Python Tutorial

Daniel Zeiberg- zeiberg.d@northeastern.edu

March 15, 2020

1 Installation Prerequsites

1.1 Anaconda Installation

Anaconda / Conda

- Platform that allows you to manage python versions and packages through "environments"
- Includes pip - python's package manager that can be used to expand the capabilities of a base python environment
- Each environment can run different versions of python
- Each environment has a different set of installed packages

Install Link: https://www.anaconda.com/distribution/

- Download Anaconda Python 3.7 version
- Use default settings for installation
- If your username has a space in it, anaconda will warn that this might cause install issues

1.2 Create Conda Environment

Windows: Open "Anaconda Prompt" through Start menu

Mac: Open "Terminal" through spotlight search

Enter the following command to create a new environment named "py37" with python version 3.7

```
conda create --name py37 python=3.7
```

Activate the environment you just created

```
conda activate py37
```

Conda Cheet Sheet is a useful reference: http://bit.ly/2xknudL

1.3 Install Packages

Many features are already included in python, but many packages can be installed to expand the capabilities of python

These packages can be installed using pip - python's package installer - or conda

1.3.1 jupyter-notebook

- a web-based interactive development environment (IDE) that allows you to interactively develop python scripts, analyze and visualize data
- A notebook is comprised of a series of cells; can either be a python code cell or a markdown text cell
- Can execute the contents of the active cell by clicking (Shift+Enter)
- Code cells are run in the order you execute them Install jupyter-notebook and add our conda environment as a python kernel (the python environment in which we'll run our code)

```
pip install jupyter
python -m ipykernel install --user --name=py37
```

1.3.2 Matplotlib

- A package for creating visualizations

```
pip install matplotlib
```

1.3.3 Pandas

- Used for working with datasets
- Load data into pandas DataFrame which are similar to database tables
- Can query large datasets

```
pip install pandas
```

1.3.4 Numpy

- Package providing useful tools for mathematical operations and analyzing multidimensional datasets

```
pip install numpy
```

1.3.5 Sklearn

- Package providing tools to quickly train and analyze machine learning models

```
pip install sklearn
```

1.4 Start-up jupyter

In Terminal/ Anaconda Prompt run:

```
jupyter-notebook
```

This will open the jupyter console in your web browser

Navigate to the pythonTutorial directory and open pythonTutorial.ipynb

2 Getting Started with python

3 Printing

In any tutorial, the obligatory first step is printing the phrase "Hello World!". The cell below makes a call to python's print function and we pass the string we want to print

[ ]:
```
print("Hello World!")
```

4 Variables and Data Types

We will now see how to create variables and explore some of the basic data types available in python.

Variables are symbols referencing data that is stored in the computer's memory.

In python, you can create a variable by either assigning the value of interest or using the data type's constructor.

4.1 Integers and Floating Point Numbers

Integers and floats are two data types that can be used to represent a number. Integers represent whole numbers and floats can represent decimal values.

Lets create an integer by assigning a value to a variable

[ ]:
```
int_1 = 4
```

Now, lets use the int constructor:

[ ]:
```
int_2 = int(5)
```

We can do the same with floating point numbers:

[ ]:
```
float_1 = 5.123456789
float_2 = float(10.987654321)
```

To check the value of a variable in a jupyter notebook, you can simply execute a cell containing the name of the variable, as seen below.

[ ]:
```
int_1
```

[ ]:
```
int_2
```

[ ]:
```
float_1, float_2
```

Additionally, you can print the values of these variables. As you can see below, any number of arguments can be passed to the print function, and the arguments will be printed as a list.

```
[ ]: print("The value of int_1 is", int_1,"and the value of float_1 is", float_1)
```

Below, we'll see how to do mathematical operations on integers and floats

```
[ ]: print(4 + 3)
```

```
[ ]: print(int_1 - 5)
```

```
[ ]: print(int_1 * float_1)
```

```
[ ]: print(float_1 / 2)
```

4.2 Boolean Values - Bool

Boolean values represent true/false values

Below are two ways to create a boolean variable

```
[ ]: first_bool = True
second_bool = bool(False)
```

