diff)**2))
```

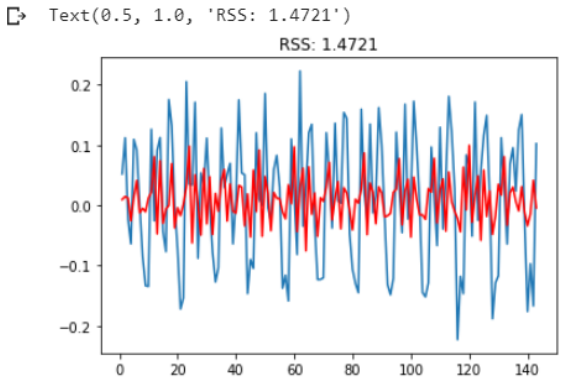


Figure 4.3: Fitting MA on the data

Combind Model

```
#Combind model
model = ARIMA(flights_log, order=(2, 1, 2))
results_ARIMA = model.fit(disp=-1)
plt.plot(flights_log_diff)
plt.plot(results_ARIMA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-flights_log_diff)**2)
    )
```

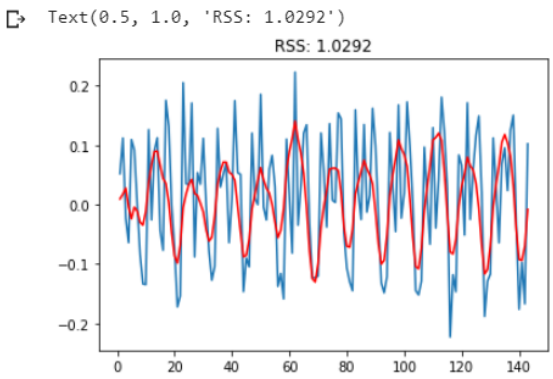


Figure 4.4: Fitting ARMA on the data

Here we can see that the AR and MA models have almost the same RSS, but the combined is significantly better. Now, we are left with 1 last step, i.e. taking these values back to the original scale

4.1.1 Taking it back to original scale

Since the combined model gave best result, lets scale it back to the original values and see how well it performs there. First step would be to store the predicted results as a separate series and observe it.

```
# Prediction by ARIMA
predictions_ARIMA_diff = pd.Series(results_ARIMA.fittedvalues, copy=True)
predictions_ARIMA_diff
```

```
1      0.009580
2      0.017491
3      0.027670
4     -0.004521
5     -0.023890
         ...
139   -0.041176
140   -0.092350
141   -0.094013
142   -0.069924
143   -0.008127
Length: 143, dtype: float64
```

Figure 4.5: The predicted data by ARIMA

Notice that these start from '1949-02-01' and not the first month. Why? This is because we took a lag by 1 and first element doesn't have anything before it to subtract from. The way to convert the differencing to log scale is to add these differences consecutively to the base number. An easy way to do it is to first determine the cumulative sum at index and then add it to the base number. The cumulative sum can be found as:

```
# Prediction by ARIMA
predictions_ARIMA_diff_cumsum = predictions_ARIMA_diff.cumsum()
predictions_ARIMA_diff_cumsum
```

```
1      0.009580
2      0.027071
3      0.054742
4      0.050221
5      0.026331
         ...
139    1.372553
140    1.280203
141    1.186190
142    1.116266
143    1.108139
Length: 143, dtype: float64
```

Figure 4.6: The predicted data by ARIMA by using cumsum

You can quickly do some back of mind calculations using previous output to check if these are correct. Next we've to add them to base number. For this lets create a series with all values as base

number and add the differences to it. This can be done as:

```
predictions_ARIMA_log = pd.Series(flights_log)
predictions_ARIMA_log = predictions_ARIMA_log.add(
    predictions_ARIMA_diff_cumsum,fill_value=0)
predictions_ARIMA_log.head()
```

```
0    4.718499
1    4.780265
2    4.909873
3    4.914554
4    4.846011
dtype: float64
```

Figure 4.7: ACF and PACF plots for data

Here the first element is base number itself and from thereon the values cumulatively added. Last step is to take the exponent and compare with the original series.

```
predictions_ARIMA = np.exp(predictions_ARIMA_log)
plt.plot(ts)
plt.plot(predictions_ARIMA)
plt.title('RMSE: %.4f'% np.sqrt(sum((predictions_ARIMA-ts)**2)/len(ts)))
```

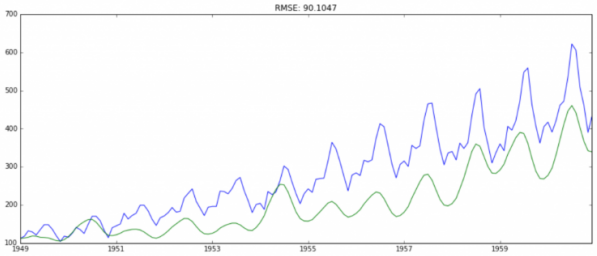


Figure 4.8: Compare the original data and re-back predicted data

4.2 Time Series by LSTM

In this section, we will learn how to use advanced deep learning models such as Long Short Term Memory Networks (LSTM), are capable of capturing patterns in the time series data, and therefore can be used to make predictions regarding the future trend of the data.

It is worthwhile to mention that LSTM is one of the most widely used algorithm to solve sequence problems.

Also you will see how to use LSTM algorithm to make future predictions using time series data. In the first section, we will be using the PyTorch library, which is one of the most commonly used Python libraries for deep learning. In the second section,however, , we will learn how to apply another library, Keras.

4.2.1 LSTM by Pytorch

The first step is how to call the required libraries and load **data**. let begin calling the libraries:

```
#Required Libraries
import torch
import torch.nn as nn

import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we'll need to load the data file:

```
# Get the data set from Seaborn library
sns.get_dataset_names()


  gh_list = BeautifulSoup(http)
['anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'exercise',
 'flights',
 'fmri',
 'gammas',
 'iris',
 'mpg',
 'planets',
 'tips',
 'titanic']
```

From above datasets, we pick the **flights**:

```
#load the "flights" dataset
df = sns.load_dataset("flights")
df.head()
```

Figure 5.44 it summarizes the data.

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Figure 4.9: The head of dataset

The dataset has three columns: year, month, and passengers. The passengers column contains the total number of traveling passengers in a specified month. From above datasets, we pick the **flights**:

```
#shape of data
df.shape
```

(144,33)

The dataset contains 12 year traveling record of the passengers.

Here, the our plan is to predict the number of passengers who traveled in the last 12 months based on first 132 months.

In other words, the data set divided to parts, trian and test.

Train: The first 132 months

Test: The last 12 months

We will use the LSTM model to evalaute the last 12 months.

```

#Set the plot size
fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 15
fig_size[1] = 5
plt.rcParams["figure.figsize"] = fig_size
#plot the frequency of the passengers traveling per month
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.xlabel('Months')
plt.grid(True)
plt.autoscale(axis='x',tight=True)
plt.plot(flight_data['passengers'])
```

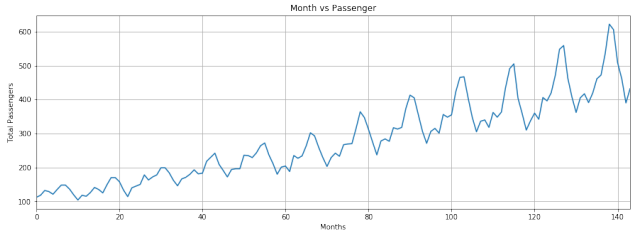


Figure 4.10: The month vs passenger

The output shows that over the years the average number of passengers traveling by air increased. The number of passengers traveling within a year fluctuates, which makes sense because during summer or winter vacations, the number of traveling passengers increases compared to the other parts of the year.

4.2.2 Data Processing

```
# Check the type of dataset:
flight_data.columns
```

Index([’year’, ’month’, ’passengers’], dtype=’object’)

```
#Convert the type of "passengers" to float:
all_data = flight_data['passengers'].values.astype(float)
all_data:
flight_data.columns
```

```
array([112., 118., 132., 129., 121., 135., 148., 148., 136., 119., 104.,
       118., 115., 126., 141., 135., 125., 149., 170., 170., 158., 133.,
       114., 140., 145., 150., 178., 163., 172., 178., 199., 199., 184.,
       162., 146., 166., 171., 180., 193., 181., 183., 218., 230., 242.,
       209., 191., 172., 194., 196., 196., 236., 235., 229., 243., 264.,
       272., 237., 211., 180., 201., 204., 188., 235., 227., 234., 264.,
       302., 293., 259., 229., 203., 229., 242., 233., 267., 269., 270.,
       315., 364., 347., 312., 274., 237., 278., 284., 277., 317., 313.,
       318., 374., 413., 405., 355., 306., 271., 306., 315., 301., 356.,
       348., 355., 422., 465., 467., 404., 347., 305., 336., 340., 318.,
       362., 348., 363., 435., 491., 505., 404., 359., 310., 337., 360.,
       342., 406., 396., 420., 472., 548., 559., 463., 407., 362., 405.,
       417., 391., 419., 461., 472., 535., 622., 606., 508., 461., 390.,
       432.])
```

Figure 4.11: The array of passengers

```
# Split our data set into training and test sets:
test_data_size = 12

train_data = all_data[:-12]# ignore the last 12 months
test_data = all_data[-12:]

((132,), (12,))
```

```
# Normalize the data:
from sklearn.preprocessing import MinMaxScaler # using MinMaxScaler class
    from the sklearn.preprocessing module to scale our data.

scaler = MinMaxScaler(feature_range=(-1, 1)) #set the bounds of scaler
```

```
train_data_normalized = scaler.fit_transform(train_data .reshape(-1, 1)) #
    apply transform

# print the first normalized 5 records of train
train_data_normalized[:5]
```

```
array([[1.        ],
       [0.57802198],
       [0.33186813],
       [0.13406593],
       [0.32307692]])
```

Figure 4.12: The normalized data

It is important to mention here that data normalization is only applied on the training data and not on the test data. If normalization is applied on the test data, there is a chance that some information will be leaked from training set into the test set.

```
#convert our dataset into tensors since PyTorch models
train_data_normalized = torch.FloatTensor(train_data_normalized).view(-1)
```

```
train_data_normalized[:13]
```

```
tensor([-0.9648, -0.9385, -0.8769, -0.8901, -0.9253, -0.8637, -0.8066, -0.8066,
        -0.8593, -0.9341, -1.0000, -0.9385, -0.9516])
```

Figure 4.13: Convert data to tensor

```
#set the input sequence length
train_window = 12
```

```
#The function will accept the raw input data and will return a list of
    tuples
def create_inout_sequences(input_data, tw):
    inout_seq = []
    L = len(input_data)
    for i in range(L-tw):
        train_seq = input_data[i:i+tw]
        train_label = input_data[i+tw:i+tw+1]
        inout_seq.append((train_seq ,train_label))
    return inout_seq
```

```
#create sequences and corresponding labels for training:
train_inout_seq = create_inout_sequences(train_data_normalized, train_window
    )
```

```
train_inout_seq[:5]
```

```
[(tensor([-0.9648, -0.9385, -0.8769, -0.8901, -0.9253, -0.8637, -0.8066, -0.8066,
          -0.8593, -0.9341, -1.0000, -0.9385]), tensor([-0.9516])),
 (tensor([-0.9385, -0.8769, -0.8901, -0.9253, -0.8637, -0.8066, -0.8066, -0.8593,
          -0.9341, -1.0000, -0.9385, -0.9516]),
  tensor([-0.9033])),
 (tensor([-0.8769, -0.8901, -0.9253, -0.8637, -0.8066, -0.8066, -0.8593, -0.9341,
          -1.0000, -0.9385, -0.9516, -0.9033]), tensor([-0.8374])),
 (tensor([-0.8901, -0.9253, -0.8637, -0.8066, -0.8066, -0.8593, -0.9341, -1.0000,
          -0.9385, -0.9516, -0.9033, -0.8374]), tensor([-0.8637])),
 (tensor([-0.9253, -0.8637, -0.8066, -0.8066, -0.8593, -0.9341, -1.0000, -0.9385,
          -0.9516, -0.9033, -0.8374, -0.8637]), tensor([-0.9077]))]
```

Figure 4.14: The sequence of tensor training data

As you can see the first sequence consists of the first 12 items and the 13th item is the label for the first sequence which means each item is a tuple where the first element consists of the 12 items of a sequence, and the second tuple element contains the corresponding label.

```
# Creat the LSTM
class LSTM(nn.Module):
    def __init__(self, input_size=1, hidden_layer_size=100, output_size=1):
        super().__init__()
        self.hidden_layer_size = hidden_layer_size

        self.lstm = nn.LSTM(input_size, hidden_layer_size)

        self.linear = nn.Linear(hidden_layer_size, output_size)

        self.hidden_cell = (torch.zeros(1,1,self.hidden_layer_size),
                            torch.zeros(1,1,self.hidden_layer_size))

    def forward(self, input_seq):
        lstm_out, self.hidden_cell = self.lstm(input_seq.view(len(input_seq)
    ,1, -1), self.hidden_cell)
        predictions = self.linear(lstm_out.view(len(input_seq), -1))
        return predictions[-1]
```

```
# create an object of the LSTM() class, define a loss function and the
    optimizer:
model = LSTM()
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

model
```

4.2.3 Training Model

```
# Train the model for 25 epochs
epochs = 150

for i in range(epochs):
    for seq, labels in train_inout_seq:
        optimizer.zero_grad()
        model.hidden_cell = (torch.zeros(1, 1, model.hidden_layer_size),
                             torch.zeros(1, 1, model.hidden_layer_size))

        y_pred = model(seq)

        single_loss = loss_function(y_pred, labels)
        single_loss.backward()
```

```
        optimizer.step()

    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.01804501
epoch:  26 loss: 0.01084352
epoch:  51 loss: 0.01349457
epoch:  76 loss: 0.00052031
epoch: 101 loss: 0.00021809
epoch: 126 loss: 0.00062108
epoch: 149 loss: 0.0000571791
```

Figure 4.15: The last result of apply model after 150 epochs

You may get different values since by default weights are initialized randomly in a PyTorch neural network.

