

MACHINE LEARNING

(HOUSE SALE PRICES PREDICTION USING LINEAR REGRESSION)

*Summer Internship Report Submitted in partial fulfillment
of the requirement for undergraduate degree of*

Bachelor of Technology

In

Electronics and Communication Engineering

By

KARTHIK M.B

2210416132

Under the Guidance of

Mr. K. Sathish, M.Tech

Assistant Professor



Department Of Electronics and Communication Engineering

GITAM School of Technology

GITAM (Deemed to be University)

Hyderabad-502329

May-June 2019

DECLARATION

I submit this industrial training work entitled "**PREDICTING THE HOUSE SALE PRICES USING LINEAR REGRESSION**" to GITAM (Deemed to be University), Hyderabad in partial fulfillment of the requirements for the award of the degree of "**Bachelor of Technology**" in "**Electronics and Communication Engineering**". I declare that it was carried out independently by me under the guidance of **Mr. K. Sathish**, Asst. Professor, GITAM (Deemed to be University), Hyderabad, India.

The results embodied in this report has not been submitted to any other University or Institute for the award of any degree or diploma.

Place: **HYDERABAD**

KARTHIK M.B

Date: 20.07.2019

2210416132



GITAM UNIVERSITY

Hyderabad-502329, India

Dated: 25 July 2019

CERTIFICATE

This is to certify that the Industrial Training Report entitled “**Predicting the House Sale Prices using Linear Regression.**” is being submitted by **Karthik MB** (2210416132) in partial fulfillment of the requirement for the award of **Bachelor of Technology in Electronics & Communication Engineering** at GITAM (Deemed to be University), Hyderabad.

It is faithful record work carried out by her at the **Electronics & Communication Engineering Department**, GITAM (Deemed to be University) Hyderabad Campus under my guidance and supervision.

Mr. K. Sathish

Assistant Professor

Department of ECE

Dr. K. Manjunathachari


Professor and HOD

Department of ECE


Date: 16th June 2019

CERTIFICATE

This is to certify that the Internship titled “Based on the given dataset predict the House Sale Prices using Linear Regression” is the bona fide work carried out by **KARTHIK M B** student of the **GITAM University, Hyderabad**, in partial fulfillment for the award of **Bachelor of Technology** in Electronics and Communication Engineering during the period 29th April 2019 – 15th June 2019 at **PROMIZE IT SERVICES PRIVATE LIMITED – HYDERABAD**. During this period his conduct was found to be very good and he has shown good technical skills.


(Dinesh Kumar Jooshetti)
Project Head
Promize IT Services Pvt Ltd.




(Suresh Shankar)
Director of Operations
Promize IT Services Pvt Ltd.

ACKNOWLEDGMENT

I wish to take this opportunity to express my deep gratitude to all those who helped, encouraged, motivated and have extended their cooperation in various ways during my training program.

I would like to thank respected **Dr. N. Siva Prasad**, Pro Vice Chancellor, GITAM Hyderabad and **Dr. Ch. Sanjay, Principal**, GITAM Hyderabad.

I would like to thank **DR. K. Manjunathachari**, Head of the department of ECE for giving me such a wonderful opportunity to expand my knowledge for my own branch and giving me guidelines to present a seminar report. It helped me a lot to realize of what we study for.

I would like to thank to respected faculties **Mr. K. Sathish** who helped me to make this seminar a successful accomplishment.

I would like to thank my friends who helped me to make my work more organized and well-stacked till the end.

KARTHIK M.B

2210416132

ABSTRACT

Data science refers to the process of extraction of useful insights from data. This interdisciplinary approach merges various fields of computer science, scientific processes and methods, and statistics in order to extract data in automated ways. In order to mine big data, which is closely associated with the field, data science uses a diverse range of techniques, tools and algorithms gleaned from the fields.

In machine learning (ML), statistical methods are used to empower machines to learn without being programmed explicitly. The field focuses on letting algorithms learn from the provided data, collect insights, and make predictions on unanalyzed data based on the gathered information. In general, ML is based on three key models of learning algorithms:

- Supervised machine learning algorithms
- Unsupervised machine learning algorithms
- Reinforcement machine learning algorithms

In the first model, a dataset is present with inputs and known outputs. In the second one, the machine learns from a dataset that comes with input variables only. In reinforcement learning model, algorithms are used to select an action. This project is implemented using *supervised machine learning algorithms*. The outcome of our project is to make predictions on the sales prices of the houses of California State with the dataset provided.

It is hoped this study will inform better analyzation of gathered information (unanalyzed data) and other machine learning techniques.

Table of Contents

1	Introduction	1
1.1	What Is Data Science?.....	1
2	Installation	3
2.1	Software's Description	3
3	Python introduction	4
3.1	What is python ?.....	4
3.2	What can Python do?.....	4
3.3	Why Python?	4
4	Introduction to NumPy	5
4.1	NumPy Array Attributes	5
4.2	Fixed-Type Arrays in Python	6
4.3	Creating Arrays from Python Lists	7
4.4	Creating Arrays from Scratch	8
4.5	NumPy Standard Data Types	9
4.6	The Basics of NumPy Arrays.....	10
4.7	Array Indexing: Accessing Single Elements.....	10
4.8	Array Slicing: Accessing Subarrays.....	10
4.9	Reshaping of Arrays.....	11
4.10	Aggregations: Min, Max	12
5	Data Manipulation with Pandas.....	13
5.1	Installing and Using Pandas	13
5.2	The Pandas Series Object.....	13
5.3	Lists of Python operators and their equivalent Pandas object methods	15
5.4	Handling Missing Data.....	15

5.5	Lists of upcasting conventions in Pandas when NA values are introduced.	16
5.6	Operating on Null Values.....	17
6	Visualization with Matplotlib.....	18
6.1	Importing matplotlib	18
6.2	Basic Errorbars	18
6.3	Scatter Plot	19
6.4	Line Chart.....	20
6.5	Histogram	20
6.6	Bar Chart	21
7	Machine Learning.....	22
7.1	Some machine learning methods:	22
7.2	Importance of Machine learning	23
7.3	Uses of Machine Learning	24
7.4	Introducing Scikit-Learn	24
7.5	Data Representation in Scikit-Learn	24
8	Predicting The House Sales Prices Using Linear Regression	27
8.1	Problem Statement	27
8.2	Firstly ,what is the problem?	27
8.3	Mission 1: Importing Libraries	27
8.4	Mission 2: Introduction to the Data.....	28
8.5	Mission 3: Simple Linear Regression	29
8.6	Mission 4: Least Squares.....	32
8.7	Mission 5: Using Scikit-Learn to Train and Predict	33
8.8	Mission 6: Making Predictions	34
9	References:	36

List Of Figures

Figure 1: Drew Conway's Data Science Venn Diagram	1
Figure 2: Arrays of a uniform type	7
Figure 3: Importing Numpy	7
Figure 4: Integers up-casted to floating point	7
Figure 5 Initializing a multidimensional array.....	7
Figure 6: Examples for creating arrays.....	8
Figure 7: An errorbar example.....	19
Figure 8: Matplotlib Scatter plot.....	19
Figure 9: Scatter Plot colored by class.....	20
Figure 10: Line Chart.....	20
Figure 11: Histogram	21
Figure 12: Bar-Chart	21
Figure 13: Machine learning process flow.....	23
Figure 14: A visualization of the Iris dataset	26
Figure 15: Scikit-Learn's data layout	26
Figure 16: Importing Libraries	27
Figure 17: Ames Housing Dataset	28
Figure 18: Displaying information about each column.	29
Figure 19: Different simple linear regression models.....	30

Figure 20: Correlation between various parameters with 'SalePrice'	31
Figure 21: Correlation Table.....	32
Figure 22: Errorbar graph	32
Figure 23: RSS on training set	33
Figure 24: Using Scikit-Learn to Train and Predict.....	34
Figure 25: simple linear regression fit to the data.....	35

List of Tables

Table 1: List of Data types in NumPy	9
Table 2: Aggregation functions available in NumPy.....	12
Table 3: Mapping between Python operators and Pandas methodserator.....	15
Table 4: Pandas handling of NAs by type	16

1 INTRODUCTION

1.1 What Is Data Science?

The project is done using data science with Python, which immediately begs the question: what is *data science*? It's a surprisingly hard definition to nail down, especially given how ubiquitous the term has become. Vocal critics have variously dismissed the term as a superfluous label (after all, what science doesn't involve data?) or a simple buzzword that only exists to salt resumes and catch the eye of overzealous tech recruiters.