We can also use boolean variables to evaluate logical statements. You can check inequalities as is done below. Notice that when checking for equality, we use "==", because "=" is already used to assign values to variables.

```
[ ]: is_bigger = 5>4
is_smaller = (6<5)
is_equal = (int_1==4)
```

```
[ ]: print(is_bigger, is_smaller, is_equal)
```

We can negate a bool using the "not" keyword

```
[ ]: print(is_bigger, not is_bigger)
```

We can chain together logical statements using and/or

```
[ ]: first_and_second = is_bigger and is_smaller
first_or_second = is_bigger or is_smaller
first_and_second_or_third = (is_bigger and is_smaller) or is_equal
```

```
[ ]: print(first_and_second, first_or_second, first_and_second_or_third)
```

4.3 Lists, Tuples

Lists are objects representing ordered sequences of values. Tuples are similar, but as we'll see soon are **unchangeable**.

List Reference: https://www.w3schools.com/python/python_ref_list.asp

We can create lists and tuples in the below ways:

```
[ ]: list_1 = [1,2,3,4,5]
     tup_1 = (6,7,8)
     list_2 = list(tup_1)
```

The items of a list don't need to be of the same type:

```
[ ]: list_3 = ["a",1,"b",2]
     list_3
```

To concatenate lists, simply add them together

```
[ ]: list_1 + list_2
```

Below we reverse a list by calling the list object's reverse function. This reverses the contents of the list and does not return any value.

```
[ ]: list_1.reverse()
     list_1
```

```
[ ]: list_1.reverse()
     list_1
```

You can index a list and get a subsequence. Python is zero-indexed, meaning the first item in a list is at position 0.

```
[ ]: list_1[3]
```

You can get a slice of a list using the "start:end" notation below. These ranges include the start value, but exclude the end value. Below we show two ways of getting the first three elements of a list.

```
[ ]: list_1[0:3]
```

```
[ ]: list_1[:3]
```

This "start:end" slice can similarly be represented using the range function:

```
[ ]: list(range(0,4))
```

range() has multiple forms: * range(end) * range(start, end) * range(start, end, step_size)

```
[ ]: list(range(10))
```

```
[ ]: list(range(3,10))
```

```
[ ]: list(range(0,10,2))
```

Notice above that 10 is not included as it is the end value. To include it we would change the above line to:

[ ]:
```
list(range(0,11,2))
```

We can also index ranges from the right using negative indices. To get everything up to the last item in the list, we'd write

[ ]:
```
list_1[:-1]
```

We can also create ranges in reverse order:

[ ]:
```
list(range(-1,-4,-1))
```

Thus, to get the last three elements of a list in reverse order, we'd write:

[ ]:
```
list_1[-1:-4:-1]
```

Lists are mutable, meaning you can change its values:

[ ]:
```
print(list_1)
list_1[2] = 55
print(list_1)
```

Tuples are not mutable. Below we try changing the second value of tup_1 to 5.

Here, we also are using a try/except block. These blocks first run the code in the try block, and if an error is raised, we jump to the except block

[ ]:
```
# try running this code block and if an error gets raised do that
try:
    print("trying to change tuple")
    tup_1[1] = 5
    print(tup_1)
except:
    print("Error: TUPLES ARE NOT MUTABLE")
```

4.4 Strings

Strings are a class, discussed later, that hold a series of characters. Internally, the sequence of characters is stored as a list, so many of the list operations discussed above can be leveraged.

Resource for python strings: https://www.w3schools.com/python/python_ref_string.asp

We can create strings in the following two ways:

[ ]:
```
first_half = "the quick brown fox"
second_half = str("jumped over the lazy dog")
```

[ ]:
```
print(first_half)
print(second_half)
```

There are a few special characters that can be used to format a string: * “\n”: new line * “\t”: tab

[ ]:
```
letter = "Hi Everyone,\n \t I hope you are enjoying this tutorial on python!"
```

[ ]:
```
print(letter)
```

To concatenate two strings, simple add their values:

[ ]:
```
full_sentence = first_half + " " + second_half
```

[ ]:
```
full_sentence
```

There are many functions in the string class. Lets see a few:

Below we will replace all matches of the first string with the contents of the second string

[ ]:
```
s = "aaaaaaaaa"
print(s.replace("a","b"))
```

We can also limit the number of replacements:

[ ]:
```
s.replace("aa","b",3)
```

We can convert a string to upper case and lower case and even title format

[ ]:
```
book = "the cat in the hat"
print(book.upper())
print(book.title())
print("DAN".lower())
```

Internally, strings are just lists of characters

[ ]:
```
first_half
```

[ ]:
```
list(first_half)
```

This means that we can index a string just as we would index a list

[ ]:
```
first_half[0]
```

[ ]:
```
first_half[4:9]
```

One useful string function is “find”. This function returns the index of the first match of the string argument passed.

```
[ ]: print(list(second_half))
     print(second_half.find("the"))
     print(second_half.find(" dog"))
     print(second_half[ second_half.find("the") + 4 : second_half.find(" dog") ])
```

4.5 Sets

Sets are unordered groups of items

```
[ ]: fruits = {"apple", "orange", "banana"}
     vegetables = set(("broccoli", "carrot", "lettuce", "olives"))
     food_I_dont_like = {"olives", "anchovy"}
```

```
[ ]: fruits
```

You can perform standard set operations

```
[ ]: fruits.union(vegetables)
```

```
[ ]: vegetables.difference(food_I_dont_like)
```

```
[ ]: fruits.update(["grape"])
     fruits
```

```
[ ]: fruits.remove("orange")
     fruits
```

4.6 Dictionaries

Dictionaries are key/value stores. They can be created in two ways:

```
[ ]: dan = {"first name": "Dan", "last name": "Zeiberg", "age": 24}
     pedja = dict([("first name","Predrag"), ("last name","Radivojac")])
```

Here we create a list containing the two dictionaries we just created

```
[ ]: people = [dan, pedja]
     people
```

4.7 Nump Arrays and Matrices

As discussed earlier, Numpy allows you to analyze multidimensional data.

Numpy Reference: https://docs.scipy.org/doc/numpy-1.17.0/

Below, we import the numpy package. To cutdown on typing as we use the package, you can give the package a name that it will be imported as. Numpy is often imported as is done below:

[ ]:
```
import numpy as np
```

Here we create a np array from a list

[ ]:
```
arr_1 = np.array([6,2,6,8,1,6,22,6,8,999])
```

[ ]:
```
arr_1
```

np arrays have an attribute .shape, giving the dimensions of the array

[ ]:
```
arr_1.shape
```

You can reshape an array as is done below

[ ]:
```
arr_2 = arr_1.reshape((2,5))
```

[ ]:
```
print(arr_2)
```

[ ]:
```
print(arr_2.shape)
```

You can perform operations on np arrays.

We can get the mean value of an entire array:

[ ]:
```
arr_2.mean()
```

We can additionally do operations along certain axis. Below, we get the sum of each column. The argument "axis=0" means perform the operation along the row axis

[ ]:
```
arr_2.sum(axis=0)
```

Similarly, we can get the median of each row by calling the np.median function and applying the operation along the column axis

[ ]:
```
np.median(arr_2,axis=1)
```

np.random provides many functions for randomly sampling values.

Below we sample 10 values from the discrete uniform distribution U(0,25)

[ ]:
```
arr_3 = np.random.randint(0,25,size=10)
arr_3
```

Here we draw 10 values from the standard normal distribution

[ ]:
```
arr_4 = np.random.normal(loc=0, scale=1, size=10)
arr_4
```

You can do arithmetic on arrays

[ ]:
```
square_1 = np.array([[1,2],[3,4]])
square_2 = np.array([[10,9],[8,7]])
```

[ ]:
```
square_2
```

[ ]:
```
square_1
```

[ ]:
```
square_2 / square_1
```

You can index and slice np arrays similarly to how you would a list. The first axis is the row, second axis is the column, and additional axis could be included after, such as a depth dimension

[ ]:
```
matrix_0 = np.random.randint(0,10,size=(5,5))
matrix_0
```

We can get the value at row 2 column 3:

[ ]:
```
matrix_0[1,2]
```

We can get the 3x3 square at the middle of this matrix:

[ ]:
```
matrix_0[1:-1, 1:-1]
```

5 Pandas DataFrames

As mentioned before, pandas allows you to create dataframes, which are similar to tables in a database.