Let try making model prediction:

```

# Make prediction:
model.eval()

for i in range(fut_pred):
    seq = torch.FloatTensor(test_inputs[-train_window:])
    with torch.no_grad():
        model.hidden = (torch.zeros(1, 1, model.hidden_layer_size),
                        torch.zeros(1, 1, model.hidden_layer_size))
        test_inputs.append(model(seq).item())
```

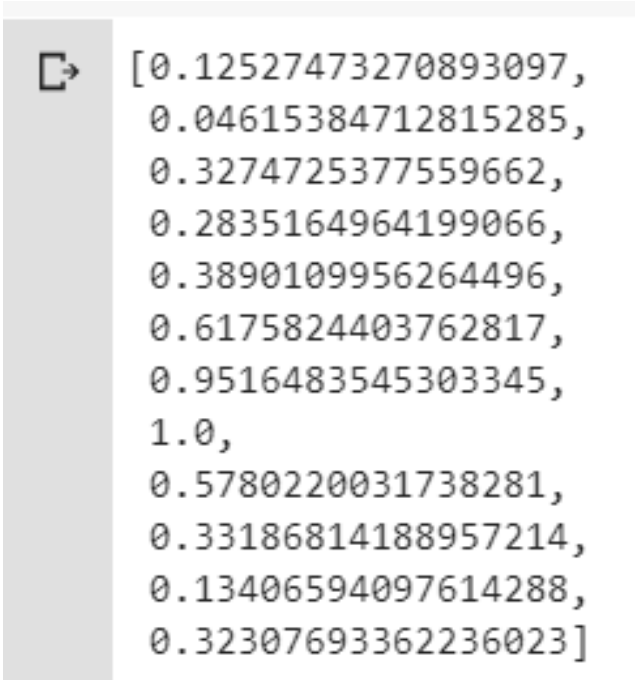


Figure 4.16: The normalized predicted data

```
len(test_inputs)
```

12

```
# Trnsform to the actual values:
actual_predictions = scaler.inverse_transform(np.array(test_inputs ).reshape
    (-1, 1))
actual_predictions
```

```
array([[360.00000169],
       [342.00000022],
       [406.00000234],
       [396.00000294],
       [420.00000151],
       [472.00000519],
       [548.00000066],
       [559.        ],
       [463.00000572],
       [407.00000228],
       [362.00000157],
       [405.0000024 ]])
```

Figure 4.17: The inverse-transformed data

Let's now plot the predicted values against the actual values. Look at the following code:

```
# create a list that contains numeric values for the last 12 months
x = np.arange(132, 144, 1)
x
```

array([132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143])

In the script above we create a list that contains numeric values for the last 12 months. The first month has an index value of 0, therefore the last month will be at index 143.

In the following script, we will plot the total number of passengers for 144 months, along with the predicted number of passengers for the last 12 months.

```
#plot the predicted values against the actual values
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.grid(True)
plt.autoscale(axis='x', tight=True)
plt.plot(flight_data['passengers'])
plt.plot(x,actual_predictions)
plt.show()
```

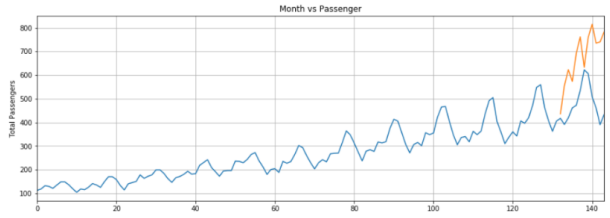


Figure 4.18: The month vs passenger

The predictions made by our LSTM are depicted by the orange line. You can see that our algorithm is not too accurate but still it has been able to capture upward trend for total number of passengers traveling in the last 12 months along with occasional fluctuations. You can try with a greater number of epochs and with a higher number of neurons in the LSTM layer to see if you can get better performance.

To have a better view of the output, we can plot the actual and predicted number of passengers for the last 12 months as follows:

```
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.grid(True)
plt.autoscale(axis='x', tight=True)

plt.plot(flight_data['passengers'][-train_window:])
plt.plot(x,actual_predictions)
plt.show()
```

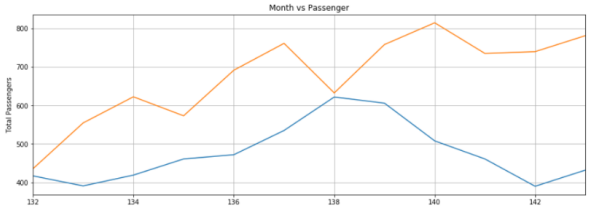


Figure 4.19: The month vs passenger

4.3 LSTM by Keras

In this section, we will discover how to develop LSTM networks in Python using the Keras deep learning library to address a demonstration time-series prediction problem.

So far, we could make forecasts on a time series of international airline passengers by various methods such as AR, MA, ARIMA, LSTM by Pytorch. However, we wish to do repeat LSTM but through Keras.

Let load and plot the flights data again, here:

```
#load the "Airline Passengers" dataset
df=pd.read_csv("/airline-passengers.csv")
df
    Month  Passengers
```

```
0 1949-01 112
1 1949-02 118
2 1949-03 132
3 1949-04 129
4 1949-05 121
... ... ...
139 1960-08 606
140 1960-09 508
141 1960-10 461
142 1960-11 390
143 1960-12 432

# Create a time series plot.
plt.figure(figsize = (15, 5))
plt.plot(data, label = "Airline Passengers")
plt.xlabel("Months")
plt.ylabel("1000 International Airline Passengers")
plt.title("Monthly Total Airline Passengers 1949 - 1960")
plt.legend()
plt.show()
```

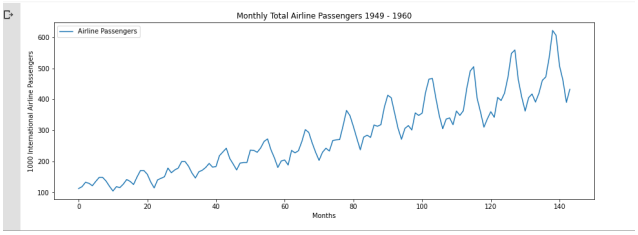


Figure 4.20: The month vs passenger

As already mentioned above, we can observe a strong upwards trend in terms of numbers of passengers with some seasonality component. The seasonality may be understood to coincide with holiday periods, but we'd need to have a closer look at the actual time periods to confirm this.

We did de-trending the time series and applying further "cleaning" techniques, which would be a prerequisite e.g. in an ARIMA setting.

However, for simplicity reasons we will just proceed with transformed data as the previous section, LSTM by Pytorch.

The only transformations we'll be doing are:

Scale data to the (0,1)(0,1) interval for increased numerical stability. Re-shape the data so we have one column as response (called YY in the code) and another one as predictor variable (called X in the code).

Now we are ready to build the LSTM model

```

# Let's load the required libs.
# We'll be using the Tensorflow backend (default).
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
```

```
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

Before we do anything, it is a good idea to fix the random number seed to ensure our results are reproducible.

```
# fix random seed for reproducibility
numpy.random.seed(7)
```

We can also use the following code to load the data and convert the integer values to floating point values, which are more suitable for modeling with a neural network.

```
# load the dataset
dataframe = pandas.read_csv('/airline-passengers.csv', usecols=[1], engine='
    python')
dataset = dataframe.values
dataset = dataset.astype('float32')
```

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. We can easily normalize the dataset using the MinMaxScaler preprocessing class from the scikit-learn library.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

Let split dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations that we can use to train our model, leaving the remaining 33% for testing the model.

```
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))

#output: 96 48
```

Now we can define a function to create a new dataset, as described above. The function takes two arguments: the dataset, which is a NumPy array that we want to convert into a dataset, and the look_back, which is the number of previous time steps to use as input variables to predict the next time period — in this case defaulted to 1.

This default will create a dataset where X is the number of passengers at a given time (t) and Y is the number of passengers at the next time (t + 1).

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
```

Let's use this function to prepare the train and test datasets for modeling.

```
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

The LSTM network expects the input data (X) to be provided with a specific array structure in the form of: [samples, time steps, features].

Currently, our data is in the form: [samples, features] and we are framing the problem as one time step for each sample. We can transform the prepared train and test input data into the expected structure using numpy.reshape() as follows:

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

We are now ready to design and fit our LSTM network for this problem.

The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons, and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM blocks. The network is trained for 100 epochs and a batch size of 1 is used.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

Once the model is fit, we can estimate the performance of the model on the train and test datasets. This will give us a point of comparison for new models.

Note that we invert the predictions before calculating error scores to ensure that performance is reported in the same units as the original data (thousands of passengers per month).

```
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))

# output
Train Score: 22.93 RMSE
Test Score: 47.60 RMSE
```

Finally, we can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model.

Because of how the dataset was prepared, we must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the training dataset in green, and the predictions on the unseen test dataset in red.

```
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] =
    testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset),label = "True value")
plt.plot(trainPredictPlot,label = "Training set prediction")
plt.plot(testPredictPlot,label = "Test set prediction")
plt.legend()
plt.show()
```

We can see that the model did an excellent job of fitting both the training and the test datasets.

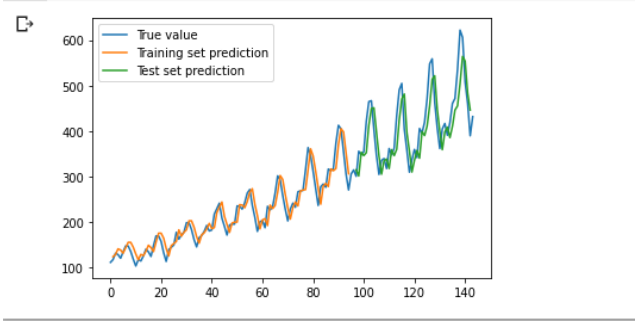


Figure 4.21: The real vs predicted passengers

4.4 Conclusion

In this , we saw how to make future predictions using time series data with LSTM. You also saw how to implement LSTM with PyTorch library and then how to plot predicted results against actual values to see how well the trained algorithm is performing.

5. Deep learning for times series

5.1 Times Series Audio

While much of the literature and buzz on deep learning concerns computer vision and natural language processing(NLP), audio analysis — a field that includes automatic speech recognition(ASR), digital signal processing, and music classification, tagging, and generation — is a growing subdomain of deep learning applications. Some of the most popular and widespread machine learning systems, virtual assistants Alexa, Siri, and Google Home, are largely products built atop models that can extract information from audio signals.

Audio data analysis is about analyzing and understanding audio signals captured by digital devices, with numerous applications in the enterprise, healthcare, productivity, and smart cities. Applications include customer satisfaction analysis from customer support calls, media content analysis and retrieval, medical diagnostic aids and patient monitoring, assisting technologies for people with hearing impairments, and audio analysis for public safety.

In the first part of this article series, we will talk about all you need to know before getting started with the audio data analysis and extract necessary features from a sound/audio file. We will also build an Artificial Neural Network(ANN) for the music genre classification. In the second part, we will accomplish the same by creating the Convolutional Neural Network and will compare their accuracy.

5.1.1 Audio Data Analysis

The sound excerpts are digital audio files in .wav format. Sound waves are digitized by sampling them at discrete intervals known as the sampling rate (typically 44.1kHz for CD-quality audio meaning samples are taken 44,100 times per second).

Each sample is the amplitude of the wave at a particular time interval, where the bit depth determines how detailed the sample will be also known as the dynamic range of the signal (typically 16bit which means a sample can range from 65,536 amplitude values).

What is Sampling and Sampling frequency?

In signal processing, sampling is the reduction of a continuous signal into a series of discrete

values. The sampling frequency or rate is the number of samples taken over some fixed amount of time. A high sampling frequency results in less information loss but higher computational expense, and low sampling frequencies have higher information loss but are fast and cheap to compute.

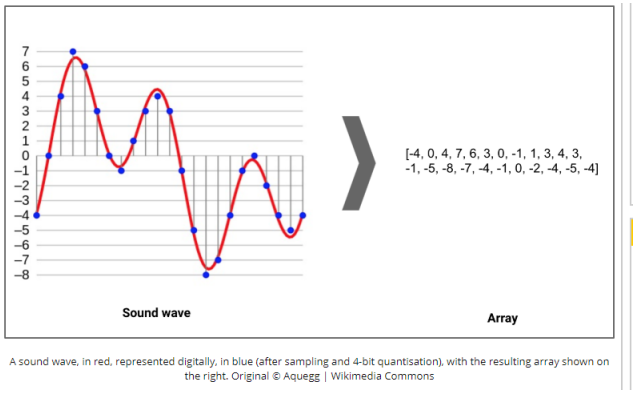


Figure 5.1: The sound wave vs array

Applications of Audio Processing:

What are the potential applications of audio processing? A list a few of them will be listed here:

- Indexing music collections according to their audio features.
- Recommending music for radio channels
- Similarity search for audio files (aka Shazam)
- Speech processing and synthesis — generating artificial voice for conversational agents

Audio Data Handling using Python:

Sound is represented in the form of an audio signal having parameters such as frequency, bandwidth, decibel, etc. A typical audio signal can be expressed as a function of Amplitude and Time.

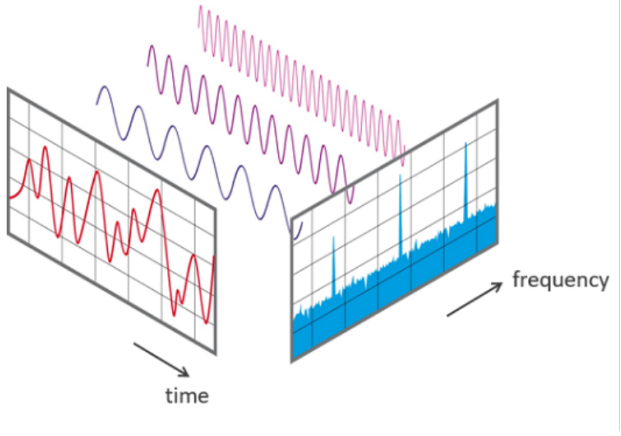


Figure 5.2: The audio forms

There are devices built that help you catch these sounds and represent it in a computer-readable format. Examples of these formats are

- wav (Waveform Audio File) format

- mp3 (MPEG-1 Audio Layer 3) format
- WMA (Windows Media Audio) format
  A typical audio processing process involves the extraction of acoustics features relevant to the task at hand, followed by decision-making schemes that involve detection, classification, and knowledge fusion. Thankfully we have some useful python libraries which make this task easier.

Python Audio Libraries:

Python has some great libraries for audio processing like Librosa and PyAudio.There are also built-in modules for some basic audio functionalities.

1. Librosa

It is a Python module to analyze audio signals in general but geared more towards music. It includes the nuts and bolts to build a MIR(Music information retrieval) system. It has been very well documented along with a lot of examples and tutorials.

Installation:

We will mainly use two libraries for audio acquisition and playback:

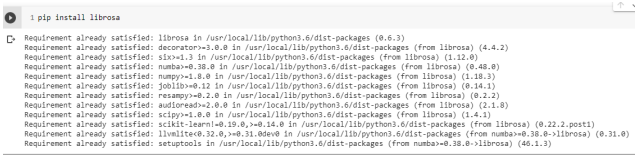


Figure 5.3: Installing *librosa* library

and now we can recall the required libraries:

```
import librosa
import IPython.display as ipd
%matplotlib inline
import matplotlib.pyplot as plt
import librosa.display
import numpy as np
```

To fuel more audio-decoding power, you can install *ffmpeg* which ships with many audio decoders.

2. IPython.display.Audio

IPython.display.Audio lets you play audio directly in your notebook.

Here a random file has been uploaded:

```
from google.colab import drive
drive.mount('/content/drive')

#Loading an audio file:
audio_data = '/content/drive/My Drive/Data_storage/Vocaroo iMPozIbzB8T.mp3'
x , sr = librosa.load(audio_data)
print(type(x), type(sr))

# Dimension of file:
#<class 'numpy.ndarray'> <class 'int'>print(x.shape, sr)#(94316,) 22050
```

This returns an audio time series as a numpy array with a default sampling rate(sr) of 22KHZ mono. We can change this behavior by resampling at 44.1KHz.

```
#Resampling the data:
librosa.load(audio_data, sr=44100)
```

Playing Audio:

We should know the sample rate is the number of samples of audio carried per second, measured in Hz or kHz.

Using,IPython.display.Audio, we can play the audio in our notebook.

```
ipd.Audio(audio_data)
```

This returns an audio widget:

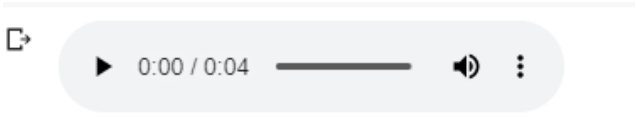


Figure 5.4: Playing audio file

Visualizing Audio:

We can plot the audio array using librosa.display.waveplot:

```

#Visualizing Audio:
plt.figure(figsize=(14, 5))
librosa.display.waveplot(x, sr=sr)
```

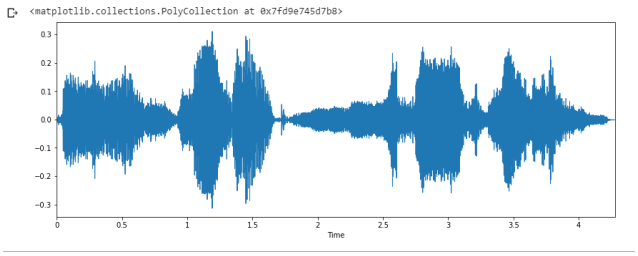


Figure 5.5: The plot of audio data

Spectrogram

A spectrogram is a visual way of representing the signal strength, or “loudness”, of a signal over time at various frequencies present in a particular waveform. Not only can one see whether there is more or less energy at, for example, 2 Hz vs 10 Hz, but one can also see how energy levels vary over time.

A spectrogram is usually depicted as a heat map, i.e., as an image with the intensity shown by varying the color or brightness.

We can display a spectrogram using librosa.display.specshow.

```

X = librosa.stft(x)
Xdb = librosa.amplitude_to_db(abs(X))
plt.figure(figsize=(14, 5))
librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='hz')
plt.colorbar()
```

.stft() converts data into short term Fourier transform. **STFT** converts signals such that we can know the amplitude of the given frequency at a given time. Using **STFT**, we can determine the amplitude of various frequencies playing at a given time of an audio signal. **.specshow** is used to display a spectrogram.

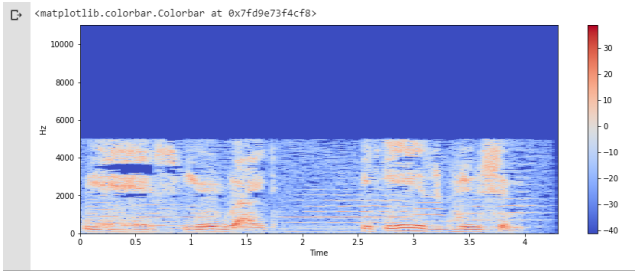


Figure 5.6: The plot of spectrogram

The vertical axis shows frequencies (from 0 to 10kHz), and the horizontal axis shows the time of the clip. Since we see that all action is taking place at the bottom of the spectrogram, we can convert the frequency axis to a logarithmic one.

```
1
2
3 librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='log')
4 plt.colorbar()
```

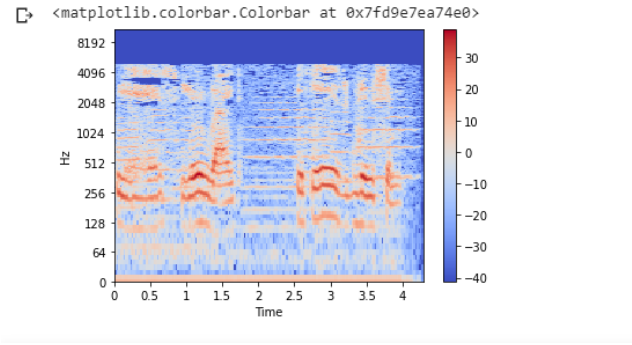


Figure 5.7: The plot of transformed spectrogram

```
1
2 sr2 = 22050 # sample rate
3 T = 5.0    # seconds
4 t2 = np.linspace(0, T, int(T*sr2), endpoint=False) # time variable
5 x2 = 0.5*np.sin(2*np.pi*220*t2)# pure sine wave at 220 Hz
6 #Playing the audio
7 ipd.Audio(x2, rate=sr2) # load a NumPy array
8 #Saving the audio
9 librosa.output.write_wav('tone_220.wav', x2, sr2)
```

Feature extraction from Audio signal

Every audio signal consists of many features. However, we must extract the characteristics that are relevant to the problem we are trying to solve. The process of extracting features to use them for analysis is called feature extraction. Let us study a few of the features in detail.

The spectral features (frequency-based features), which are obtained by converting the time-based signal into the frequency domain using the Fourier Transform, like fundamental frequency, frequency components, spectral centroid, spectral flux, spectral density, spectral roll-off, etc.

1. Spectral Centroid

The spectral centroid indicates at which frequency the energy of a spectrogram is centered upon. in other words, it indicates where the " center of mass" for a sound is located. This is like a weighted mean:

$$f_c = \frac{\sum_k S(k) f(k)}{\sum_k S(k)}$$

Figure 5.8: The spectral centroid mean formula

where S(k) is the spectral magnitude at frequency bin k, f(k) is the frequency at bin k.

librosa.feature.spectral centroid computes the spectral centroid for each frame in a signal:

```
1
2  import sklearn
3  spectral_centroids = librosa.feature.spectral_centroid(x, sr=sr)[0]
4  spectral_centroids.shape
5  # Computing the time variable for visualization
6  plt.figure(figsize=(12, 4))
7  frames = range(len(spectral_centroids))
8  t= librosa.frames_to_time(frames)
9  # Normalising the spectral centroid for visualisation
10 def normalize(x, axis=0):
11     return sklearn.preprocessing.minmax_scale(x, axis=axis)
12 #Plotting the Spectral Centroid along the waveform
13 librosa.display.waveplot(x, sr=sr, alpha=0.4)
14 plt.plot(t, normalize(spectral_centroids), color='r')
```

.spectral$_c$entroid will return an array with columns equal to a number of frames present in your sample.

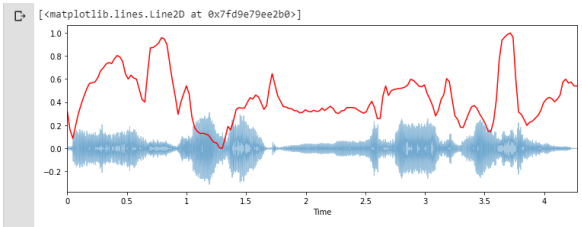


Figure 5.9: The plot of spectral centroid

There is a rise in the spectral centroid in the beginning.

2. Spectral Rolloff

It is a measure of the shape of the signal. It represents the frequency at which high frequencies decline to 0. To obtain it, we have to calculate the fraction of bins in the power spectrogram where 85% of its power is at lower frequencies. *librosa.feature.spectral,olloff* computes the rolloff frequency for each frame in a signal:

```
# Create a rollof frequency
spectral_rolloff = librosa.feature.spectral_rolloff(x+0.01, sr=sr)[0]
plt.figure(figsize=(12, 4))
librosa.display.waveplot(x, sr=sr, alpha=0.4)
plt.plot(t, normalize(spectral_rolloff), color='r')
```

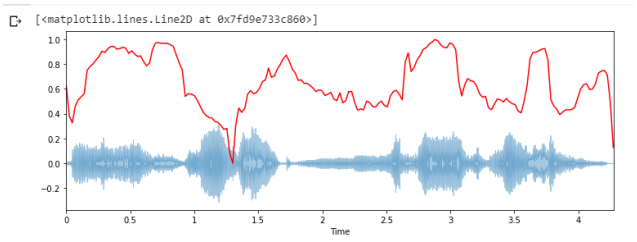


Figure 5.10: The plot of spectral rolloff

3. Spectral Bandwidth

The spectral bandwidth is defined as the width of the band of light at one-half the peak maximum (or full width at half maximum [FWHM]) and is represented by the two vertical red lines and SB on the wavelength axis.

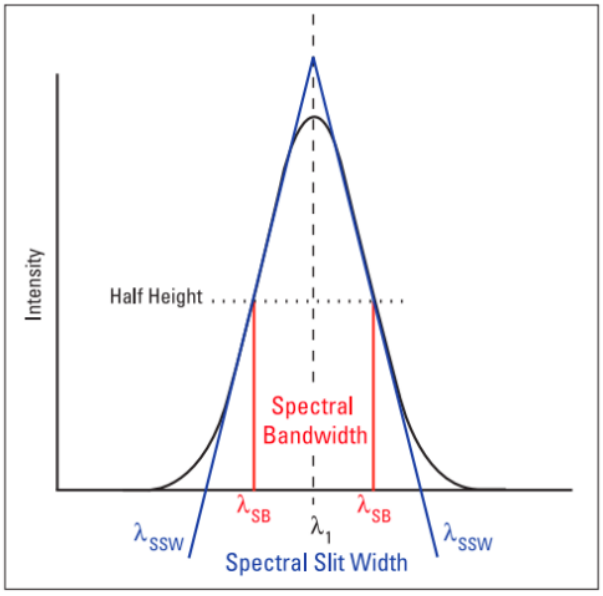


Figure 5.11: The plot of Gaussian intensity distribution of wavelength emerging from monochromator

```
spectral_bandwidth_2 = librosa.feature.spectral_bandwidth(x+0.01, sr=sr)[0]
spectral_bandwidth_3 = librosa.feature.spectral_bandwidth(x+0.01, sr=sr, p
    =3)[0]
spectral_bandwidth_4 = librosa.feature.spectral_bandwidth(x+0.01, sr=sr, p
    =4)[0]
plt.figure(figsize=(15, 9))
librosa.display.waveplot(x, sr=sr, alpha=0.4)
plt.plot(t, normalize(spectral_bandwidth_2), color='r')
plt.plot(t, normalize(spectral_bandwidth_3), color='g')
plt.plot(t, normalize(spectral_bandwidth_4), color='y')
plt.legend(('p = 2', 'p = 3', 'p = 4'))
```

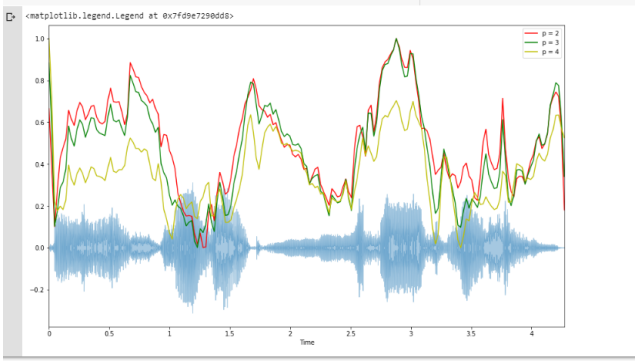


Figure 5.12: The plot of spectral rolloff

4. Zero-Crossing Rate

A very simple way for measuring the smoothness of a signal is to calculate the number of zero-crossing within a segment of that signal. A voice signal oscillates slowly — for example, a 100 Hz signal will cross zero 100 per second — whereas an unvoiced fricative can have 3000 zero crossings per second.

$$zcr = \frac{1}{T-1}\sum_{t=1}^{T-1}\mathbb{I}\{s_t s_{t-1} < 0\}$$

Figure 5.13: The formula to calculate the zero crossing rate

s_t is the signal length t

IIX is the indicator function(=1 if X is true, else=0)

It usually has higher values for highly percussive sounds like those in metal and rock. Now let us visualize it and see how we calculate zero crossing rate.

