Book Recommender System for Readers in a University Library.
(Major Project)

M.Ruthvik Mohan

mruthvikmohan_@_protonmail.com

Swami Vivekananda Institute Of Technology,Hyderabad,India.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS AND ABBREVIATIONS

Abbreviation	Full Form
CF	Collaborative Filtering
KNN	K Nearest Neighbor
RS	Recommendation System
JS	Jaccard Similarity

ABSTRACT

Presently a-days, many significant internet business and websites are utilizing suggestion frameworks to give important proposals to their clients and customers. The suggestions could be founded on different parameters, for example, things mainstream on the company's Website; client/ customer qualities, for example, land area or other statistic data; or past purchasing conduct of top clients/ customers. In this project, a book suggestion motor is proposed which utilizes content-based filtering technique for recommending the books to the customer. The content based filtering technique doesn't requires a big amount of data to get trained and can work on significantly less amount of data even from a single customer. The Algorithm used here is KNN with Cosine similarity.

CHAPTER 1

INTRODUCTION

A Recommendation System, in genuine definition can be described to as a framework that can run on grouped/non grouped environment by taking client/customer's online impression as one of its input and producing a likely result for the client along these lines giving its clients an expectation closer to the real world. Recommender system generally require a huge dataset and a quick registering framework that can perform examination on the equivalent within seconds.

Recommendation Systems, in easier terms are programs that are information escalated and include complex example coordinating on a lot of predefined parameters and they become proficient with the expansion in the size of the substance being sustained to them. Recommender frameworks represents client inclinations with the end goal of proposing things to buy or look at. They have become basic applications in electronic business also, giving proposals that viably prune huge data spaces with the goal that clients are coordinated toward those things that best address their issues and interests. An assortment of systems have been proposed till today for performing proposals. The systems for example, content-based, communitarian, information based and statistic are utilized for proposals.

In the proposed book Recommendation System, books will be shown by using content based filtering technique, which can work even in a smaller amount of data.

CHAPTER 2

LITERATURE SURVEY

2.1 INTRODUCTION

Online Book Recommendation System using Collaborative Filtering

https://iopscience.iop.org/article/10.1088/1742-6596/1362/1/012130/pdf

Recommendation System (RS) is software that suggests similar items to a purchaser based on his/her earlier purchases or preferences. RS examines huge data of objects and compiles a list of those objects which would fulfil the requirements of the buyer. Nowadays most ecommerce companies are using Recommendation systems to lure buyers to purchase more by offering items that the buyer is likely to prefer. Book Recommendation System is being used by Amazon, Barnes and Noble, Flipkart, Goodreads, etc. to recommend books the customer would be tempted to buy as they are matched with his/her choices. The challenges they face are to filter, set a priority and give recommendations which are accurate. RS systems use Collaborative Filtering (CF) to generate lists of items similar to the buyer's preferences. Collaborative filtering is based on the assumption that if a user has rated two books then to a user who has read one of these books, the other book can be recommended (Collaboration). CF has difficulties in giving accurate recommendations due to problems of scalability, sparsity and cold start. Therefore this paper proposes a recommendation that uses Collaborative filtering with Jaccard Similarity (JS) to give more accurate recommendations. JS is based on an index calculated for a pair of books. It is a ratio of common users (users who have rated both books) divided by the sum of users who have rated the two books individually. Larger the number of common users higher will be the JS Index and hence better recommendations. Books with high JS index (more recommended) will appear on top of the recommended books list.