Below we import the pandas package and name it "pd"

[ ]:
```
import pandas as pd
```

5.1 Creating DataFrames

We can create a DataFrame from a numpy array:

[ ]:
```
pd.DataFrame(matrix_0)
```

We can also create a dataframe from a dictionary:

[ ]:
```
people
```

[ ]:
```
people_df = pd.DataFrame(people)
people_df
```

We can also load a file into a dataframe

[ ]:
```
housing = pd.read_csv("BostonHousing.csv")
```

[ ]:
```
housing
```

5.2 Setting DataFrame Index

Similar to a database table, each row is uniquely identified by its index value. We can replace the current integer index in the people DataFrame with the person's last name

[ ]:
```
people_df = people_df.set_index("last name")
people_df
```

5.3 Indexing DataFrame

You can get a subset of the columns of a DataFrame as is done below:

[ ]:
```
housing[["age", "tax"]]
```

You can index a dataframe by row number

[ ]:
```
housing
```

[ ]:
```
housing.iloc[0]["age"]
```

Or you can get values by specifying the index and column values

[ ]:
```
people_df
```

Get Dan's age

[ ]:
```
people_df.loc["Zeiberg","age"]
```

You can extract an entire row from a DataFrame

[ ]:
```
people_df.loc["Zeiberg"]
```

You can also extract an entire column

[ ]:
```
people_df["first name"]
```

5.4 Convert DataFrame to Numpy Array

Often it is necessary to convert DataFrame columns to np arrays. You can do this using the .values attribute.

Below, we create an array from the housing DataFrame where the first column is "age" and the second column is "tax"

[ ]:
```
housing[["age", "tax"]].values
```

5.5 Querying DataFrame

You can query a DataFrame to get a subset of rows that are of interest.

[ ]:
```
housing[housing["age"] > 33]
```

You can construct complex queries by joining queries with and operations (&) and or operations (|)

Below, we select the rows where either the age is greater than 33 and tax is less than 350 or the "nox" value is greater than 0.5

[ ]:
```
housing[((housing["age"] > 33) & (housing["tax"] <= 350)) | (housing["nox"] > 0.
↪5)]
```

5.6 Joining DataFrames

In many cases, data from a variety of sources needs to be merged together to create one DataFrame.

Recall we previously created the people_df

[ ]:
```
people_df
```

Lets add another person to people_df

[ ]:
```
people_df.loc["Jain"] = ["Shantanu", np.nan]
```

[ ]:
```
people_df
```

Now, we create a dataframe of publications

[ ]:
```
publications_df = pd.DataFrame([
    {
        "title": "Fast Nonparametric Estimation of Class Proportions in the␣
↪Positive-Unlabeled Classification Setting",
        "year": 2020,
        "first author": "Zeiberg"},
    {
        "title": "Prediction of boundaries between intrinsically ordered and␣
↪disordered protein regions",
        "year": 2003,
        "first author": "Radivojac"},
    {
        "title": "Deep contextualized word representations",
        "year": 2018,
        "first author": "Peters"
    }])
```

[ ]:
```
publications_df
```

Now for each person in person_df, lets add that person's publication.

We do this by calling the merge function of a DataFrame. The merge function takes in several arguments: * right: the other dataframe you are merging this dataframe with * how: The type of merge we want to perform. There are a variety of merge types. Taken from the Pandas docs: * left: use only keys from left frame, similar to a SQL left outer join; preserve key order. * right: use only keys from right frame, similar to a SQL right outer join; preserve key order. * outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically. * inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys. * left_on: The column of the calling DataFrame you want to use for matching * right_on: The column of the "right" DataFrame you want to use for matching

[ ]:
```
people_df.merge(right=publications_df, how="left", left_on="last name",␣
↪right_on="first author")
```

[ ]:
```
people_df.merge(right=publications_df, how="right", left_on="last name",␣
↪right_on="first author")
```