In my mind, these critiques miss something important. Data science, despite its hype-laden veneer, is perhaps the best label we have for the cross-disciplinary set of skills that are becoming increasingly important in many applications across industry and academia. This cross-disciplinary piece is key: in my mind, the best existing definition of data science is illustrated by Drew Conway's Data Science Venn Diagram.

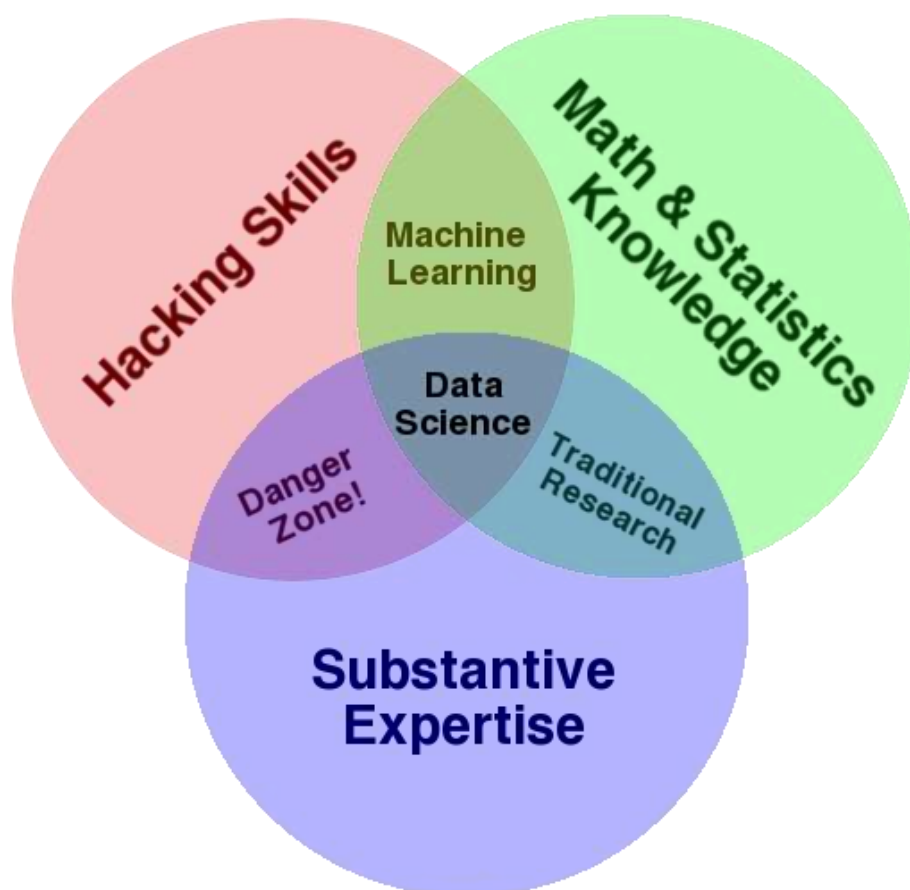


Figure 1: Drew Conway's Data Science Venn Diagram

While some of the intersection labels are a bit tongue-in-cheek, this diagram captures the essence of what I think people mean when they say "data science": it is fundamentally an *interdisciplinary* subject. Data science comprises three distinct and overlapping areas: the skills of a *statistician* who knows how to model and summarize datasets (which are growing ever larger); the skills of a *computer scientist* who can design and use algorithms to efficiently store, process, and visualize this data; and the *domain expertise*—what we might think of as "classical" training in a subject—necessary both to formulate the right questions and to put their answers in context.

Fields where data science can be applied are when you are reporting election results, forecasting stock returns, optimizing online ad clicks, identifying microorganisms in microscope photos, seeking new classes of astronomical objects, or working with data in any other field.

2 INSTALLATION

Before we start with the project let's begin by installing the required software's. Here I'm using Python platform to build my machine learning model. One can easily download Python software directly from the website python.org or there are many ways to start with. Here I'm installing python through Anaconda Distribution or you can follow the below steps:

1. Open anaconda.org and click download the latest version based on your pc
2. After downloading, follow the installation steps and conda will be installed
3. Now open anaconda icon and launch Jupyter notebooks

Your python is indirectly downloaded through the anaconda. To start coding into Python, launch anaconda and navigate to Jupyter Notebooks. Here create a root folder and name it as your like (I've named it as Internship), do save all your files, codes, datasets, images regarding your project here.

2.1 Software's Description

Anaconda is a free and open source distribution of Python and R programming languages for scientific computing (data science, machine learning, applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment.

Jupyter Notebook is a web based interactive computational environment for creating Jupyter notebooks documents. The 'notebook' term colloquially make reference to many different entities, mainly the Jupyter web application, Jupyter Python web server, or jupyter document format depending on context. A Jupyter Notebook document is a **JSON** document, following a versionised schema, and containing an ordered list of input/output cells which can contain code, text (using Markdown), mathematics, plots and rich media, usually ending with the ".ipynb" extension.

For the documentation of anaconda and jupyter you can visit the official website of [anaconda](https://anaconda.org) and [jupyter](https://jupyter.org).

3 PYTHON INTRODUCTION

3.1 What is python ?

Python is a popular programming language. It was created by **Guido van Rossum** and released in 1991. It is used for:

- 1) Web development (server-side),
- 2) Software development,
- 3) Mathematics,
- 4) System scripting...

3.2 What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

3.3 Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax like the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Good to know

The most recent major version of Python is Python 3, which we shall be using in this study. In this project Python will be written in a Jupyter Notebooks. It is possible to write Python in an Integrated Development Environment (IDE), such as Colab, Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing large collections of Python files.

4 INTRODUCTION TO NUMPY

This chapter, outlines few techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images—particularly digital images—can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making it analyzable will be to transform them into arrays of numbers.

For this reason, efficient storage and manipulation of numerical arrays is fundamental to the process of doing data science. We'll now look at the specialized tools that Python has for handling such numerical arrays: the NumPy package, and the Pandas package (discussed in Chapter 5).

NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in “list” type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the Preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to <http://www.numpy.org/> and follow the installation instructions found there.