Book Recommender System
https://webpages.charlotte.edu/nmatta1/cloudproject/Project_Report.pdf

Both the online entertainment and e-commerce companies are trying to retain their customers by taking their access to the website to more personalized manner. So, provide additional recommendations based on users past activity. Our project would be one of such system that recommends additional books that belongs to similar genre, author or publisher. Such systems result in increase in

rate of purchase, these may also include unplanned purchases driven by surprise factor from the recommendations made.
Personalized Book Recommendation System using Machine Learning Algorithm

https://thesai.org/Downloads/Volume12No1/Paper_26-Personalized_Book_Recommendation_System.pdf

As the amounts of online books are exponentially increasing due to COVID-19 pandemic, finding relevant books from a vast e-book space becomes a tremendous challenge for online users. Personal recommendation systems have been emerged to conduct effective search which mine related books based on user rating and interest. Most of these existing systems are user-based ratings where content-based and collaborativebased learning methods are used. These systems' irrationality is their rating technique, which counts the users who have already been unsubscribed from the services and no longer rate books. This paper proposed an effective system for recommending books for online users that rated a book using the clustering method and then found a similarity of that book to suggest a new book. The proposed system used the K-means Cosine Distance function to measure distance and Cosine Similarity function to find Similarity between the book clusters. Sensitivity, Specificity, and F Score were calculated for ten different datasets. The average Specificity was higher than sensitivity, which means that the classifier could re-move boring books from the reader's list. Besides, a receiver operating characteristic curve was plotted to find a graphical view of the classifiers' accuracy. Most of the datasets were close to the ideal diagonal classifier line and far from the worst classifier line. The result concludes that recommendations, based on a particular book, are more accurately effective than a user-based recommendation system.

2.2 EXISTING SYSTEM

Following are some of the existing book recommendation engines used by the top rated book purchasing websites.

The existing engines make use of conventional algorithms for recommendations. In Content based Recommendation Engine, system generates recommendations from source based on the features associated with products and the user‟s information.

Content-based recommenders treat recommendation as a user-specific classification problem and learn a classifier for the user's likes and dislikes

based on product features. In Collaborative recommendation engines, suggestions are generated on the basis of ratings given by group of people. It locates peer users with a rating history similar to the current user and generates recommendations for the user.

In Context based Recommendation Engine, system requires the additional data about the context of item consumption like time, mood and behavioural aspects. These data may be used to improve the recommendation compared to what could be performed without this additional source of information.

2.3 DISADVANTAGES OF EXISTING SYSTEM

The major problem with existing system is it needs a good amount of data to even work considerably good which can be a challenge for small businesses and startups.

The data which is to be used for training should be precise and filtered. Any mistake in the data can lead to inaccuracy of the whole system.

2.4 PROPOSED SYSTEM

The Proposed Book Recommender System will use Content based filtering technique using cosine similarity algorithm. This methodology depends on making a plenty of parameters to describe a particular product.. Thinking about an Book as an model the potential parameters could be Author, Publisher, Year Published etc.. The bigger the parameter set the better and simpler it is to coordinate examples with customer's profile and his online impression. The parameters would then be able to be assigned weight and consequently a relative need is set for every one of the parameter. All these parameters are then used to make a customer's profile. Henceforth we see that the system finds out about the client inclinations and choice patters by his online impression.

A website System can be made where the user can select the books which he/she likes or the book which user is currently reading. In real-time, the system will be recommending them other books. Such system in future scope can also be integrated with ecommerce website to increase sales.

2.5 ADVANTAGES OF PROPOSED SYSTEM

The major advantage of using Content based filtering algorithm is no requirement of huge dataset.The Content Based filtering algorithm is flexible in nature.

CHAPTER 3

ANALYSIS

3.1 Software Design Method

Software design is the process of conceptualizing the software requirements of a software implementation. Software design takes user requirements as an issue and seeks to find the best solution. Once the software is conceptualized, a plan is created to find the best design to implement the desired solution.

There are several variations in software design. Let's take a quick look at them:

3.1.1 Structured Design

Structured design conceptualizes the problem into several well-organized solution elements. It is something about the essentially solutions of design. The advantage of structured design is that you can better understand what is resolved how the problem. The structured design, the designer makes it easy to further concentrate on the problem.

Structured design is based mainly on the "divide and conquer" strategy. In this strategy, is divided into a plurality of small problem is a problem, until the whole issue is resolved, small problems will be resolved one by one.

Small problems are resolved by the components of the solution. The structured design emphasizes that these modules are properly organized to achieve an accurate solution.