[ ]:
```
all_info = people_df.merge(right=publications_df, how="outer", left_on="last␣
↪name", right_on="first author")
all_info
```

[ ]:
```
people_df.merge(right=publications_df, how="inner", left_on="last name",␣
↪right_on="first author")
```

5.7 Rename Columns

You can rename columns in a dataframe by passing a dictionary containing old name: new name key/value pairs

[ ]:
```
all_info.rename(columns={"first author": "last name","title":"publication name"})
```

6 Conditionals and Loops

6.1 If/Else

If/elif/else statements can direct the execution of code by checking conditions. The conditions are checked sequentially until one evaluates to true and defaults to the else block if none are true, if the else block is present.

[ ]:
```
randnum = np.random.choice([1,2,3,4])
if randnum == 1:
    print("chose first value")
elif randnum == 2:
    print("chose second value")
elif randnum == 3:
    print("chose third value")
```

```
else:
    print("choes fourth value")
```

6.2 For loops

For loops repeat a block of code a set number of times. The block of code will be executed with the control-variable assigned to each of the values in the iterable.

In this case, the control variable is "i", and the block of code will run for i=1, i=2, . . . , i=9

[ ]:
```
for i in range(10):
    print(i)
```

The iterable doesn't have to be a range. Here we iterate over the list of dictionaries, people:

[ ]:
```
for person in people:
    print(person["first name"])
```

We can nest for loops within each other. With two nested loops, for each iteration of the outer loop, the entire cycle of the inner loop is completed. This can be used to loop over matrix values.

In the below example, I use the string format function. To use format you put placeholders in your string, denoted as {}. You then call the .format function and pass as arguments the values you would like to insert into those placeholders. This allows you to format the values for printing. Here, the third placeholder {:.3f} indicates that a float will be placed here and I want to limit the float to 3 decimal places.

[ ]:
```
matrix_1 = np.random.normal(size=(5,5))
```

[ ]:
```
# Loop over each row
for r in range(matrix_1.shape[0]):
    # Loop over each column
    for c in range(matrix_1.shape[1]):
        print("The value at position ({},{}) is {:.3f}".
↪format(r,c,matrix_1[r,c]))
    print()
```

6.3 While Loops

While loops continue executing a block of code while the specified condition is true.

Below, we first create a list of numbers, then we initialize an index variable to position 0, then we increment the index until the value at the index is greater than or equal to 50.

[ ]:
```
values = [1,7,1,6,888,221,5]
idx = 0
while values[idx] < 50:
    idx += 1
```

```
print("first big number is ",values[idx])
```

7 Functions

We are not limited to the functions pre-defined in python and the packages we import. We can define our own. Below we define the function "add" that takes in two required arguments "a" and "b". Functions can optionally return a value. Here we return the sum a+b

[ ]:
```
def add(a,b):
    return a+b
```

[ ]:
```
total = add(4,5)
total
```

One can create functions that call themselves until some base case is reached. Here, we implement the factorial function. Recursive functions must include a base-case, which indicates when to stop calling the function. Here we use the property that 0! = 1 as the base-case.

Let's visualize what recursive functions are doing:

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

We have reached our base-case, so now we start climbing back up

factorial(4) = 4 * (3 * (2 * (1)))

= 4 * (3 * (2))

= 4 * (6)

= 24

[ ]:
```
def factorial(x):
    if x > 0:
        return x * factorial(x-1)
    elif x == 0:
        return 1
    else:
        raise Exception("Input must be non-negative")
```

[ ]:
```
factorial(4)
```

8 Classes

Classes are templates comprised of attributes and member functions. When you create an instance of a class, it is called an object. For example the following code:

myint = int(4)

creates an object named myint of type int.

When we directly assign a value:

other_int = 5

We are actually implicitly calling the int constructor.

The first component of a class is the constructor. All classes require a constructor function "__init__". This is the function that is called when you create a new object. Constructors can take in arguments that can be assigned to object attributes (self.attribute_name).

One can also create additional member functions for the class that can interact with the object's attributes or even other objects. Notice that every member function, including the constructor, has "self" as the first argument.