```
x3, sr3 = librosa.load('/content/drive/My Drive/Data_storage/wolf8.wav')
#Plot the signal:
plt.figure(figsize=(14, 5))
librosa.display.waveplot(x3, sr=sr3)
# Zooming in
n0 = 9000
```

```
n1 = 9100
plt.figure(figsize=(14, 5))
plt.plot(x3[n0:n1])
plt.grid()
```

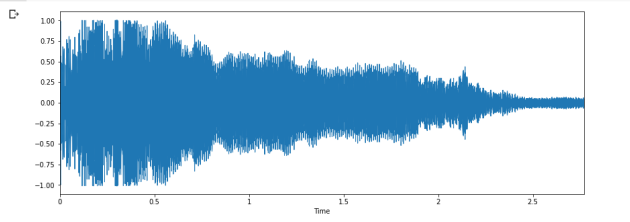


Figure 5.14: The plot of the above audio rate

Let zooming in:

```
n0 = 9000
n1 = 9100
plt.figure(figsize=(14, 5))
plt.plot(x3[n0:n1])
plt.grid()
```

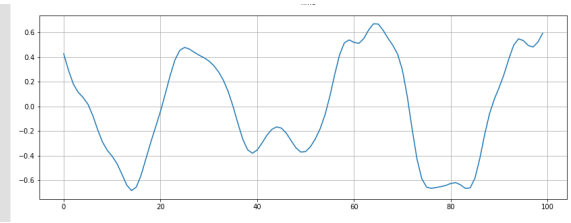


Figure 5.15: The plot of curve for the signal rate

```
zero_crossings = librosa.zero_crossings(x3[n0:n1], pad=False)
print(sum(zero_crossings))#6
```

5. Mel-Frequency Cepstral Coefficients(MFCCs)

The Mel frequency cepstral coefficients (MFCCs) of a signal are a small set of features (usually about 10–20) which concisely describe the overall shape of a spectral envelope. It models the characteristics of the human voice

```
mfccs = librosa.feature.mfcc(x, sr=sr)
mfccs.shape# (20, 120)
#Displaying  the MFCCs:
plt.figure(figsize=(15, 7))
librosa.display.specshow(mfccs, sr=sr, x_axis='time')
```

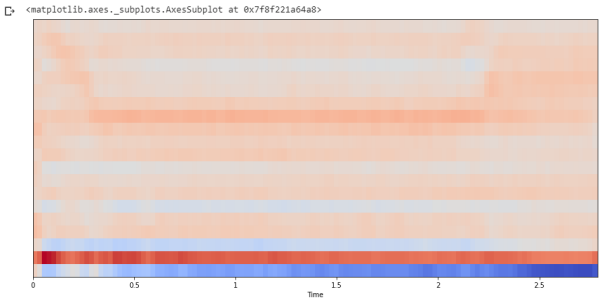


Figure 5.16: The spectrogram plot of audio rate2 by mfcc

6. Chroma feature

A chroma feature or vector is typically a 12-element feature vector indicating how much energy of each pitch class, $\{C, C, D, D, E, \ldots, B\}$, is present in the signal. In short, It provides a robust way to describe a similarity measure between music pieces.

librosa.feature.chroma_stft is used for the computation of Chroma features.

```
chromagram = librosa.feature.chroma_stft(x, sr=sr, hop_length=512)
plt.figure(figsize=(15, 5))
librosa.display.specshow(chromagram, x_axis='time', y_axis='chroma',
    hop_length=512, cmap='coolwarm')
```

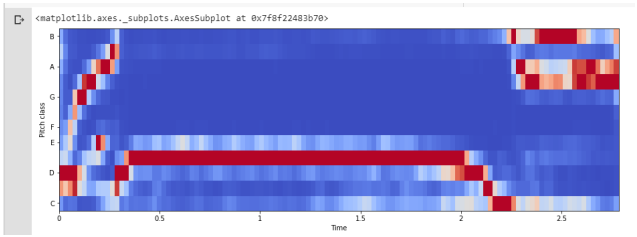


Figure 5.17: The spectrogram plot of audio rate2 by chroma

Now that we understood how we can play around with audio data and extract important features using python. In the following section, we are going to use these features and build a ANN model for music genre classification.

5.1.2 Music genre classification using ANN

AS mentioned above, before leaving this section we are going to wrap up all methods we learned so far for extracting features in the previous subsection. In this regard we will use ANN(artificial neural network) for classification. However, it is better we first understand what is ANN.

Artificial neural networks are one of the main tools used in machine learning. As the "neural" part of their name suggests, they are brain-inspired systems which are intended to replicate the way that we humans learn. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.

Here, the dataset consists of 1000 audio tracks each 30 seconds long. It contains 10 genres, each represented by 100 tracks. The tracks are all 22050 Hz monophonic 16-bit audio files in .wav format.

The dataset can be download from marsyas website and it contains 10 genres i.e

- Blues
- Classical
- Country
- Disco
- Hiphop
- Jazz
- Metal
- Pop
- Reggae
- Rock

Each genre contains 100 songs. Total dataset: 1000 songs, but we will use 330 of which because of technical limitations to just see how it is working.

Befor moving ahead, let go over the road-map:

First of all, we need to convert the audio files into PNG format images(spectrogram). From these spectrogram, we have to extract meaningful features, i.e. MFCCs, Spectral Centroid, Zero Crossing Rate, Chroma Frequencies, Spectral Roll-off.

Once the features have been extracted, they can be appended into a CSV file so that ANN can be used for classification.

However, if we want to work with image data instead of CSV we can use CNN.

```

import librosa
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import os
from PIL import Image
import pathlib
import csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import keras
from keras import layers
from keras import layers
import keras
from keras.models import Sequential
import warnings
```

Now convert the audio data files into PNG format images or basically extracting the spectrogram for every Audio.

```

cmap = plt.get_cmap('inferno')
plt.figure(figsize=(8,8))
genres = 'blues classical country disco hiphop jazz metal pop reggae rock'.
    split()
for g in genres:
    pathlib.Path(f'img_data2/{g}').mkdir(parents=True, exist_ok=True)
```

```
    for filename in os.listdir(f'./drive/My Drive/Data_storage/genres/{g}'):
        songname = f'./drive/My Drive/Data_storage/genres/{g}/{filename}'
        y, sr = librosa.load(songname, mono=True, duration=5)
        plt.specgram(y, NFFT=2048, Fs=2, Fc=0, noverlap=128, cmap=cmap,
    sides='default', mode='default', scale='dB');
        plt.axis('off');
        plt.savefig(f'img_data2/{g}/{filename[:-3].replace(".", "")}.png')
```

Sample spectrogram of a song having genre as blues.

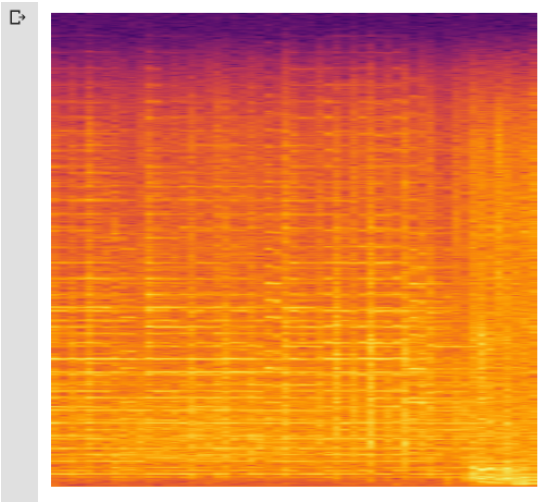


Figure 5.18: The spectrogram of a song having genre as Blues

Now since all the audio files got converted into their respective spectrogram it's easier to extract features.

So let creating a header for our CSV file:

```

header = 'filename chroma_stft rmse spectral_centroid spectral_bandwidth
    rolloff zero_crossing_rate'
for i in range(1, 21):
    header += f' mfcc{i}'
header += ' label'
header = header.split()
```

Extracting features from spectrogram: We will extract Mel-frequency cepstral coefficients (MFCC), Spectral Centroid, Zero Crossing Rate, Chroma Frequencies, and Spectral Roll-off.

```
file = open('dataset.csv', 'w', newline='')
with file:
    writer = csv.writer(file)
    writer.writerow(header)
genres= 'blues classical country disco hiphop jazz metal pop reggae rock'.
    split()
for g in genres:
    pathlib.Path(f'img_data2/{g}').mkdir(parents=True, exist_ok=True)
    for filename in os.listdir(f'./drive/My Drive/Data_storage/genres/{g}'):
        songname = f'./drive/My Drive/Data_storage/genres/{g}/{filename}'
        y, sr4 = librosa.load(songname, mono=True, duration=30)
```

```
        rmse = librosa.feature.rmse(y=y)
        chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr4)
        spec_cent = librosa.feature.spectral_centroid(y=y, sr=sr4)
        spec_bw = librosa.feature.spectral_bandwidth(y=y, sr=sr4)
        rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr4)
        zcr = librosa.feature.zero_crossing_rate(y)
        mfcc = librosa.feature.mfcc(y=y, sr=sr4)
        to_append = f'{filename} {np.mean(chroma_stft)} {np.mean(rmse)} {np.
    mean(spec_cent)} {np.mean(spec_bw)} {np.mean(rolloff)} {np.mean(zcr)}'
        for e in mfcc:
            to_append += f' {np.mean(e)}'
        to_append += f' {g}'
        file = open('dataset.csv', 'a', newline='')
        with file:
            writer = csv.writer(file)
            writer.writerow(to_append.split())
```

Data preprocessing: It involves loading CSV data, label encoding, feature scaling and data split into training and test set.

```
data = pd.read_csv('dataset.csv')
data.head()# Dropping unneccesary columns
data = data.drop(['filename'],axis=1)#Encoding the Labels
genre_list = data.iloc[:, -1]
encoder = LabelEncoder()
y = encoder.fit_transform(genre_list)#Scaling the Feature columns
scaler = StandardScaler()
X = scaler.fit_transform(np.array(data.iloc[:, :-1], dtype = float))#
    Dividing data into training and Testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Building an ANN model:

```

model = Sequential()
model.add(layers.Dense(256, activation='relu', input_shape=(X_train.shape
    [1],)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fit the model:

```
classifier = model.fit(X_train,
                    y_train,
                    epochs=100,
                    batch_size=128)
```

```
Epoch 90/100
256/256 [==============================] - 0s 52us/step - loss: 0.0121 -
    accuracy: 1.0000
Epoch 91/100
256/256 [==============================] - 0s 56us/step - loss: 0.0117 -
    accuracy: 1.0000
Epoch 92/100
```

```
256/256 [==============================] - 0s 47us/step - loss: 0.0113 -
    accuracy: 1.0000
Epoch 93/100
256/256 [==============================] - 0s 49us/step - loss: 0.0110 -
    accuracy: 1.0000
Epoch 94/100
256/256 [==============================] - 0s 53us/step - loss: 0.0107 -
    accuracy: 1.0000
Epoch 95/100
256/256 [==============================] - 0s 61us/step - loss: 0.0104 -
    accuracy: 1.0000
Epoch 96/100
256/256 [==============================] - 0s 53us/step - loss: 0.0101 -
    accuracy: 1.0000
Epoch 97/100
256/256 [==============================] - 0s 57us/step - loss: 0.0098 -
    accuracy: 1.0000
Epoch 98/100
256/256 [==============================] - 0s 57us/step - loss: 0.0096 -
    accuracy: 1.0000
Epoch 99/100
256/256 [==============================] - 0s 51us/step - loss: 0.0093 -
    accuracy: 1.0000
Epoch 100/100
256/256 [==============================] - 0s 38us/step - loss: 0.0090 -
    accuracy: 1.0000
```

Good job, after 100 epochs, Accuracy: 1.

Up to now, we understood how to extract important features and also implemented Artificial Neural Networks(ANN) to classify the music genre.

5.2 Convolution for time series

In contrast to the classical techniques such as ARIMA involved stochastic processes for modelling the data to denoise and / or predict it, there is also the spectral analysis playing a preponderant role in time series analysis. But all these methods rely on the assumption that the data fits a hand-crafted model. A more general approach consists of using state-space models, and relies on *Graphical Model theory*. The core of this type of method is finding the probability distribution of some observations given a stochastic process of latent variables. This task can also be done through a neural network, so in the following, we are going to explore this method known as **Convolution Network** . To begin with, let recall the required libraries:

```

import os
import torch
import numpy as np
import pandas as pd
from torch import nn
from torch import optim
from matplotlib import cm
from itertools import repeat
from google.colab import drive
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

```
from torch.autograd import Variable
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
from torch.utils.data.dataset import Dataset
from sklearn.preprocessing import MinMaxScaler
from matplotlib.dates import MonthLocator, DateFormatter
```

A very classic data type for time series are stock prices. We are going to focus on 'AAPL', 'AMZN', 'GOOGL', 'NVDA', 'GS', 'ZION' and 'FB' from the SP500 data set - which is available here:

```
#loading the dataset
from google.colab import drive
drive.mount('/content/drive')
os.listdir('/content/drive/My Drive/Data_storage/data')
#['^GSPC.csv', '.DS_Store', 'nyse', 'sandp500']
```

Here we are going to prepare data before applying our interested prediction methods. First, we should scale the data, thus we chose the *MinMaxScaler* from the *sklearn library* to scale all the price in the $[0, 1]$ range; the data needs to be scaled back at the end of the experiment in order to get the real estimated price (the sklearn scalers have this inverse transform function included).