These modules are arranged hierarchically. They communicate with each other. A well-structured design always adheres to some rules for communication between multiple modules:
Cohesion - grouping of all functionally related elements.
Coupling communications between different modules.

A well-structured design with high cohesion and low coupling.

3.1.2 Function-oriented design

In a function-oriented design, a system consists of many small subsystems called functions. These features allow you to perform important tasks on your system. The system is displayed as a top view of all features.
Function-oriented design inherits some characteristics of structured design using divide-and-conquer method.

This design mechanism provides a means of abstraction by dividing the entire system into smaller functions and hiding information and its operations. These functional modules can exchange information with each other by passing information or using globally available information.

Another characteristic of a function is that when a program calls a function, that function changes the state of the program. This may not be acceptable in other modules. Function-oriented design works well when the state of the system is not important and the program / function works on the input rather than the state.

Design process

- The entire system is displayed as a data flow in the system using a data flow diagram.
- DFD shows how a function changes the data and state of the entire system.
- The entire system is logically divided into smaller units called functions based on how it behaves within the system.
- The functions are then implemented.

3.1.3 Object-oriented design

Object-oriented design avoids entities and their properties rather than the features contained in the software system. This design strategy focuses on the entity and its properties. The overall concept of a software solution revolves around the units involved.

Let's review the important concepts of object-oriented design.

- **Objects** - All entities involved in solution design are called objects. For example, people, banks, businesses, and customers are treated as objects. Each entity has several attributes associated with it, and there are several methods to execute on those attributes.
- **Class** - A class is a generalized description of an object. The object is an instance of the class. The class defines all the attributes that an object can have and the methods that define the functionality of the object.

- In solution design, attributes are stored as variables and functions are defined by methods or procedures.

- **Encapsulation** - OOD bundles attributes (data variables) and methods (operations on data), which is called encapsulation. Encapsulation not only bundles important information about an object, but also limits access to data and methods from the outside. This is known as hiding information.

- **Inheritance** - OOD allows you to stack similar classes hierarchically. Subclasses or subclasses can import, implement, and reuse valid variables and methods from direct superclasses. This property of OOD is called inheritance. This makes it easy to define a particular class and create a generalized class from a particular class.

- **Polymorphism** - The polymorphism OOD language provides a mechanism that allows methods that perform similar tasks but have different arguments to have the same name. This is called polymorphism and allows you to perform different types of tasks with a single interface. The appropriate part of the code is executed, depending on how the function is called.

Design process

The software design process can be recognized as a series of well-defined steps. Depending on your design approach (functional or object-oriented), it may include the following steps:

- Solution designs are created from requirements or previously used systems and / or system flow charts.
- Objects are identified and grouped into classes by the name of attribute characteristic similarity.
- The class hierarchy and the relationships between them are defined.

- The application framework is defined.

3.1.4 Software design approach

Two common approaches to software design are:

- **Top-down design**

It is known that the system consists of multiple subsystems and contains many components. In addition, these subsystems and components have their own set of subsystems and components, allowing you to create a hierarchical structure in your system.

The top-down design takes the entire software system as a unit and disassembles it to get multiple subsystems or components based on some characteristics. Each subsystem or component is then treated as a system and further subdivided. This process continues until you reach the lowest system level in the top-down hierarchy.

The top-down design starts with a generalized system model and defines its more specific parts. Putting all the components together creates the entire system.

The top-down design is suitable when you need to design a software solution from scratch and you do not know the specific details.

- **Bottom-up Design**

The bottom-up design model starts with the most specific and basic components. Continue assembling high-level components with basic or low-level components. Continue to create higher level components until the desired system is developed as a single component. The higher the level, the greater the set of abstractions.

The bottom-up strategy is suitable when you need to build a system from an existing system that can use basic primitives.

Both top-down and bottom-up approaches are not individually practical. Instead, the appropriate combination of both is used.