Below, we create a Person Class. The constructor takes in 3 arguments and assigns each of them to attributes. We also define the introduce() and converse() member functions.

[ ]:
```
class Person:
    def __init__(self, name, hair_color, eye_color):
        self.name = name
        self.hair_color = hair_color
        self.eye_color = eye_color

    def introduce(self):
        return "{} says: Hi, my name is {}".format(self.name, self.name)

    def converse(self, other_person):
        return "{} says: Hi {}, how are you today?".format(self.name,␣
 ↪other_person.name)

    def describe(self):
        return "{}. I have {} hair and {} eyes".format(self.introduce(), self.
 ↪hair_color, self.eye_color)
```

[ ]:
```
dan = Person("Dan", "brown", "green")
emily = Person("Emily", "blond", "blue")
print(dan.introduce())
print(emily.converse(dan))
```

You can directly change the value of attributes

[ ]:
```
dan.hair_color = "red"
dan.describe()
```

9 Plotting

Matplotlib is a package that allows you to visualize data. Matplotlib encapsulates each visualization in a figure. A figure contains axis which you create your visualization on.

The below line is a magic notebook-specific command that allows for interaction with visualizations. This will need to be removed if running as a python script.

[ ]:
```
# iPython magic to allow interactive plots
%matplotlib notebook
```

We can now import matplotlib's pyplot module. This is commonly imported as plt

[ ]:
```
# Plotting library
import matplotlib.pyplot as plt
```

9.1 Basic Line Plot

Lets read in a text file containing data that we would like to plot

[ ]:
```
ph = pd.read_csv("pH-example.txt")
```

[ ]:
```
ph
```

Now, we make a call to plt.subplots(), which returns a figure containing a single axis.

[ ]:
```
line_fig,line_ax = plt.subplots()
```

We can now draw a line plot on the line_ax axis. the plot function draws a line plot, plotting the second argument as a function of the first argument.

[ ]:
```
line_ax.plot(ph["time"], ph["v"])
```

Lets now add axis labels and a title to the plot

[ ]:
```
line_ax.set_xlabel("Time, h")
line_ax.set_ylabel("pH")
line_ax.set_title("pH over time")
```

9.2 Subplots

Lets repeat the above example, but now have two subplots, the pH as a function of time on the top, and the change in pH as a function of time on the bottom.

Because both plots are plotted with respect to time, we'll add the sharex=True argument so there is only one x axis shared by both subplots

[ ]:
```
subplot_fig,subplot_axes = plt.subplots(2,1,sharex=True)
```

Lets adjust the vertical spacing between the subplots

[ ]:
```
subplot_fig.subplots_adjust(hspace=0.3)
```

Lets add the pH plot on top

[ ]:
```
subplot_axes[0].plot(ph["time"], ph["v"], label="pH")
# subplot_axes[0].set_xlabel("Time, h")
subplot_axes[0].set_ylabel("pH")
subplot_axes[0].set_title("pH over time")
```

Now lets calculate the change in pH at each step

[ ]:
```
delta_pH = ph.values[1:,1] - ph.values[:-1,1]
```

Lets plot the change in pH over time in red, giving it the label "Δ pH" and also add labels to the axes

[ ]:
```
subplot_axes[1].plot(ph.values[1:,0],delta_pH, color="red", label="$\Delta$ pH")
subplot_axes[1].set_ylabel("$\Delta$ pH")
subplot_axes[1].set_xlabel("Time, h")
```

Lets add a legend to the figure

[ ]:
```
subplot_fig.legend(loc="center right")
```

Lets save the above figure as an .eps file

[ ]:
```
subplot_fig.savefig("ph_Plot.eps", format="eps")
```

To save memory, close the figure when you are done with it

[ ]:
```
plt.close(subplot_fig)
```

9.3 Histogram

Below is an example of creating a histogram. We'll plot the distribution of ages in the Boston Housing dataset from earlier

[ ]:
```
housing
```

We create a new figure and axis

Then we create a histogram with 50 bins with color green

Finally we add axis labels and a title

[ ]:
```
hist_fig,hist_ax = plt.subplots()
hist_ax.hist(housing["age"],bins=50, color="green")
hist_ax.set_xlabel("Age, years")
hist_ax.set_ylabel("Count")
hist_ax.set_title("Distribution of Ages in Boston Housing Dataset")
```