4.1 NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays: a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:


```

In[1]:import numpy as np
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array

```

Each array has attributes *ndim* (the number of dimensions), *shape* (the size of each dimension), and *size* (the total size of the array):

```

In[2]:    print("x3 ndim: ", x3.ndim)
         print("x3 shape:", x3.shape)
         print("x3 size: ", x3.size)
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60

In[3]:    print("dtype:", x3.dtype)
dtype: int64

```

Other attributes include *itemsize*, which lists the size (in bytes) of each array element, and *nbytes*, which lists the total size (in bytes) of the array:

```

In[4]:    print("itemsize:", x3.itemsize, "bytes")
         print("nbytes:", x3.nbytes, "bytes")
itemsize: 8 bytes
nbytes: 480 bytes

```

4.2 Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in *array* module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
```

```
Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Figure 2: Arrays of a uniform type

Here 'i' is a type code indicating the contents are integers.

Here I'm importing standard NumPy, under the alias np:

```
In [7]: import numpy as np
```

Figure 3: Importing Numpy

4.3 Creating Arrays from Python Lists

We can use np.array to create arrays from Python lists. Unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
In [9]: np.array([3.14, 4, 2, 3])
```

```
Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

Figure 4: Integers up-casted to floating point

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In [11]: # nested lists result in multi-dimensional arrays
        np.array([range(1, i + 3) for i in [2, 4, 6]])
```

```
Out[11]: array([[2, 3, 4],
               [4, 5, 6],
               [6, 7, 8]])
```

Figure 5 Initializing a multidimensional array

4.4 Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In [13]: # Create a 3x5 floating-point array filled with ones
         np.ones((3, 5), dtype=float)
```

```
Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])
```

```
In [14]: # Create a 3x5 array filled with 3.14
         np.full((3, 5), 3.14)
```

```
Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```
In [15]: # Create an array filled with a linear sequence
         # Starting at 0, ending at 20, stepping by 2
         # (this is similar to the built-in range() function)
         np.arange(0, 20, 2)
```

```
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Figure 6: Examples for creating arrays

4.5 NumPy Standard Data Types

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (−128 to 127)
int16	Integer (−32768 to 32767)
int32	Integer (−2147483648 to 2147483647)
int64	Integer (−9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

Table 1: List of Data types in NumPy

4.6 The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 5) are built around the NumPy array.

Categories of basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

4.7 Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, you can access the i_{th} value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
In[6]: x1[0]
Out[6]: 5
```

4.8 Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array *x*, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=size of dimension`, `step=1`.

4.8.1 One-dimensional subarrays

```
In[16]: x = np.arange(10)
```

```

x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[17]: x[:5] # first five elements
Out[17]: array([0, 1, 2, 3, 4])

```

4.8.2 Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas. For example:

```

In[24]: x2
Out[24]: array([[12, 5, 2, 4],
 [ 7, 6, 8, 8],
 [ 1, 6, 7, 7]])
In[25]: x2[:2, :3] # two rows, three columns
Out[25]: array([[12, 5, 2],
 [ 7, 6, 8]])

```

4.9 Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```

In[38]: grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
In[39]: x = np.array([1, 2, 3])
        # row vector via reshape
        x.reshape((1, 3))
Out[39]: array([[1, 2, 3]])
In[40]: # row vector via newaxis

```

```

x[np.newaxis, :]
Out[40]: array([[1, 2, 3]])
In[41]: # column vector via reshape
        reshape((3, 1))
Out[41]: array([[1],
                [2],
                [3]])

```

4.10 Aggregations: Min, Max

Often when you are faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the “typical” values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

Function name	Nan version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

Table 2: Aggregation functions available in NumPy

5 DATA MANIPULATION WITH PANDAS

Pandas is a newer package built on top of NumPy and provides an efficient implementation of a DataFrame. DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements several powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us. Pandas and its Series and DataFrame objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time.

5.1 Installing and Using Pandas

Installation of Pandas on your system requires NumPy to be installed. Details on this installation can be found in the Pandas documentation. If you followed the advice outlined in the Installation chapter and used the Anaconda stack, you already have Pandas installed.

5.2 The Pandas Series Object

A Pandas *Series* is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
Out[2]: 0 0.25
```


5.2.1 Series as generalized NumPy array

From what we've seen so far, it may look like the *Series* object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the NumPy array has an *implicitly defined* integer index used to access the values, the Pandas *Series* has an *explicitly defined* index associated with the values. This explicit index definition gives the *Series* object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
data
Out[7]: a 0.25
b 0.50
c 0.75
d 1.00
dtype: float64
```

5.2.2 Series as specialized dictionary

You can think of a Pandas *Series* a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a *Series* is a structure that maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas *Series* makes it much more efficient than Python dictionaries for certain operations.

We can make the Series-as-dictionary analogy even more clear by constructing a *Series* object directly from a Python dictionary:

```
In[11]: population_dict = {'California': 38332521,
'Texas': 26448193,
'New York': 19651127,
'Florida': 19552860,
'Illinois': 12882135}
```

```

population = pd.Series(population_dict)
population
Out[11]:  California 38332521
         Florida 19552860
         Illinois 12882135
         New York 19651127
         Texas 26448193
dtype: int64

```

5.3 Lists of Python operators and their equivalent Pandas object methods

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Table 3: Mapping between Python operators and Pandas methods

5.4 Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. Many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will at some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here we'll refer to missing data in general as *null*, *NaN*, or *NA* values.

5.4.1 **None**: Pythonic missing data

The first sentinel value used by Pandas is ***None***, a Python singleton object that is often used for missing data in Python code. Because *None* is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type '*object*' (i.e., arrays of Python objects):

```
In[1]: import numpy as np
import pandas as pd
In[2]: vals1 = np.array([1, None, 3, 4])
vals1
Out[2]: array([1, None, 3, 4], dtype=object)
```

5.4.2 **NaN**: Missing numerical data

The other missing data representation, *NaN* (acronym for Not a Number), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
Out[5]: dtype('float64')
```

5.5 Lists of upcasting conventions in Pandas when NA values are introduced.

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	<code>np.nan</code>
object	No change	<code>None</code> or <code>np.nan</code>
integer	Cast to float64	<code>np.nan</code>
boolean	Cast to object	<code>None</code> or <code>np.nan</code>

Table 4: Pandas handling of NAs by type

5.6 Operating on Null Values

As we have seen, Pandas treats *None* and *NaN* as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()` :
Generate a Boolean mask indicating missing values
- `notnull()`
Opposite of `isnull()`
- `dropna()`
Return a filtered version of the data
- `fillna()`
Return a copy of the data with missing values filled or imputed

6 VISUALIZATION WITH MATPLOTLIB

We'll now look at the Matplotlib tool for visualization in Python. Matplotlib is a multiplatform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this as a cue to set out on his own, and the Matplotlib package was born, with version 0.1 released in 2003.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. For this reason, I believe that Matplotlib itself will remain a vital piece of the data visualization stack, even if new tools mean the community gradually moves away from using the Matplotlib API directly.

6.1 Importing matplotlib

Just as we use the **np** shorthand for NumPy and the **pd** shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

```
In[1]: import matplotlib as mpl
import matplotlib.pyplot as plt
```

6.2 Basic Errorbars

A basic errorbar can be created with a single Matplotlib function call :

```
In[1]: %matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
In[2]: x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k');
```

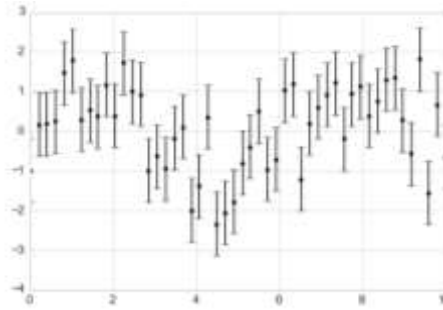


Figure 7: An errorbar example

6.3 Scatter Plot

To create a scatter plot in Matplotlib we can use the scatter method. We will also create a figure and an axis using `plt.subplots` so we can give our plot a title and labels.