For a better visibility, and scalability of the experiments, we chose to inherit the dataset class from Pytorch. Here is the code of my SP500 class for the SP500 dataset:

```
class SP500(Dataset):
    def __init__(self, folder_dataset, T=10, symbols=['AAPL'], use_columns=['Date', 'Close'], start_date='2012-01-01', end_date='2015-12-31', step=1):

        self.scaler = MinMaxScaler()
        self.symbols = symbols
        self.start_date = start_date
        self.end_date = end_date
        self.use_columns = use_columns
        self.T = T

        self.dates = pd.date_range(self.start_date, self.end_date)

        self.df_data = pd.DataFrame(index=self.dates)

        for symbol in symbols:
            # fn = os.path.join(folder_dataset, symbol + "_data.csv")
            fn = "/content/drive/My Drive/Data_storage/" + folder_dataset + "/" + symbol + "_data.csv"
            print(fn)
            df_current = pd.read_csv(fn, index_col='Date', usecols=self.use_columns, na_values='nan', parse_dates=True)
            df_current = df_current.rename(columns={'Close': symbol})
            self.df_data = self.df_data.join(df_current)

        # Replace NaN values with forward then backward filling
        self.df_data.fillna(method='ffill', inplace=True, axis=0)
        self.df_data.fillna(method='bfill', inplace=True, axis=0)

        self.numpy_data = self.df_data[self.symbols].values
        self.train_data = self.scaler.fit_transform(self.numpy_data)
```

```
29
30          # Get history for each data point
31          self.chunks = torch.FloatTensor(self.train_data).unfold(0, self.T,
    step).permute(0, 2, 1)
32
33      def __getitem__(self, index):
34          x = self.chunks[index, :-1, :]
35          y = self.chunks[index, -1, :]
36          return x, y
37
38      def __len__(self):
39          return self.chunks.size(0)
```

We are going to consider for each data point the previous T points as a fixed size sub time sequence, as represented in the graph below when hovering the mouse on it. This is called a next-day prediction.

```
1  class SP500Multistep(Dataset):
2      def __init__(self, folder_dataset, symbols=['AAPL'], use_columns=['Date
       ', 'Close'], start_date='2012-01-01', end_date='2015-12-31', step=1,
       n_in=10, n_out=5):
3
4          self.scaler = MinMaxScaler()
5          self.symbols = symbols
6          self.start_date = start_date
7          self.end_date = end_date
8          self.use_columns = use_columns
9
10         self.dates = pd.date_range(self.start_date, self.end_date)
11         self.df_data = pd.DataFrame(index=self.dates)
12
13         for symbol in symbols:
14             fn = os.path.join(folder_dataset, symbol + "_data.csv")
15             fn = "/content/drive/My Drive/Data_storage/" + folder_dataset +
       "/" + symbol + "_data.csv"
16             print(fn)
17             df_current = pd.read_csv(fn, index_col='Date', usecols=self.
       use_columns, na_values='nan', parse_dates=True)
18             df_current = df_current.rename(columns={'Close': symbol})
19             self.df_data = self.df_data.join(df_current)
20
21         # Replace NaN values with forward then backward filling
22         self.df_data.fillna(method='ffill', inplace=True, axis=0)
23         self.df_data.fillna(method='bfill', inplace=True, axis=0)
24
25         self.numpy_data = self.df_data[self.symbols].values
26         self.train_data = self.scaler.fit_transform(self.numpy_data)
27
28         self.chunks = []
29         self.chunks_data = torch.FloatTensor(self.train_data).unfold(0, n_in
       +n_out, step)
30
31         k = 0
32         while k < self.chunks_data.size(0):
33             self.chunks.append([self.chunks_data[k, :, :n_in], self.
       chunks_data[k, :, n_in:]])
34             k += 1
```

```
35
36      def __getitem__(self, index):
37          x = torch.FloatTensor(self.chunks[index][0])
38          y = torch.FloatTensor(self.chunks[index][1])
39          return x, y
40
41      def __len__(self):
42          return len(self.chunks)
```

Here a first experiment consisted of using 1D convolutions on different stocks, in order to see the effect of dilated convolutions on each series independently. We decided to test the dilated convolution approach first by using n stocks in parallel and perform 1D dilated convolution on each stock independently. The Pytorch library will be used , with input of dimensions

$$n_{batch} \times n_{stocks} \times T$$

```
1  class DilatedNet(nn.Module):
2      def __init__(self, num_securities=5, hidden_size=64, dilation=2, T=10):
3
4          #param num_securities: int, number of stocks
5          #param hidden_size: int, size of hidden layers
6          #param dilation: int, dilation value
7          #param T: int, number of look back points
8
9          super(DilatedNet, self).__init__()
10         self.dilation = dilation
11         self.hidden_size = hidden_size
12
13         self.dilated_conv1 = nn.Conv1d(num_securities, hidden_size,
       kernel_size=2, dilation=self.dilation)
14         self.relu1 = nn.ReLU()
15
16         self.dilated_conv2 = nn.Conv1d(hidden_size, hidden_size, kernel_size
       =1, dilation=self.dilation)
17         self.relu2 = nn.ReLU()
18
19         self.dilated_conv3 = nn.Conv1d(hidden_size, hidden_size, kernel_size
       =1, dilation=self.dilation)
20         self.relu3 = nn.ReLU()
21
22         self.dilated_conv4 = nn.Conv1d(hidden_size, hidden_size, kernel_size
       =1, dilation=self.dilation)
23         self.relu4 = nn.ReLU()
24
25         self.conv_final = nn.Conv1d(hidden_size, num_securities, kernel_size
       =1)
26
27         self.T = T
28
29     def forward(self, x):
30
31         #batch_size x n_stocks x T
32
33         out = self.dilated_conv1(x)
34         out = self.relu1(out)
35
```

```
        out = self.dilated_conv2(out)
        out = self.relu2(out)

        out = self.dilated_conv3(out)
        out = self.relu3(out)

        out = self.dilated_conv4(out)
        out = self.relu4(out)

        out = self.conv_final(out)
        out = out[:, :, -1]

        return out
```

```
learning_rate = 0.001
batch_size = 16
display_step = 500
max_epochs = 1000
symbols = ['AAPL']
# symbols = ['GOOGL', 'AAPL', 'AMZN', 'FB', 'ZION', 'NVDA', 'GS']
n_stocks = len(symbols)
n_hidden1 = 128
n_hidden2 = 128
n_steps_encoder = 20  # time steps, length of time window
n_output = n_stocks
T = 30
start_date = '2013-01-01'
end_date = '2013-10-31'

# training data
dset = SP500('data/sandp500/individual_stocks_5yr', symbols=symbols,
    start_date='2013-01-01', end_date='2013-07-31', T=T, step=1)

train_loader = DataLoader(dset, batch_size=batch_size, shuffle=True,
    num_workers=4, pin_memory=True)

x, y = train_loader.dataset[0]

print(x.shape,y.shape)
```

```
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    AAPL_data.csv
torch.Size([29, 1]) torch.Size([1])
```

```
model = DilatedNet(num_securities=n_stocks, T=T).cuda()
optimizer = optim.RMSprop(model.parameters(), lr=learning_rate, weight_decay
    =0.0)  # n
scheduler_model = lr_scheduler.StepLR(optimizer, step_size=1, gamma=1.0)

# loss function
criterion = nn.MSELoss(size_average=True).cuda()
```

```

losses = []
it = 0
for i in range(max_epochs):
```

```
    loss_ = 0.
    predicted = []
    gt = []
    for batch_idx, (data, target) in enumerate(train_loader):

        data = Variable(data.permute(0, 2, 1)).contiguous()
        target = Variable(target.unsqueeze_(0))

        data = data.cuda()
        target = target.cuda()

        optimizer.zero_grad()

        if target.data.size()[1] == batch_size:

            output = model(data)
            loss = criterion(output, target)
            loss_ += loss.item()

            loss.backward()
            optimizer.step()

            for k in range(batch_size):
                predicted.append(output.data[k, 0])
                gt.append(target.data[:, k, 0])

        it += 1

    print(i, loss_)
    losses.append(loss_)

    scheduler_model.step()

    if i % 20 == 0:
        predicted = np.array(predicted).reshape(-1, 1)
        gt = np.array(gt).reshape(-1, 1)
        x = np.array(range(predicted.shape[0]))
        h = plt.figure()
        plt.plot(x, predicted[:, 0], label="predictions")
        plt.plot(x, gt[:, 0], label="true")
        plt.legend()
        plt.show()
```

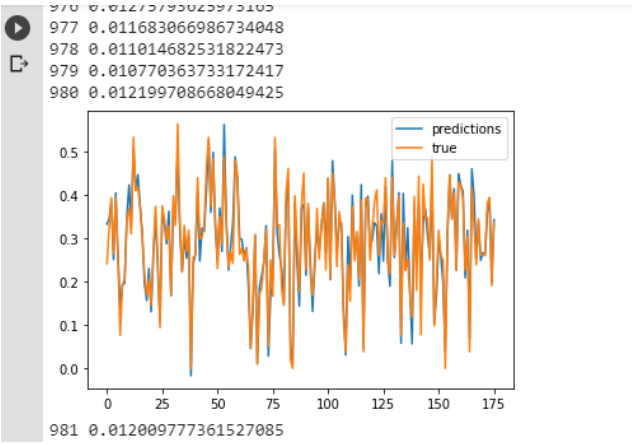


Figure 5.19: The plot of curve for the signal rate

```
# torch.save(model, 'dilated_net_1d.pkl')

h = plt.figure()
x = np.arange(len(losses))
plt.plot(x, np.array(losses), label="loss")
plt.legend()
plt.show()
```

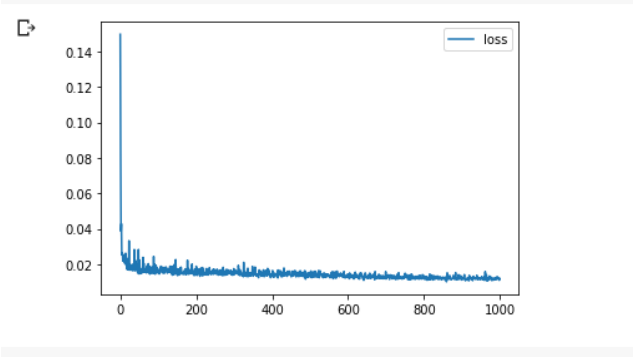


Figure 5.20: The loss plot

```
# TEST
predictions = np.zeros((len(train_loader.dataset.chunks), n_stocks))
ground_tr = np.zeros((len(train_loader.dataset.chunks), n_stocks))
batch_size_pred = 4

dtest = SP500('data/sandp500/individual_stocks_5yr', symbols=symbols,
    start_date='2013-01-01', end_date='2013-10-31', T=T)
test_loader = DataLoader(dtest, batch_size=batch_size_pred, shuffle=False,
    num_workers=4, pin_memory=True)


predictions = [[] for i in repeat(None, len(symbols))]
```

```
gts = [[] for i in repeat(None, len(symbols))]
k = 0



for batch_idx, (data, target) in enumerate(test_loader):
    data = Variable(data.permute(0, 2, 1)).contiguous()
    target = Variable(target.unsqueeze_(1))

    data = data.cuda()
    target = target.cuda()

    if target.data.size()[0] == batch_size_pred:
        output = model(data)
        for k in range(batch_size_pred):
            s = 0
            for stock in symbols:
                predictions[s].append(output.data[k, s])
                gts[s].append(target.data[k, 0, s])
                s += 1
        k += 1



if len(symbols) == 1:
    pred = dtest.scaler.inverse_transform(np.array(predictions[0]).reshape((
    len(predictions[0]), 1)))
    gt = dtest.scaler.inverse_transform(np.array(gts[0]).reshape(len(gts[0])
    , 1))
if len(symbols) >= 2:
    p = np.array(predictions)
    pred = dtest.scaler.inverse_transform(np.array(predictions).transpose())
    gt = dtest.scaler.inverse_transform(np.array(gts).transpose())

x = np.array(range(pred.shape[0]))
x = [np.datetime64(start_date) + np.timedelta64(x, 'D') for x in range(0,
    pred.shape[0])]
x = np.array(x)
months = MonthLocator(range(1, 10), bymonthday=1, interval=1)
monthsFmt = DateFormatter("%b '%y")

s = 0

for stock in symbols:
    fig, ax = plt.subplots()
    plt.plot(x, pred[:, s], label="predictions", color=cm.Blues(300))
    plt.plot(x, gt[:, s], label="true", color=cm.Blues(100))
    ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
    ax.xaxis.set_major_locator(months)
    ax.xaxis.set_major_formatter(monthsFmt)
    plt.title(stock)
    plt.xlabel("Time (2013-01-01 to 2013-10-31)")
    plt.ylabel("Stock Price")
    plt.legend()
    fig.autofmt_xdate()
    plt.show()
```

```
64      s += 1
```