9.4 3D-Plot

Below we go through an example of creating a 3D surface plot visualizing the pdf of a multivariate normal distribution

We first import two additional tools, one necessary for 3d plotting and another for using a colormap

[ ]:
```
# Import for 3D plots
from mpl_toolkits.mplot3d import Axes3D
# Colormap Library
from matplotlib import cm
```

We now import the Multivariate Normal Distribution class from scipy.stats so we can plot the pdf in our surface plot

[ ]:
```
from scipy.stats import multivariate_normal as mvn
```

Now we create and fill the matrix containing our pdf values. We first create a 200x200 numpy array of 0s. Next we go through each cell of the array and calculate the pdf at that cell.

[ ]:
```
normal_mean = [100,100]
normal_covariance = [[2000,0],[0,2000]]
grid = np.zeros((200,200))
for i in range(grid.shape[0]):
    for j in range(grid.shape[1]):
        grid[i,j] = mvn.pdf([i,j],normal_mean, normal_covariance)
```

[ ]:
```
fig_3d = plt.figure()
ax_3d = fig_3d.gca(projection='3d')
X,Y = np.meshgrid(list(range(grid.shape[0])), list(range(grid.shape[1])))
surface = ax_3d.plot_surface(X,Y,grid, cmap=cm.coolwarm)
```

10 Machine Learning Pipeline Example - Predicting home Price in Boston Housing dataset using an ensemble of regression trees

We will now go through the steps of creating and evaluating a machine learning model for predicting median home prices for towns in the Boston Housing dataset. This dataset contains 13 features: * CRIM - per capita crime rate by town * ZN - proportion of residential land zoned for lots over 25,000 sq.ft. * INDUS - proportion of non-retail business acres per town. * CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise) * NOX - nitric oxides concentration (parts per 10 million) * RM - average number of rooms per dwelling * AGE - proportion of owner-occupied units built prior to 1940 * DIS - weighted distances to five Boston employment centres * RAD - index of accessibility to radial highways * TAX - full-value property-tax rate per $10,000 * PTRATIO - pupil-teacher ratio by town * B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town * LSTAT - Percentage lower status of the population

Target Variable: * MEDV - Median value of owner-occupied homes in $1000's

We'll train an ensemble of regression trees for the task of predicting median home value. The goal in supervised machine learning is to use labeled data to learn a model that minimizes a loss function. Because the median home value is a continuous target variable, this is a regression task and not a classification task. In this example we'll be minimizing the mean squared error (MSE) between the true median house price and our model's predicted median house price. MSE can be expressed as:

$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Where y_i is the true median housing price for example i and $\hat{y}_i$ is our models predicted house price for example i.

[ ]:
```
housing
```

Sklearn provides many features that are useful for developing machine learning pipelines. Below we import the package

[ ]:
```
import sklearn
```

10.1 Split Train/Test Datasets

The first step is to separate our data into training and test sets. The training set is the data that we use to learn a model. The test set is a held-out set of instances from the same distribution which we'll use to estimate the ability of our model to generalize to unseen instances. It is critical to use a held-out test set to estimate generalizability, otherwise our model might memorize the entire dataset and we'll have over-optimistic estimates of our model's performance.

Lets convert our DataFrame into two numpy arrays containing our features and labels.

[ ]:
```
features = housing[set(housing.columns) - set(["medv"])].values
labels = housing["medv"].values
```

Now, we will randomly break our dataset up into training and test sets. We'll use 80% of our data for training and the remaining 20% for testing. We'll shuffle our data before splitting to account for any trends in the ordering of the rows.

Lets import a function that can conveniently split our data

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

[ ]:
```
features_train, features_test, labels_train, labels_test =␣
↪train_test_split(features, labels, test_size=0.2, shuffle=True)
```

10.2 Regression Trees

Regression trees are a type of machine learning model that attempts to predict a continuous target variable by iteratively splitting the training dataset into groups such that each group of examples have similar target values. Below we visualize an example of a simple regression tree trained on the above training set.