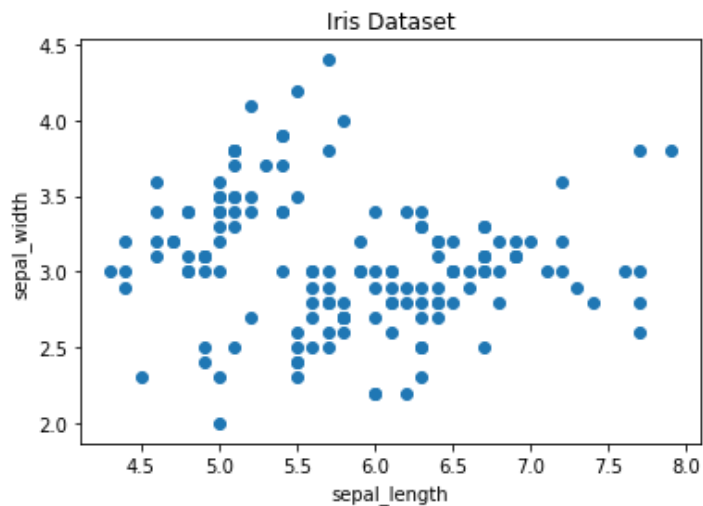


Figure 8: Matplotlib Scatter plot

We can give the graph more meaning by coloring in each data-point by its class. This can be done by creating a dictionary which maps from class to color and then scattering each point on its own using a for-loop and passing the respective color.

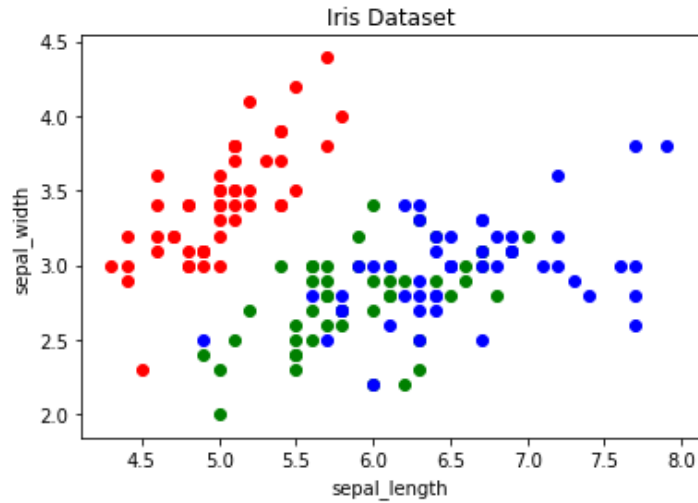


Figure 9: Scatter Plot colored by class

6.4 Line Chart

In Matplotlib we can create a line chart by calling the plot method. We can also plot multiple columns in one graph, by looping through the columns we want and plotting each column on the same axis.

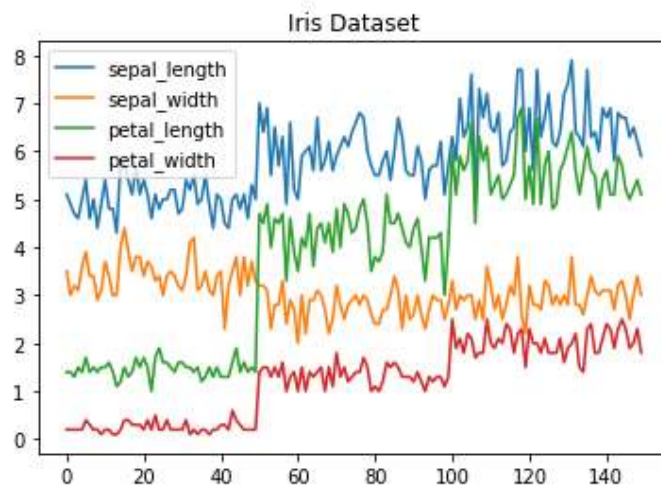


Figure 10: Line Chart

6.5 Histogram

In Matplotlib we can create a Histogram using the hist method. If we pass it categorical data like the points column from the wine-review dataset it will automatically calculate how often each class occurs.

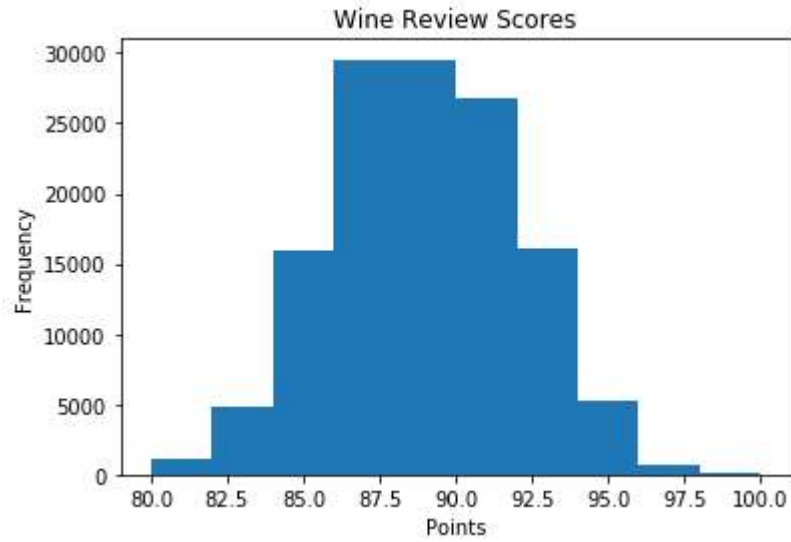


Figure 11: Histogram

6.6 Bar Chart

A bar chart can be created using the bar method. The bar-chart isn't automatically calculating the frequency of a category so we are going to use pandas value_counts function to do this. The bar-chart is useful for categorical data that doesn't have a lot of different categories (less than 30) because else it can get quite messy.

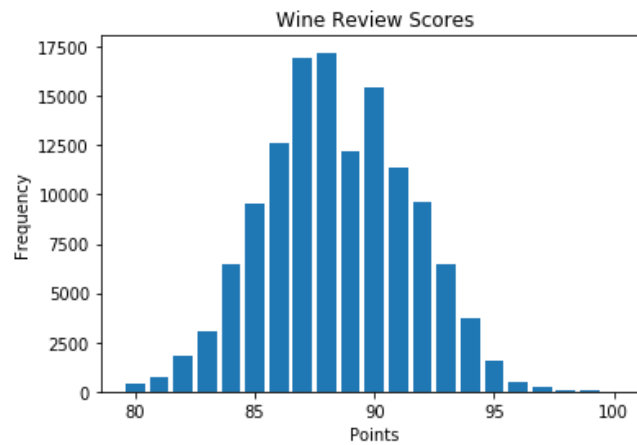


Figure 12: Bar-Chart

7 MACHINE LEARNING

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. *Machine learning* focuses on the development of computer programs that can access data and use it to learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

7.1 Some machine learning methods:

Machine learning algorithms are often categorized as supervised or unsupervised.

7.1.1 Supervised machine learning

Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly.

7.1.2 Unsupervised machine learning

These are used when the information used to train is neither classified nor labeled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data.

7.1.3 Semi-supervised machine learning

These algorithms fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data

and a large amount of unlabeled data. Usually, semi-supervised learning is chosen when the acquired labeled data requires skilled and relevant resources in order to train it / learn from it.

7.1.4 Reinforcement machine learning

Algorithms is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance.