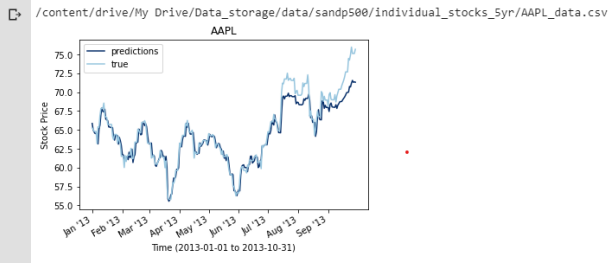


Figure 5.21: The predicted stock price of AAPL data for first 10 monthes of 2013

Then, in order to take into account the correlation between the series, 2D convolutions will be used, but dilated only on the time axis to get this "time multi scale" aspect. Again, I used Pytorch to implement this network, and used inputs of size

$$n_{batch} \times 1 \times n_{stocks} \times T$$

. The **1** in the previous dimensions corresponds to the fact that we only used closing prices for estimating stock prices, but other features could easily be added on that dimension in order to take into account other metrics, like the highest/lowest values or the volume.

```
1
2   class DilatedNet2D(nn.Module):
3       def __init__(self, hidden_size=64, dilation=1, T=10):
4           """
5
6           :param hidden_size: int, size of hidden layers
7           :param dilation: int, dilation value in the time dimension (1 for
        the other dimension, aka between the stocks)
8           :param T: int, number of look back points
9           """
10          super(DilatedNet2D, self).__init__()
11          self.dilation = dilation
12          self.hidden_size = hidden_size
13
14          self.dilated_conv1 = nn.Conv2d(1, hidden_size, kernel_size=(1, 2),
        dilation=(1, self.dilation))
15          self.relu1 = nn.ReLU()
16
17          self.dilated_conv2 = nn.Conv2d(hidden_size, hidden_size, kernel_size
        =(1, 2), dilation=(1, self.dilation))
18          self.relu2 = nn.ReLU()
19
20          self.dilated_conv3 = nn.Conv2d(hidden_size, hidden_size, kernel_size
        =(1, 2), dilation=(1, self.dilation))
21          self.relu3 = nn.ReLU()
22
23          self.dilated_conv4 = nn.Conv2d(hidden_size, hidden_size, kernel_size
        =(1, 2), dilation=(1, self.dilation))
24          self.relu4 = nn.ReLU()
25
```

```
        self.conv_final = nn.Conv2d(hidden_size, 1, kernel_size=(1, 2))

        self.T = T

    def forward(self, x):

        #batch_size x 1 x n_stocks x T

        out = self.dilated_conv1(x)
        out = self.relu1(out)

        out = self.dilated_conv2(out)
        out = self.relu2(out)

        out = self.dilated_conv3(out)
        out = self.relu3(out)

        out = self.dilated_conv4(out)
        out = self.relu4(out)

        out = self.conv_final(out)
        out = out[:, :, :, -1]

        return out
```

```

/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    GOOGL_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    AAPL_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    AMZN_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    FB_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    ZION_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    NVDA_data.csv
/content/drive/My Drive/Data_storage/data/sandp500/individual_stocks_5yr/
    GS_data.csv
```

Let set the model:

```
model = DilatedNet2D(T=T, hidden_size=64, dilation=2)
model = model.cuda()
optimizer = optim.RMSprop(model.parameters(), lr=learning_rate, weight_decay
    =weight_decay)
scheduler_model = lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.9)
```

```
# loss function
criterion = nn.MSELoss(size_average=False).cuda()
# Store successive losses
losses = []
for i in range(max_epochs):
    loss_ = 0.
    predicted = []
    gt = []
```

```
    # Go through training data set
    for batch_idx, (data, target) in enumerate(train_loader):
        data = Variable(data.permute(0, 2, 1)).unsqueeze_(1).contiguous()
        target = Variable(target.unsqueeze_(1))

        data = data.cuda()
        target = target.cuda()

        if target.data.size()[0] == batch_size:
            # Set gradient of optimizer to 0
            optimizer.zero_grad()
            # Compute predictions
            output = model(data)
            # Compute loss
            loss = criterion(output, target)
            loss_ += loss.item()
            # Backpropagation
            loss.backward()
            # Gradient descent step
            optimizer.step()
            # Store current results for visual check
            for k in range(batch_size):
                predicted.append(output.data[k, 0, :].cpu().numpy())
                gt.append(target.data[k, 0, :].cpu().numpy())

    print("Epoch = ", i)
    print("Loss = ", loss_)
    losses.append(loss_)
    # Store for display in Tensorboard

    # Apply step of scheduler for learning rate change
    scheduler_model.step()
    # Plot current predictions
    if i % display_step == 0:
        predicted = np.array(predicted)
        gt = np.array(gt)
        x = np.array(range(predicted.shape[0]))
        h = plt.figure()
        plt.plot(x, predicted[:, 0], label="predictions")
        plt.plot(x, gt[:, 0], label="true")
        plt.legend()
        plt.show()
```

The result for 10 last epochs:

```
Epoch =  989
Loss =  0.9569155871868134
Epoch =  990
Loss =  0.9449984431266785
Epoch =  991
Loss =  0.9500388205051422
Epoch =  992
Loss =  0.8583411872386932
Epoch =  993
Loss =  0.9369369745254517
Epoch =  994
Loss =  0.9504155814647675
```

```
Epoch =  995
Loss =  0.9778002798557281
Epoch =  996
Loss =  0.9268246293067932
Epoch =  997
Loss =  0.9083327651023865
Epoch =  998
Loss =  0.9564555883407593
Epoch =  999
Loss =  0.922675758600235
```

The loss plot:

```

h = plt.figure()
x = range(len(losses))
plt.plot(np.array(x), np.array(losses), label="loss")
plt.xlabel("Time")
plt.ylabel("Training loss")
plt.legend()
plt.show()
```

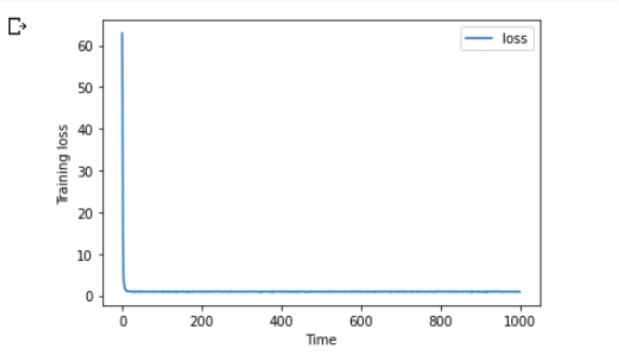


Figure 5.22: The loss plot

```
# TEST
predictions = np.zeros((len(train_loader.dataset.chunks), n_stocks))
ground_tr = np.zeros((len(train_loader.dataset.chunks), n_stocks))
batch_size_pred = batch_size

# Create test data set
start_date = '2013-01-01'
end_date = '2017-10-31'
dtest = SP500('data/sandp500/individual_stocks_5yr', symbols=symbols,
    start_date=start_date, end_date=end_date, T=T)
test_loader = DataLoader(dtest, batch_size=batch_size_pred, shuffle=False,
    num_workers=4, pin_memory=True)

# Create list of n_stocks lists for storing predictions and GT
predictions = [[] for i in repeat(None, len(symbols))]
gts = [[] for i in repeat(None, len(symbols))]
k = 0

```

```
# Predictions
for batch_idx, (data, target) in enumerate(test_loader):
    data = Variable(data.permute(0, 2, 1)).unsqueeze_(1).contiguous()
    target = Variable(target.unsqueeze_(1))

    data = data.cuda()
    target = target.cuda()

    k = 0
    if target.data.size()[0] == batch_size_pred:
        output = model(data)
        for i in range(batch_size_pred):
            s = 0
            for stock in symbols:
                predictions[s].append(output.data[i, 0, s])
                gts[s].append(target.data[i, 0, s])
                s += 1
            k += 1


if len(symbols) == 1:
    pred = dtest.scaler.inverse_transform(np.array(predictions[0]).reshape((
    len(predictions[0]), 1)))
    gt = dtest.scaler.inverse_transform(np.array(gts[0]).reshape(len(gts[0])
    , 1))
if len(symbols) >= 2:
    p = np.array(predictions)
    pred = dtest.scaler.inverse_transform(np.array(predictions).transpose())
    gt = dtest.scaler.inverse_transform(np.array(gts).transpose())



x = [np.datetime64(start_date) + np.timedelta64(x, 'D') for x in range(0,
    pred.shape[0])]
x = np.array(x)
months = MonthLocator(range(1, 10), bymonthday=1, interval=3)
monthsFmt = DateFormatter("%b '%y")
s = 0
for stock in symbols:
    fig, ax = plt.subplots(figsize=(8, 6), dpi=100)
    plt.plot(x, pred[:, s], label="predictions", color=cm.Blues(300))
    plt.plot(x, gt[:, s], label="true", color=cm.Blues(100))
    ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
    ax.xaxis.set_major_locator(months)
    ax.xaxis.set_major_formatter(monthsFmt)
    plt.title(stock)
    plt.xlabel("Time")
    plt.ylabel("Stock Price")
    plt.legend()
    fig.autofmt_xdate()
    plt.show()
    s += 1
```

The result of the prediction for each company as follow:

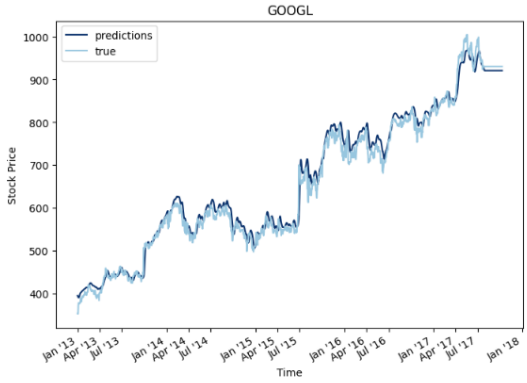


Figure 5.23: The predicted stock price of GOOGLE

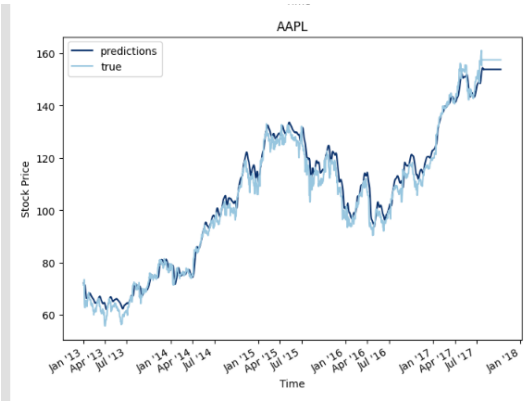


Figure 5.24: The predicted stock price of Apple

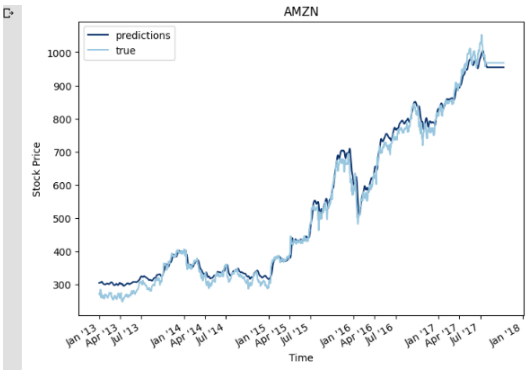


Figure 5.25: The predicted stock price of Amazon

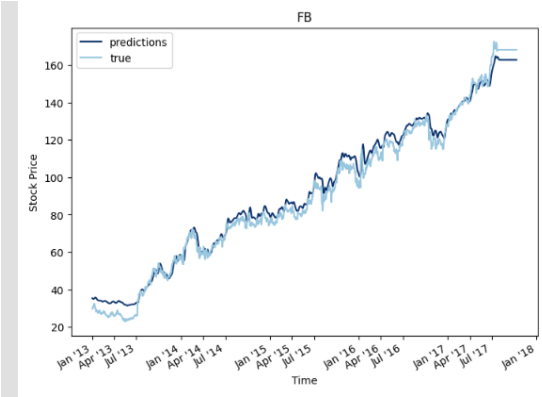


Figure 5.26: The predicted stock price of Facebook

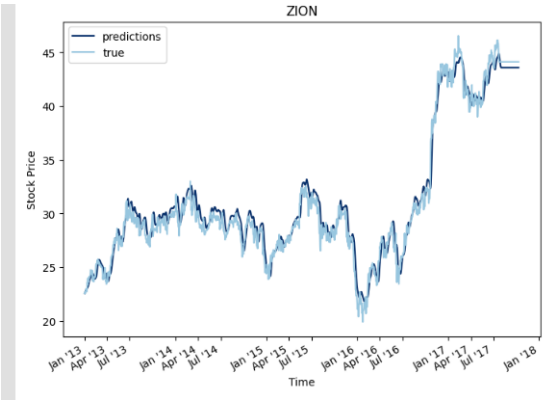


Figure 5.27: The predicted stock price of Zion

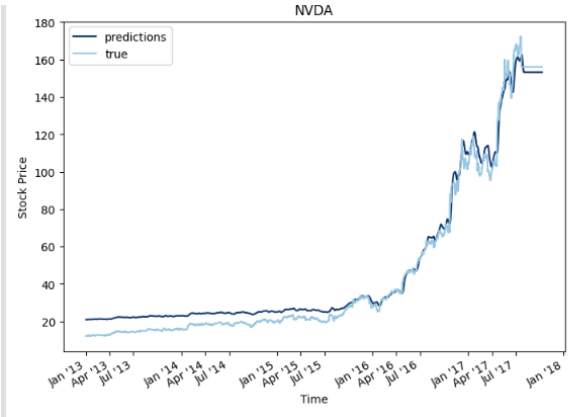


Figure 5.28: The predicted stock price of Nvda

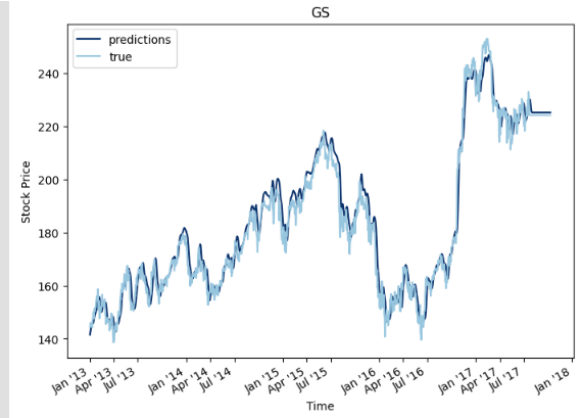


Figure 5.29: The predicted stock price of GS

It is notable that the 2D dilated convolutions perform better than the 1D dilated convolutions. It is very likely that the structure is more taken into account in the 2D convolutions and hence, if any correlation exist between the series, the convolution on the stocks dimension will capture some of it.