We first import the regression tree class "DecisionTreeRegressor"

Next we instantiate a regression tree, setting the max depth of the tree to 3 and setting the criterion we wish to minimize as the MSE

We then train our tree on the training set

After training, we import a function to visualize our tree, create a figure to plot the tree on, then plot our tree

[ ]:
```
from sklearn.tree import DecisionTreeRegressor
regression_tree = DecisionTreeRegressor(max_depth=3, criterion="mse")
regression_tree.fit(features_train, labels_train)
from sklearn import tree
plt.figure(figsize=(15,6))
_ = tree.plot_tree(regression_tree, feature_names=housing.columns[:-1])
```

In each node, we see the rule that is used to split the data, the MSE for the nodes at this level, the number of samples, and the predicted value at this node. Final predictions are made at the bottom layer, referred to as leaves.

10.3 Ensemble Models

One common issue in regression trees is that there is a high degree of variance. To reduce the variance of our model, many trees can be learned and the final prediction for an instance is the average prediction across all the trees. This idea of using many models is referred to as an ensemble. In this experiment, we'll be using an ensemble of 100 regression trees.

Below, we import the class for creating an ensemble of regression trees

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

10.4 K-Fold Cross Validation

Many machine learning models have parameters that need to be set by hand and cannot be directly learned from the data. It is up to you to find the best setting of these parameters. To choose these parameters, we want to train a model and estimate how that trained model would perform on unseen data, choosing the set of parameters that results in the model that best generalizes. We cannot use the test data for tuning these hyper-parameters, as this data would effectively no longer be unseen, so we need to hold out data from our training set. K-Fold cross validation is one strategy that can be used to validate the choices of hyperparameters.

In this process, we randomly separate our training data into k equal sized sets, called folds; k is often set to 5. We then choose 4 folds to use as the training set and after training the model on these 4 folds, we evaluate the model on the last held out fold. We repeat this so each fold is held out once, then average the performance on each held out fold. This gives an estimate of how the current hyperparameter choice generalizes to unseen data. We can repeat this process for each set of hyperparameters and choose the set that results in the best performance.

To illustrate this concept, lets decide how to optimally set the max depth of our decision trees. We first import the cross_val_score method that allows us to perform k-fold cross validation.

[ ]:
```
from sklearn.model_selection import cross_val_score
```

This function takes in a fitted model, a feature matrix and a label vector and calculates the MSE. This will be used by the cross_val_score method above.

[ ]:
```
def getMSE(regressor, X, y):
    predictions = regressor.predict(X)
    return np.mean((y - predictions)**2)
```

We now perform 5-fold cross validation for each max depth setting in the range [3,20)

After, we'll plot the validation MSE at each of these settings.

[ ]:
```
val_scores = []
for md in range(3,20):
    regressor = RandomForestRegressor(criterion="mse",max_depth=md,␣
 ↪n_estimators=100)
    val_scores.append(np.mean(cross_val_score(regressor, features_train,␣
 ↪labels_train, cv=5, scoring=getMSE)))
plt.figure()
plt.plot(range(3,20),val_scores)
plt.xlabel("Max Depth")
plt.ylabel("MSE")
```

In the above figure, you should see that the MSE decreases rapidly at first, then after a while the MSE seems to level off, resembling an elbow. This elbow is the point at which there is diminishing benefit for adding complexity to the model. We will choose to set the max depth as the value at which this elbo occurs. When I ran this, the elbow was around 8.

Using the setting we found above, we will retrain our final model on the entire dataset. Below I instantiate a new model with max depth 8 and train it on all the training data.

[ ]:
```
final_model = RandomForestRegressor(criterion="mse", max_depth=8,␣
 ↪n_estimators=100)
final_model.fit(features_train, labels_train)
```

Using this trained model, we can evaluate its final performance on held out data. We will make predictions on our test dataset then calculate the test mean squared error.

[ ]:
```
test_predictions = final_model.predict(features_test)
test_mse = np.mean((labels_test - test_predictions)**2)
print("Test Mean Squared Error is : {:.4f}".format(test_mse))
```

[ ]:
```

```