7.2 Importance of Machine learning

Consider some of the instances where machine learning is applied: the self-driving Google car, cyber fraud detection, online recommendation engines like friend suggestions on Facebook, Netflix showcasing the movies and shows you might like, and “more items to consider” and “get yourself a little something” on Amazon are all examples of applied machine learning. All these examples echo the vital role machine learning has begun to take in today’s data-rich world.

Machines can aid in filtering useful pieces of information that help in major advancements, and we are already seeing how this technology is being implemented in a wide variety of industries. The process flow depicted here represents how machine learning works

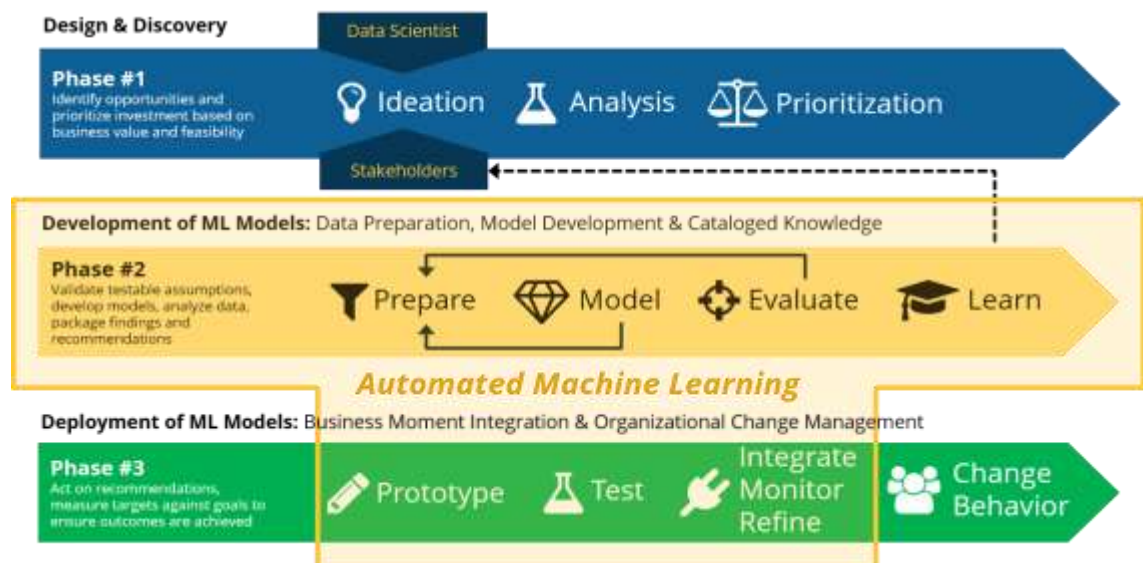


Figure 13: Machine learning process flow

7.3 Uses of Machine Learning

Earlier, we mentioned some applications of machine learning. To understand the concept of machine learning better, let's consider some more examples: web search results, real-time ads on web pages and mobile devices, email spam filtering, network intrusion detection, and pattern and image recognition. All these are by-products of applying machine learning to analyze huge volumes of data.

By developing fast and efficient algorithms and data-driven models for real-time processing of data, machine learning can produce accurate results and analysis.

7.4 Introducing Scikit-Learn

There are several Python libraries that provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn, a package that provides efficient versions of a large number of common algorithms. Scikit-Learn is characterized by a clean, uniform, and streamlined API, as well as by very useful and complete online documentation. A benefit of this uniformity is that once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward.

7.5 Data Representation in Scikit-Learn

Machine learning is about creating models from data: for that reason, we'll start by discussing how data can be represented in order to be understood by the computer. The best way to think about data within Scikit-Learn is in terms of tables of data. For example, consider the Iris dataset, famously analyzed by Ronald Fisher in 1936. We can download this dataset in the form of a Pandas DataFrame using the Seaborn library:

```
In[1]: import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
Out[1]: sepal_length sepal_width petal_length petal_width
species
0 5.1 3.5 1.4 0.2 setosa
1 4.9 3.0 1.4 0.2 setosa
2 4.7 3.2 1.3 0.2 setosa
```

```
3 4.6 3.1 1.5 0.2 setosa
4 5.0 3.6 1.4 0.2 setosa
```

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as `n_samples`. Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as *features*, and the number of columns as `n_features`.

7.5.1 Features matrix

This table layout makes clear that the information can be thought of as a two dimensional numerical array or matrix, which we will call the *features matrix*. By convention, this features matrix is often stored in a variable named `x`. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`.

The samples (i.e., rows) always refer to the individual objects described by the dataset. For example, the sample might be a flower, a person, an astronomical object, or anything else you can describe with a set of quantitative measurements. The features (i.e., columns) always refer to the distinct observations that describe each sample in a quantitative manner. Features are generally real-valued, but may be Boolean or discrete-valued in some cases.

7.5.2 Target array

In addition to the feature matrix `x`, we also generally work with a *label* or *target* array, which by convention we will usually call `y`. The target array is usually one dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas *Series*. The target array may have continuous numerical values, or discrete classes/labels.

The distinguishing feature of the target array is that it is usually the quantity we want to *predict from the data*: in statistical terms, it is the dependent variable. With this target array in mind, we can use to conveniently visualize the data :

```
In[2]: %matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```

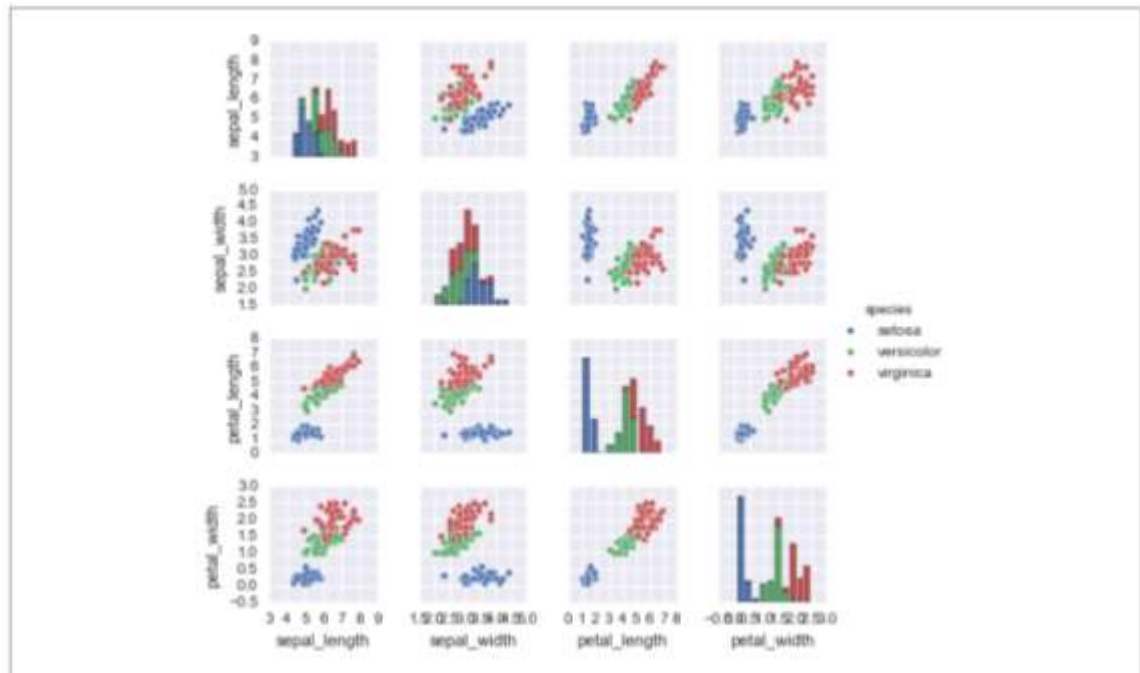


Figure 14: A visualization of the Iris dataset

For use in Scikit-Learn, we will extract the features matrix and target array from the DataFrame, which we can do using some of the Pandas DataFrame operations:

```
In[3]: X_iris = iris.drop('species', axis=1)
X_iris.shape
Out[3]: (150, 4)
In[4]: y_iris = iris['species']
y_iris.shape
Out[4]: (150,)
```

To summarize, the expected layout of features and target values is visualized in

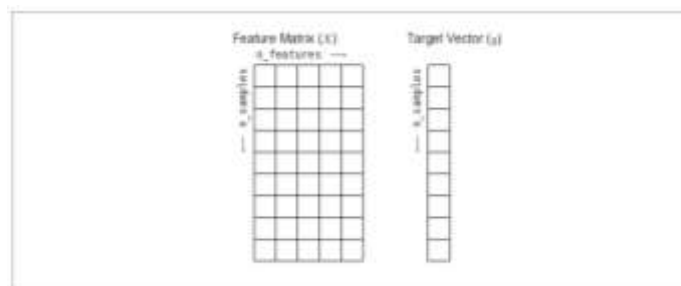


Figure 15: Scikit-Learn's data layout

8 PREDICTING THE HOUSE SALES PRICES USING LINEAR REGRESSION

8.1 Problem Statement

Based on the given dataset predicting the House Sale Prices using Linear Regression.

8.2 Firstly ,what is the problem?

We set out to use linear regression to predict housing prices in Iowa.

The problem is to build a model that will predict house prices with a high degree of predictive accuracy given the available data. More about it here. “With 77 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home.”

8.3 Mission 1: Importing Libraries

8.3.1 Where I got the data

The dataset is the prices and features of residential houses sold from 2006 to 2010 in Ames, Iowa. Check the references for the dataset link.

8.3.2 Libraries Used in this Project

First things first, Import the python libraries and dataset

```
In [26]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import figure
from sklearn.linear_model import LinearRegression
```

Figure 16: Importing Libraries

`pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. We'll use scikit-learn for the model training process, so we can focus on gaining intuition for the model-based learning approach to machine learning.

8.4 Mission 2: Introduction to the Data

To get familiar with this machine learning approach, we'll work with a dataset on sold houses in Ames, Iowa. Each row in the dataset describes the properties of a single house as well as the amount it was sold for. In this course, we'll build models that predict the final sale price from its other attributes. Specifically, we'll explore the following questions:

- Which properties of a house most affect the final sale price?
- How effectively can we predict the sale price from just its properties?

This dataset was originally compiled by Dean De Cock for the primary purpose of having a high-quality dataset for regression. Here are some of the columns:

1. Lot Area: Lot size in square feet.
2. Overall Qual: Rates the overall material and finish of the house.
3. Overall Cond: Rates the overall condition of the house.
4. Year Built: Original construction date.

```
In [2]: 1 ame=pd.read_csv('DataFile/AmesHousing.csv',delimiter='\\t')
        2 ame.head(6)
```

Out[2]:

PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition	SalePrice
526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Normal	215000
526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Normal	105000
526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Normal	172000
526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Normal	244000
527105010	60	RL	74.0	13630	Pave	NaN	IR1	Lvl	...	0	NaN	MnPrv	NaN	0	3	2010	WD	Normal	188600
527105030	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	6	2010	WD	Normal	196500

Figure 17: Ames Housing Dataset

Let's start by generating train and test datasets and getting more familiar with the data.

```
In [27]: 1 train_X=ame[['Garage Area','Gr Liv Area','Overall Cond']].iloc[:1461] # Select the first 1460 rows from data and assign
2 test_X=ame[['Garage Area','Gr Liv Area','Overall Cond']].iloc[1461:1931] # Select the remaining rows from data and assign to
3 print(train_X.info(),'\n') ## display information about each column in train_X
4 print(test_X.info(),'\n') ## display information about each column in test_X
5 ame['Target']=ame['SalePrice']
6 train_y=ame['Target'].iloc[:1461].to_frame()
7 test_y=ame['Target'].iloc[1461:1931].to_frame()
8 print(train_y.info(),'\n') ## display information about each column in train_y
9 print(test_y.info()) ## display information about each column in test_y

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1461 entries, 0 to 1460
Data columns (total 3 columns):
Garage Area    1461 non-null float64
Gr Liv Area    1461 non-null int64
Overall Cond   1461 non-null int64
dtypes: float64(1), int64(2)
memory usage: 34.3 KB
None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1461 entries, 0 to 1460
Data columns (total 1 columns):
Target         1461 non-null int64
dtypes: int64(1)
memory usage: 11.5 KB
None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 470 entries, 1461 to 1930
Data columns (total 3 columns):
Garage Area    470 non-null float64
Gr Liv Area    470 non-null int64
Overall Cond   470 non-null int64
dtypes: float64(1), int64(2)
memory usage: 11.1 KB
None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 470 entries, 1461 to 1930
Data columns (total 1 columns):
Target         470 non-null int64
dtypes: int64(1)
memory usage: 3.7 KB
None
```

Figure 18: Displaying information about each column.

Instructions:

- Read AmesHousing.txt into a dataframe using the tab delimiter (t) and assign to data.
- Select the first 1460 rows from data and assign to train.
- Select the remaining rows from data and assign to test.
- Use the dataframe.info() method to display information about each column.
- Read the data documentation to get more familiar with each column.
- Using the data documentation, determine which column is the target column we want to predict. Assign the column name as a string to target.

8.5 Mission 3: Simple Linear Regression

We'll start by understanding the univariate case of linear regression, also known as simple linear regression. The following equation is the general form of the simple linear regression model.

$$\hat{y} = ax_1 + a_0$$

\hat{y} represents the target column while x_1 represents the feature column we choose to use in our model. These values are independent of the dataset. On the other hand, a_0 and a_1 represent the parameter values that are *specific* to the dataset. The goal of simple linear regression is to find the

optimal parameter values that best describe the relationship between the feature column and the target column. The following diagram shows different simple linear regression models depending on the data:

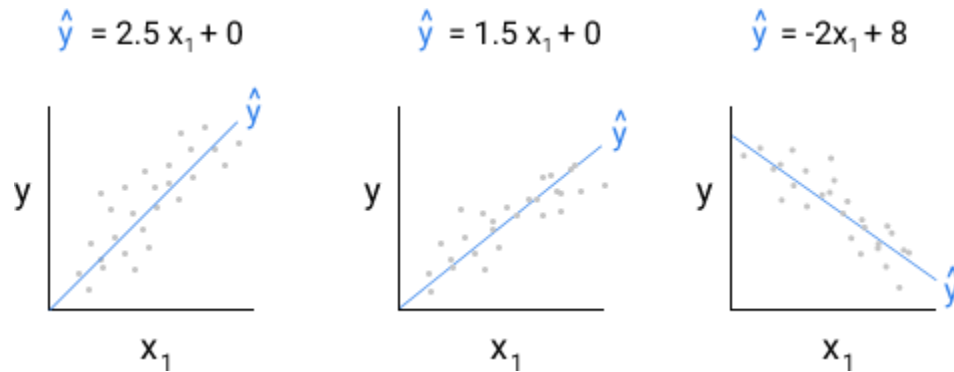


Figure 19: Different simple linear regression models

The first step is to select the feature, x_1 , we want to use in our model. Once we select this feature, we can use scikit-learn to determine the optimal parameter values a_1 and a_0 based on the training data. Because one of the assumptions of linear regression is that the relationship between the feature(s) and the target column is linear, we want to pick a feature that seems like it has the strongest correlation with the final sale price.

Instructions

- Create a figure with dimensions 7 x 15 containing three scatter plots in a single column:
- The first plot should plot the Garage Area column on the x-axis against the SalePrice column on the y-axis.
- The second one should plot the Gr Liv Area column on the x-axis against the SalePrice column on the y-axis.
- The third one should plot the Overall Cond column on the x-axis against the SalePrice column on the y-axis.
- Read more about these 3 columns in the data documentation.