2D Dilated Multistep Model:

```
1
2  class DilatedNet2DMultistep(nn.Module):
3      def __init__(self, num_securities=5, n_in=20, n_out=3, hidden_size=64,
       dilation=1, T=10):
4          """
5
6          :param num_securities:
7          :param n_in: number of time points in the input
8          :param n_out: number of time points in the output
9          :param hidden_size: int
10         :param dilation: int
11         :param T: int, length of lookback
12         """
13         super(DilatedNet2DMultistep, self).__init__()
14         self.n_out = n_out
15         self.n_in = n_in
16         self.dilation = dilation
17         self.hidden_size = hidden_size
18         # First Layer
19         # Input
20         self.dilated_conv1 = nn.Conv2d(1, hidden_size, kernel_size=(1, 2),
       dilation=(1, self.dilation))  # dilation in
21
                                 # the time dimension
22         self.relu1 = nn.ReLU()
23
24         # Layer 2
25         self.dilated_conv2 = nn.Conv2d(hidden_size, hidden_size, kernel_size
       =(1, 2), dilation=(1, self.dilation))
26         self.relu2 = nn.ReLU()
27
28         # Layer 3
29         self.dilated_conv3 = nn.Conv2d(hidden_size, hidden_size, kernel_size
```

```
    =(1, 2), dilation=(1, self.dilation))
        self.relu3 = nn.ReLU()

        # Layer 4
        self.dilated_conv4 = nn.Conv2d(hidden_size, hidden_size, kernel_size
    =(1, 2), dilation=(1, self.dilation))
        self.relu4 = nn.ReLU()

        # Output layer
        self.conv_final = nn.Conv2d(hidden_size, 1, kernel_size=(1, 2))

        self.T = T

    def forward(self, x):
        """

        :param x: Pytorch Variable, batch_size x 1 x T x n_stocks
        :return:
        """

        # First layer
        out = self.dilated_conv1(x)
        out = self.relu1(out)

        # Layer 2:
        out = self.dilated_conv2(out)
        out = self.relu2(out)

        # Layer 3:
        out = self.dilated_conv3(out)
        out = self.relu3(out)

        # Layer 4:
        out = self.dilated_conv4(out)
        out = self.relu4(out)

        # Final layer
        out = self.conv_final(out)
        out = out[:, :, :, -self.n_out:]

        return out
```

```

learning_rate = 0.001
batch_size = 5
display_step = 200
max_epochs = 1000
symbols = ['GOOGL', 'AAPL', 'AMZN', 'FB', 'ZION', 'NVDA', 'GS']
n_stocks = len(symbols)
n_hidden1 = 128
n_hidden2 = 128
n_steps_encoder = 20  # time steps, length of time window
n_output = n_stocks
T = 5
start_date = '2013-01-01'
end_date = '2013-12-31'
n_step_data = 20
```

```
n_out = 20
n_in = 30


# training data
dset = SP500Multistep('data/sandp500/individual_stocks_5yr',symbols=symbols,
     start_date=start_date, end_date=end_date, step=n_step_data, n_in=n_in,
    n_out=n_out)
train_loader = DataLoader(dset, batch_size=batch_size, shuffle=False,
    num_workers=4, pin_memory=True)
```

```
model = DilatedNet2DMultistep(num_securities=n_stocks, T=T, n_in=n_in, n_out
    =n_out).cuda()
optimizer = optim.RMSprop(model.parameters(), lr=learning_rate, weight_decay
    =0.0)  # n
scheduler_model = lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.9)

criterion = nn.MSELoss(size_average=True).cuda()
```

```

losses = []
for i in range(max_epochs):
    loss_ = 0.
    predicted = []
    gt = []
    for batch_idx, (data, target) in enumerate(train_loader):
        data = Variable(data).unsqueeze_(1).contiguous()
        target = Variable(target.unsqueeze_(1))

        data = data.cuda()
        target = target.cuda()

        optimizer.zero_grad()
        if target.data.size()[0] == batch_size:
            output = model(data)
            loss = criterion(output, target)
            loss_ += loss.item()
            loss.backward()
            optimizer.step()
            for k in range(batch_size):
                predicted.append(output.data[k, 0, :, :].cpu().numpy())
                gt.append(target.data[k, 0, :, :].cpu().numpy())

    print(i, loss_)
    losses.append(loss_)

    scheduler_model.step()
    # Plot current predictions
    if i % display_step == 0:
        gt = np.array(gt)
        series = dset.train_data
        xs = np.array(range(series.shape[0]))
        h = plt.figure()
        plt.plot(xs, series[:, 0])

        for t in range(len(predicted)):
            xaxis = [x for x in range(t*n_step_data, t*n_step_data + n_out)]
```

```
            yaxis = predicted[t][0, :]
            plt.plot(xaxis, yaxis)
        plt.legend()
        plt.show()
```

The last 10 of epochs:

```
989 0.052982025779783726
990 0.052982025779783726
991 0.052982025779783726
992 0.052982025779783726
993 0.052982025779783726
994 0.052982025779783726
995 0.052982025779783726
996 0.052982025779783726
997 0.052982025779783726
998 0.052982025779783726
999 0.052982025779783726
```

```

h = plt.figure()
x = np.arange(len(losses))
plt.plot(x, np.array(losses), label="loss")
plt.xlabel("Time")
plt.ylabel("Stock Price")
plt.legend()
plt.show()
```

```
# TEST
predictions = np.zeros((len(train_loader.dataset.chunks), n_stocks))
ground_tr = np.zeros((len(train_loader.dataset.chunks), n_stocks))
batch_size_pred = batch_size

# Create test data set
start_date = '2013-01-01'
end_date = '2017-10-31'

dtest=SP500Multistep('data/sandp500/individual_stocks_5yr', symbols=symbols,
     start_date=start_date, end_date=end_date, step=n_step_data, n_in=n_in,
    n_out=n_out)
test_loader=DataLoader(dtest, batch_size=batch_size_pred, shuffle=False,
    num_workers=4, pin_memory=True)

predictions = []
gts = []
k = 0

# Predictions
for batch_idx, (data, target) in enumerate(test_loader):
    data = Variable(data).unsqueeze_(1).contiguous()
    target = Variable(target.unsqueeze_(1))

    data = data.cuda()
    target = target.cuda()

    k = 0
    if target.data.size()[0] == batch_size_pred:
```

```
        output = model(data)
        for i in range(batch_size_pred):
            predictions.append(output.data[i, 0, :, :].cpu().numpy())
            gts.append(target.data[i, 0, :, :].cpu().numpy())



# Convert lists to np array for plot, and rescaling to original data

x = [np.datetime64(start_date) + np.timedelta64(x, 'D') for x in range(0,
    len(predictions))]
x = np.array(x)
months = MonthLocator(range(1, 10), bymonthday=1, interval=3)
monthsFmt = DateFormatter("%b '%y")
s = 0
series = dtest.train_data
predictions = np.rollaxis(np.dstack(predictions), -1)

for stock in symbols:
    fig, ax = plt.subplots(figsize=(10, 8), dpi=120)
    xs = np.array(range(series.shape[0]))
    plt.plot(xs, series[:, s])

    for t in range(len(predictions)):
        xaxis = [x for x in range(t * n_step_data, t * n_step_data + n_out)]
        yaxis = predictions[t, s, :]
        plt.plot(xaxis, yaxis)

    #TODO(louise) add dates on axis for next post
    # ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
    # ax.xaxis.set_major_locator(months)
    # ax.xaxis.set_major_formatter(monthsFmt)
    plt.title(stock)
    plt.xlabel("Time")
    plt.ylabel("Stock Price")
    # plt.legend()
    fig.autofmt_xdate()
    plt.show()
    s += 1
```

The result of prediction by Dilated Multistep method for each company:

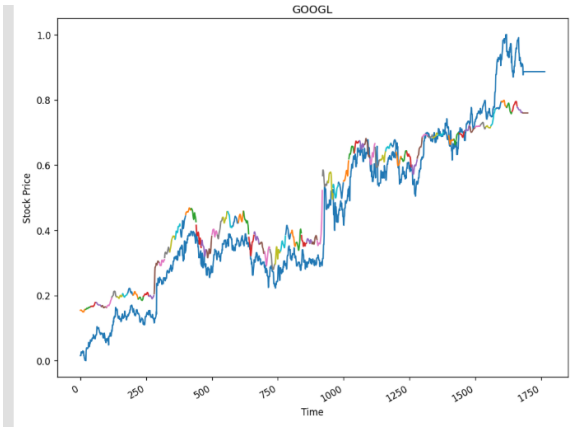


Figure 5.30: The predicted stock price of GS

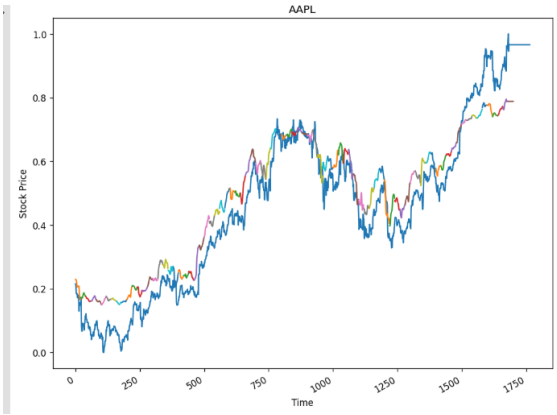


Figure 5.31: The predicted stock price of GS

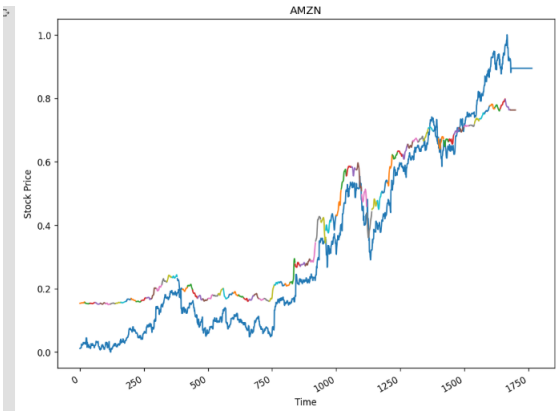


Figure 5.32: The predicted stock price of GS

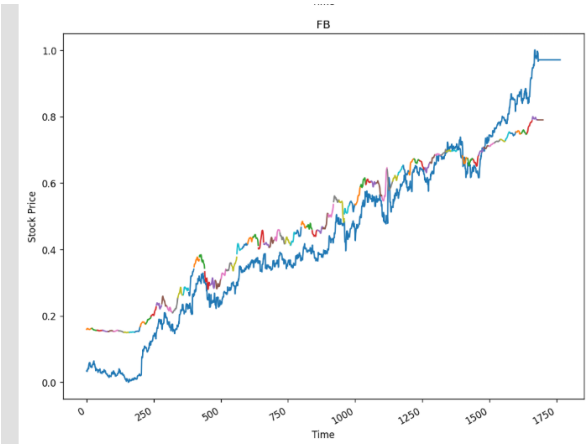


Figure 5.33: The predicted stock price of GS

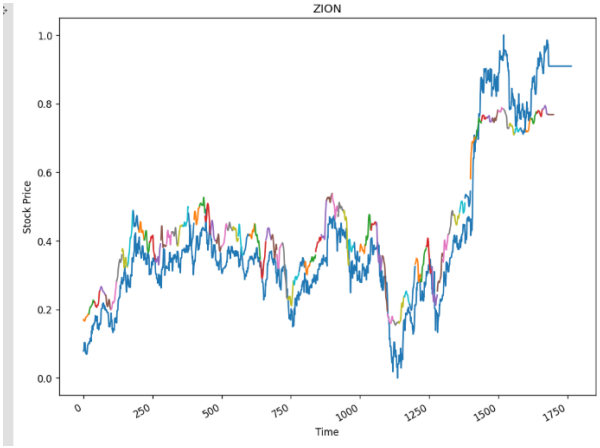


Figure 5.34: The predicted stock price of GS

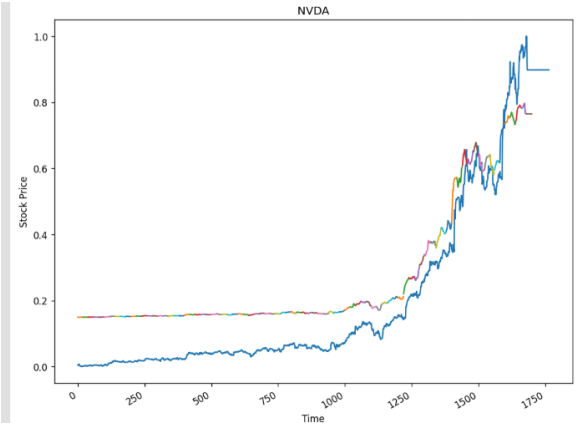


Figure 5.35: The predicted stock price of GS

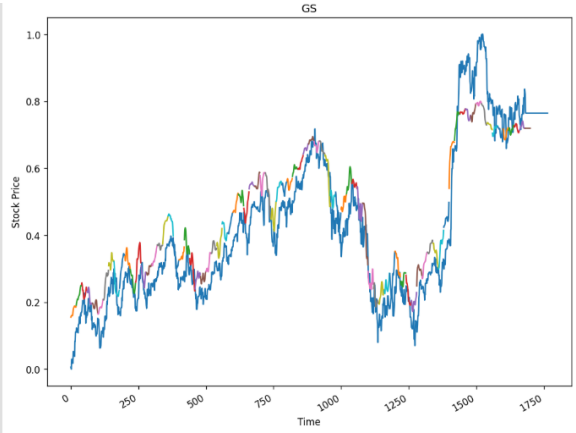


Figure 5.36: The predicted stock price of GS

5.3 Fourier

In this section, introduce the Fourier Transform and its application, and how we can use it for convolution neural network.

The Fourier Transformation is applied in engineering to determine the dominant frequencies in a vibration signal. When the dominant frequency of a signal corresponds with the natural frequency of a structure, the occurring vibrations can get amplified due to resonance. This can happen to such a degree that a structure may collapse.

Now say I have bought a new sound system and the natural frequency of the window in my living room is about 100 Hz. Let's use the Fourier Transform and examine if it is safe to turn Kendrick Lamar's song 'Alright' on full volume.

Time signal

The Fourier transform is commonly used to convert a signal in the time spectrum to a frequency spectrum. Examples of time spectra are sound waves, electricity, mechanical vibrations etc. The figure below shows 0,25 seconds of Kendrick's tune. As can clearly be seen it looks like a wave with different frequencies. Actually it looks like multiple waves.

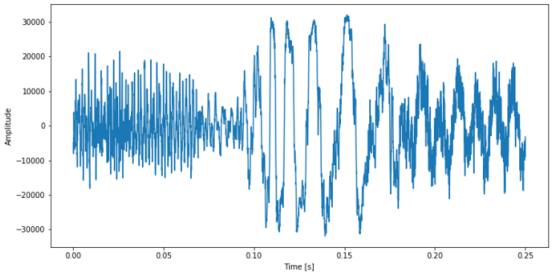


Figure 5.37: Time spectrum Kendrick Lamar - Alright

Fourier transform

This is where the Fourier Transform comes in. This method makes use of the fact that every non-linear function can be represented as a sum of (infinite) sine waves. In the underlying figure this

is illustrated, as a step function is simulated by a multitude of sine waves.

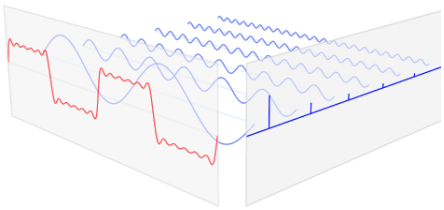


Figure 5.38: Step function simulated with sine waves

A Fourier Transform will break apart a time signal and will return information about the frequency of all sine waves needed to simulate that time signal. For sequences of evenly spaced values the Discrete Fourier Transform (DFT) is defined as

$$X_k = \sum_{n=1}^{N-1} x_n e^{2\pi i k n / N}$$

where:

- N = number of samples
- x_n = value of the signal at time n
- k = current frequency (0 Hz to N-1 Hz)
- X_k = Result of the DFT (amplitude and phase)

Note that a dot product is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

A DFT algorithm can thus be as written as:

```
import numpy as np

def DFT(x):
    """
    Compute the discrete Fourier Transform of the 1D array x
    :param x: (array)
    """

    N = x.size
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    return np.dot(e, x)
```

However, if we run this code on our time signal, which contains approximately 10,000 values, it takes over 10 seconds to compute.

Exploring the FFT Let's write some code to find out what an FFT is actually doing.

First we define a simple signal containing an addition of two sine waves. One with a frequency of 40 Hz and one with a frequency of 90 Hz.

```
t = np.linspace(0, 0.5, 500)
s = np.sin(40 * 2 * np.pi * t) + 0.5 * np.sin(90 * 2 * np.pi * t)

plt.ylabel("Amplitude")
plt.xlabel("Time [s]")
plt.plot(t, s)
plt.show()
```

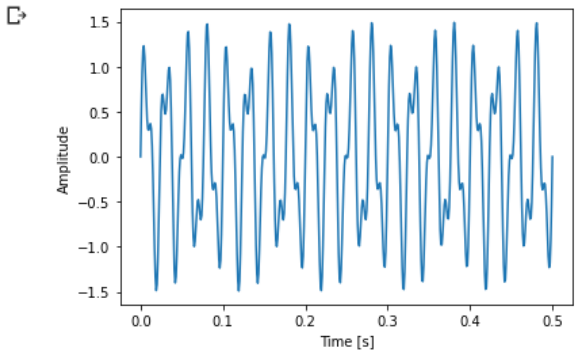


Figure 5.39: The curve of the combined sine waves

In order to retrieve a spectrum of the frequency of the time signal mentioned above we must take a FFT on that sequence.

```
fft = np.fft.fft(s)


for i in range(2):
    print("Value at index {}:\t{}".format(i, fft[i + 1]), "\nValue at index
    {}:\t{}".format(fft.size -1 - i, fft[-1 - i]))
```

The output:

```
Value at index 0: (0.0003804834928392009-0.0605550317619008j)
Value at index 499: (0.00038048349284042216+0.0605550317619024j)
Value at index 1: (0.001531771483137212-0.12188808528069521j)
Value at index 498: (0.0015317714831376839+0.12188808528069531j)
```

In the above code snippet the FFT result of the two sine waves is determined. The first two and the last two values of the FFT sequence were printed to stdout. As we can see we get complex numbers as a result. If we compare the first value of the sequence (index 0) with the last value of the sequence (index 499) we can see that the real parts of both numbers are equal and that the value of the imaginary numbers are also equal in magnitude, only one is positive and the other is negative. The numbers are each others complex conjugate. This is true for all numbers in the sequence:

For real number inputs is **n** the complex conjugate of **N - n**.

Because the second half of the sequence gives us no new information we can already conclude that the half of the FFT sequence is the output we need.

The complex output numbers of the FFT contains the following information:

- **Amplitude** of a certain frequency sine wave (energy).

- **Phase** offset of a certain frequency sine wave.

The amplitude is retrieved by taking the absolute value of the number and the phase offset is obtained by computing the angle of the number.

Spectrum

We are interested in the energy of each frequency, so we can determine the absolute value of the FFT's output. To get a good insight in the spectrum the energy should be plotted against the frequency. Each discrete number output of the FFT corresponds to a certain frequency. The frequency resolution is determined by:

$$\triangle f = \frac{f_s}{N}$$

Putting it all together we can plot the frequency spectrum for our simple sine wave function. We plot only half of the spectrum, because that is the only half giving us real information.

```
1
2  fft = np.fft.fft(s)
3  T = t[1] - t[0]  # sampling interval
4  N = s.size
5
6  # 1/T = frequency
7  f = np.linspace(0, 1 / T, N)
8
9  plt.ylabel("Amplitude")
10 plt.xlabel("Frequency [Hz]")
11 plt.bar(f[:N // 2], np.abs(fft)[:N // 2] * 1 / N, width=1.5)  # 1 / N is a
       normalization factor
12 plt.show()
```

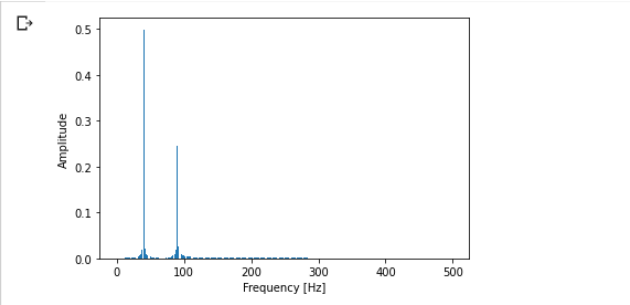


Figure 5.40: The FFt plot for sine waves

As we can see the FFT works! It has given us information about the frequencies of the waves in the time signal.

A FFT is a trade-off between time information and frequency information. By taking a FFT of a time signal, all time information is lost in return for frequency information. To keep information about time and frequencies in one spectrum, we must make a spectrogram. These are DFT's taken on discrete time windows.

By taking a FFT result of the time signal of Kendrick Lamar's song, we get the spectrum shown below. The frequency scale is plotted on log scale. As we assumed before the natural frequency of my windows are about 100 Hz. In the figure we can see that the most dominant frequencies occur

between $10^{1.5} - 10^{2.2}$ Hz (30-158 Hz). My windows natural frequency is right in the middle of the dominant frequencies of the song and thus may resonate due to the high volume.

Now it is too premature to say it wouldn't be safe to listen to this song on full volume. However if I really want to be sure about my windows I maybe should examine the frequency of another song.

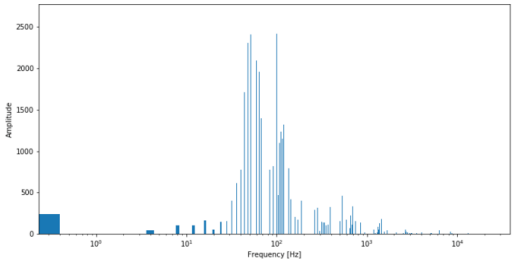


Figure 5.41: The FFt plot of Kendrick Lamar's Alright

However, I am going to finish this section by sharing the application of fast fourier transformation(FFT) on convolution.

The first question comes to mind is that what is the advantage of using FFT on convlotion?

The answer is that FFT convolution uses the principle that multiplication in the frequency domain corresponds to convolution in the time domain. The input signal is transformed into the frequency domain using the DFT, multiplied by the frequency response of the filter, and then transformed back into the time domain using the Inverse DFT. This basic technique was known since the days of Fourier; however, no one really cared. This is because the time required to calculate the DFT was longer than the time to directly calculate the convolution. This changed in 1965 with the development of the Fast Fourier Transform (FFT). By using the FFT algorithm to calculate the DFT, convolution via the frequency domain can be faster than directly convolving the time domain signals. The final result is the same; only the number of calculations has been changed by a more efficient algorithm. For this reason, FFT convolution is also called high-speed convolution.

Let illustrate the technique through the following example:

```
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt
```

The original image:

```

img = plt.imread('/content/sample_data/elephant.png')
plt.figure()
plt.imshow(img)
```

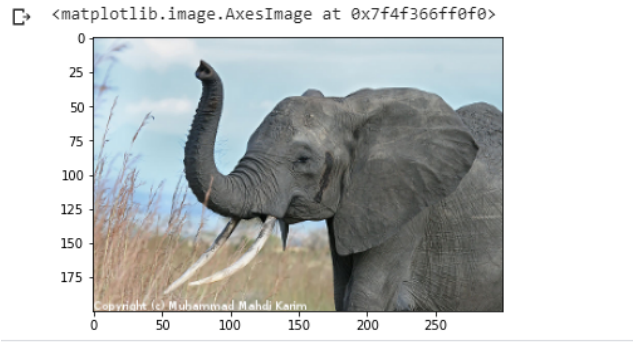


Figure 5.42: The original image

Prepare an Gaussian convolution kernel:

```
#Prepare an Gaussian convolution kernel
# First a 1-D  Gaussian
t = np.linspace(-10, 10, 30)
bump = np.exp(-0.1*t**2)
bump /= np.trapz(bump) # normalize the integral to 1

# make a 2-D kernel out of it
kernel = bump[:, np.newaxis] * bump[np.newaxis, :]
```

Implement convolution via FFT Padded fourier transform, with the same shape as the image. We use :func:‘scipy.signal.fftpack.fft2‘ to have a 2D FFT

```

kernel_ft = fftpack.fft2(kernel, shape=img.shape[:2], axes=(0, 1))

# convolve
img_ft = fftpack.fft2(img, axes=(0, 1))
# the 'newaxis' is to match to color direction
img2_ft = kernel_ft[:, :, np.newaxis] * img_ft
img2 = fftpack.ifft2(img2_ft, axes=(0, 1)).real

# clip values to range
img2 = np.clip(img2, 0, 1)

# plot output
plt.figure()
plt.imshow(img2)
```

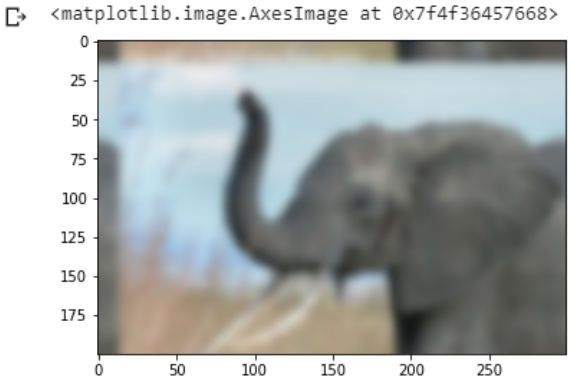


Figure 5.43: The Gaussian convolution kernel

A “wrapped border” appears in the upper left and top edges of the image. This is because the padding is not done correctly, and does not take the kernel size into account (so the convolution “flows out of bounds of the image”). But let try to remove this artifact:

The above issue was only for didactic reasons: there exists a function in scipy that will do this for us, and probably do a better job: scipy.signal.fftconvolve()

```
from scipy import signal
# mode='same' is there to enforce the same output shape as input arrays
# (ie avoid border effects)
img3 = signal.fftconvolve(img, kernel[:, :, np.newaxis], mode='same')
plt.figure()
plt.imshow(img3)
```

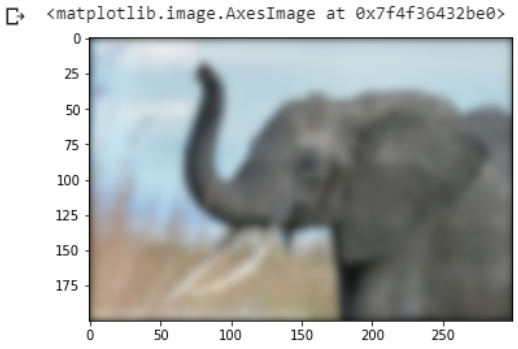


Figure 5.44: The scipy fft convolution image

6. References

- Hahn W.2020, Applied Times Series Course, Lecture Notes and Notebooks, Florida Atlantic University.
- Aileen Nielsen,2020, Practical Time Series Analysis,
- Wes McKinney,2018, Python for Data Analysis, Data Wrangling with Pandas, NumPy,and IPython.
- Avishek Pal PKS Prakash, 2017, Practical Time Series Analysis.
- N.D Lewis.2016, Deep Time Series Forcatsing With Python.
- Jamie Chan,2014, Learn Python in One Day and Learn It Well.
- Jonathan D. Cryer • Kung-Sik Chan, 2008, Time Series Analysis With Applications in R.
- Kaggle's Online Notebooks