```

In [6]: 1 import matplotlib.pyplot as plt
2
3 plt.figure(figsize=(12,18))
4 plt.xlabel('SalePrice')
5 plt.ylabel(('Garage Area', 'Gr Liv Area', 'Overall Cond'))
6 plt.subplot(3,1,1)
7 plt.scatter((np.array(ame['Garage Area']).astype(int).tolist()),(np.array(ame['SalePrice']).astype(int).tolist()))
8 plt.subplot(3,1,2)
9 plt.scatter((np.array(ame['Gr Liv Area']).astype(int).tolist()),(np.array(ame['SalePrice']).astype(int).tolist()))
10 # plt.scatter((ame['Gr Liv Area']).to_list(),ame['SalePrice'].to_list())
11 plt.subplot(3,1,3)
12 plt.scatter((np.array(ame['Overall Cond']).astype(int).tolist()),(np.array(ame['SalePrice']).astype(int).tolist()))
13 # plt.scatter(ame['Overall Cond'].to_list(),ame['SalePrice'].to_list())

Out[6]: <matplotlib.collections.PathCollection at 0xca04cb0>

```

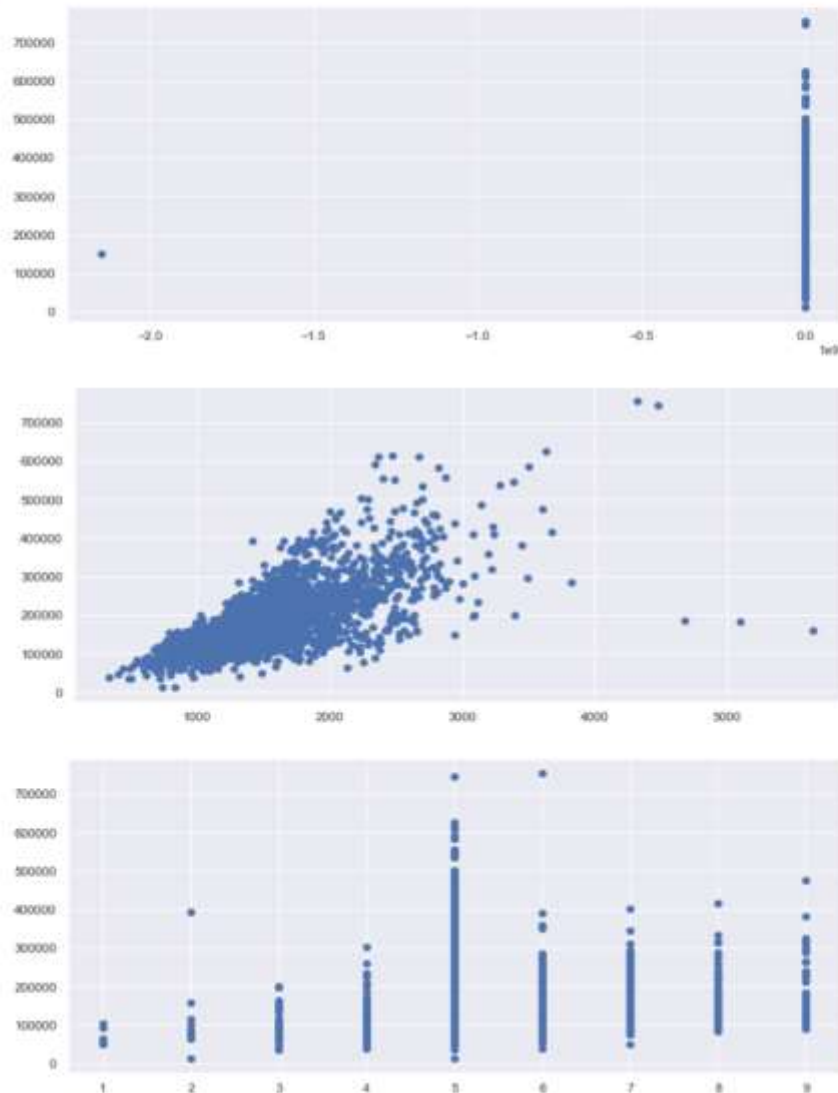


Figure 20: Correlation between various parameters with 'SalePrice'

First plot , plots the 'Overall Cond' column on the x-axis against the 'SalePrice' column on the y-axis. The second one plots the 'Gr Liv Area' column on the x-axis against the 'SalePrice' column on the y-axis. The third one plots the 'Garage Area' column on the x-axis against the 'SalePrice' column on the y-axis.

8.6 Mission 4: Least Squares

From the last screen, we can tell that the Gr Liv Area feature correlates the most with the SalePrice column. We can confirm this by calculating the correlation between pairs of these columns using the `pandas.DataFrame.corr()` method:

```
print(train[['Garage Area', 'Gr Liv Area', 'Overall Cond', 'SalePrice']].corr())
```

```
In [7]: train[['Garage Area', 'Gr Liv Area', 'Overall Cond', 'SalePrice']].corr()
Out[7]:
```

	Garage Area	Gr Liv Area	Overall Cond	SalePrice
Garage Area	1.000000	0.484892	-0.153754	0.640401
Gr Liv Area	0.484892	1.000000	-0.115643	0.706780
Overall Cond	-0.153754	-0.115643	1.000000	-0.101697
SalePrice	0.640401	0.706780	-0.101697	1.000000

Figure 21: Correlation Table

The correlation between Gr Liv Area and SalePrice is around 0.706, which is the highest. Recall that the closer the correlation coefficient is to 1.0, the stronger the correlation. Here's the updated form of our model:

$$y^{\wedge}=a_1*\text{Gr Liv Area}+a_0$$

Let's now move on to understanding the model fitting criteria.

8.6.1 Residual Sum of Squares

To find the optimal parameters for a linear regression model, we want to optimize the model's residual sum of squares (or RSS). If you recall, residual (often referred to as errors) describes the difference between the predicted values for the target column (y^{\wedge}) and the true values (y):

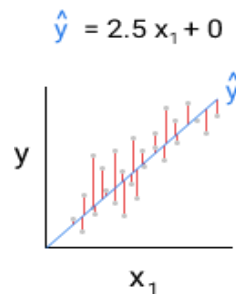


Figure 22: Errorbar graph

We want this difference to be as small as possible. Calculating RSS involves summing the *squared* errors:

$$RSS = (y_1 - y_1^{\wedge})^2 + (y_2 - y_2^{\wedge})^2 + \dots + (y_n - y_n^{\wedge})^2$$

We can shorten this to:

$$RSS = \sum_{i=1}^n (y_i - y_i^{\wedge})^2$$

If you recall, the calculation for RSS *seems* very similar to the calculation for MSE (mean squared error). Here's the formula for MSE, adapted for our new notation:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - y_i^{\wedge})^2$$

While we used the MSE on the test set, it's clear that the goal of minimizing RSS on the training set when training is a good idea.

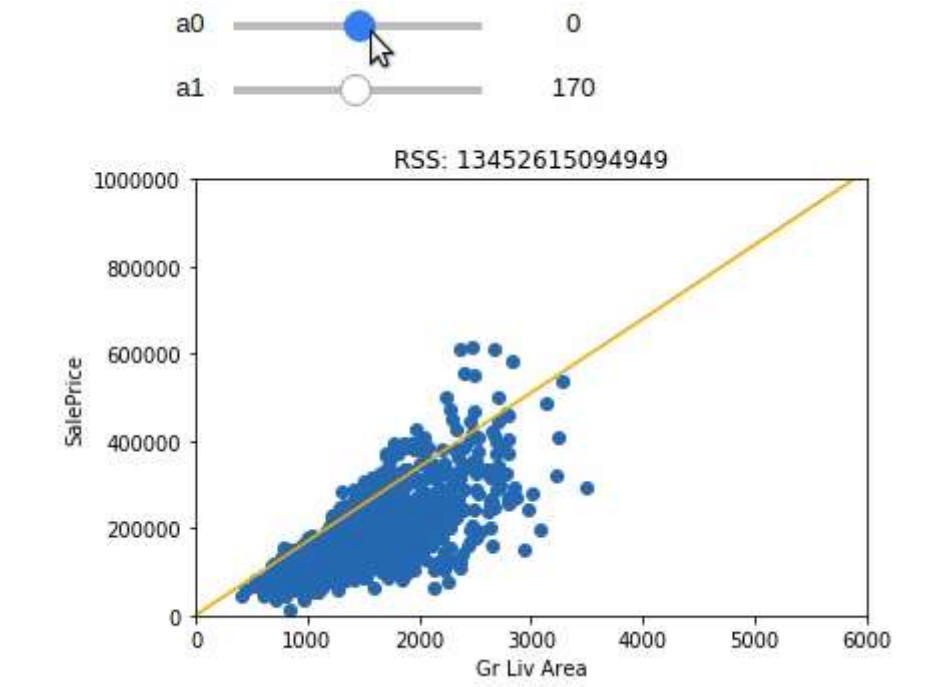


Figure 23: RSS on training set

8.7 Mission 5: Using Scikit-Learn to Train and Predict

Let's now use scikit-learn to find the optimal parameter values for our model. The scikit-learn library was designed to easily swap and try different models. Because we're familiar with the scikit-learn workflow for k-nearest neighbors, switching to using linear regression is straightforward.

Instead of working with the `sklearn.neighbors.KNeighborsRegressors` class, we work with the `sklearn.linear_model.LinearRegression` class. The `LinearRegression` class also has its own `fit()` method. Specific to this model, however, is the `coef_` and `intercept_` attributes, which returns a_1 (a_1 to a_n if it were a multivariate regression model) and a_0 accordingly.

```
In [8]: 1 regr=LinearRegression()
2 regr.fit(train_X['Gr Liv Area'].to_frame(),train_y)
3 y_pred=regr.predict(test_X['Gr Liv Area'].to_frame()).astype(int) # Predict using the linear model

In [9]: 1 print(regr.fit(test_X,y_pred) ) # Fit linear model.
2 # print('score ',regr.score(test_X,y_pred)) # Returns the coefficient of determination R^2 of the prediction.
3 a1= regr.coef_ # Estimated coefficients for the linear regression problem
4 a0=regr.intercept_
5 print('coefficient is',a1,'\n','intercept is',a0)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
coefficient is [[6.05323461e-05 1.16924615e+02 9.94449220e-03]]
intercept is [5367.34146852]
```

Figure 24: Using Scikit-Learn to Train and Predict

Instructions

- Import and instantiate a linear regression model.
- Fit a linear regression model that uses the feature and target columns we explored in the last 2 screens. Use the default arguments.
- Display the coefficient and intercept of the fitted model using the `coef_` and `intercept_` attributes.
- Assign a_1 to `a1` and a_0 to `a0`.

8.8 Mission 6: Making Predictions

In the last step, we fit a univariate linear regression model between the Gr Liv Area and SalePrice columns. We then displayed the single coefficient and the residual value. If we refer back to the format of our linear regression model, the fitted model can be represented as:

$$y^{\wedge}=116.86624683x_1+5366.82171006$$

One way to interpret this model is "for every 1 square foot increase in above ground living area, we can expect the home's value to increase by approximately 116.87 dollars".

We can now use the `predict()` method to predict the labels using the training data and compare them with the actual labels. To quantify the fit, we can use mean squared error. Let's also perform simple validation by making predictions on the test set and calculate the MSE value for those predictions as well.

Instructions

- Use the fitted model to make predictions on both the training and test sets.

Finally, let's visualize the results by plotting first the raw data, and then this model fit

```
In [10]: 1 plt.figure(figsize=(12,6))
2 # plt.scatter(np.array(test_X[['Garage Area']].astype(int)).tolist(),np.array(test_y).tolist(),color='black')
3 plt.scatter(np.array(test_X[['Gr Liv Area']].astype(int)).tolist(),np.array(test_y).tolist())
4 # plt.scatter(np.array(test_X[['Overall Cond']].astype(int)).tolist(),np.array(test_y).tolist(),color='green')
5 plt.plot(test_X['Gr Liv Area'].to_frame(),y_pred)
6 plt.xticks()
7 plt.yticks()
8 plt.show()
```

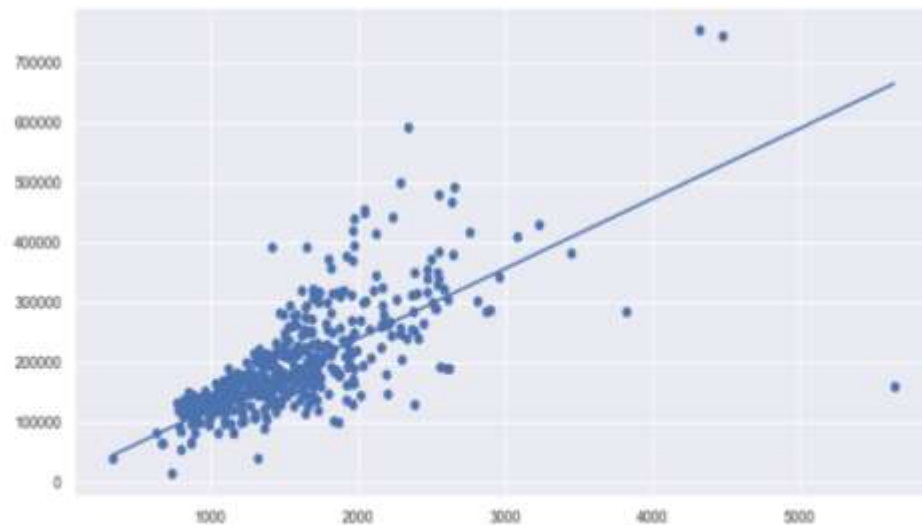


Figure 25: simple linear regression fit to the data

9 REFERENCES:

- [1] “Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow”, Packt Publishing, 2 edition, Sebastian Raschka & Vahid Mirjalili
- [2] “Data Mining: Practical Machine Learning Tools and Techniques”, 4th edition, Published by Morgan Kaufmann, Ian H. Witten, Eibe Frank, Mark A.Hall, Christopher J. Pal
- [3] “Python Data Science Handbook”, Published by O’Reilly Media, Jake VanderPlas