3.2 ARCHITECTURE DIAGRAM:

- **Research Phase Explanation**

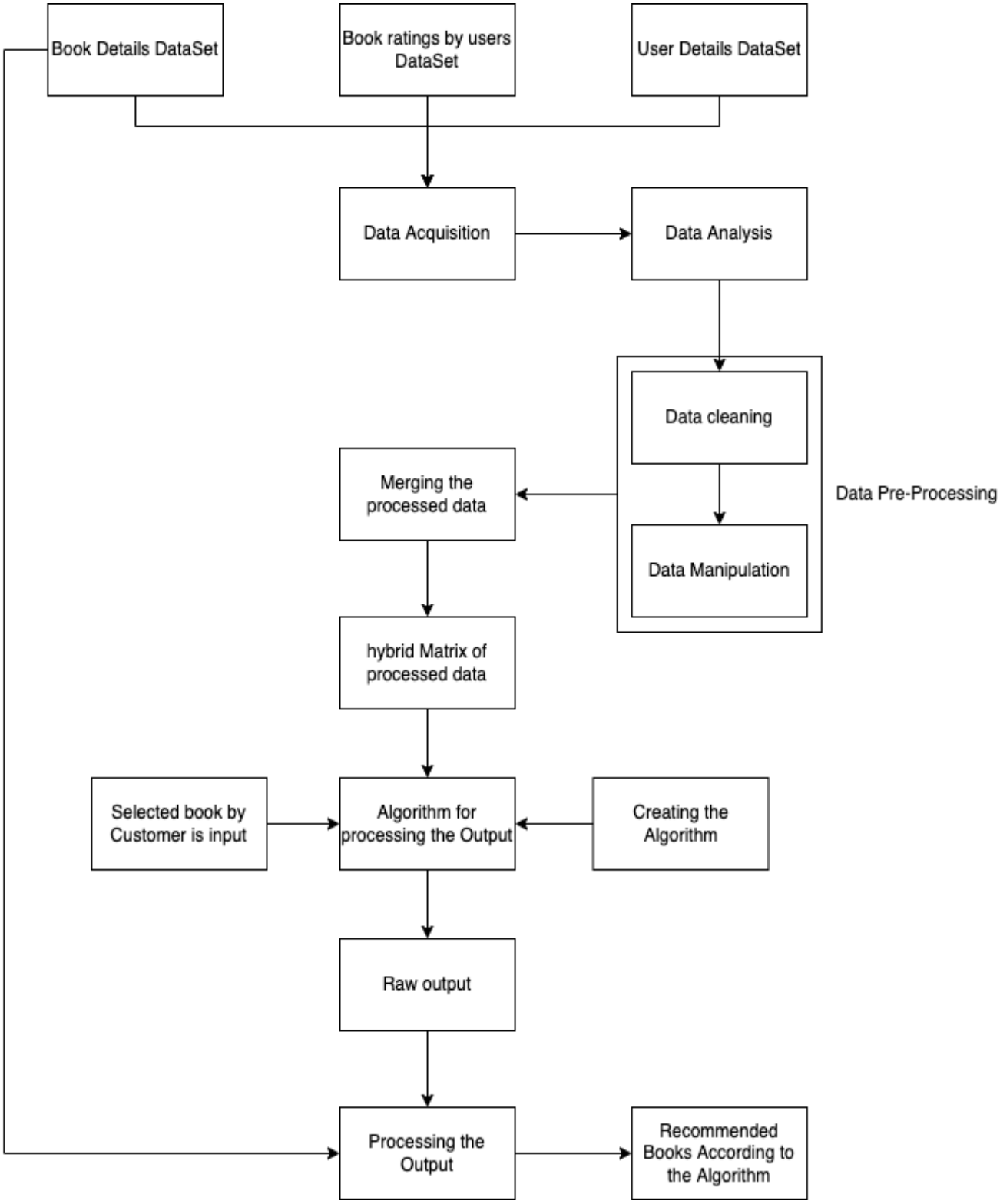


Fig1 Architecture diagram

Data Acquisition:

Data Acquisition means loading/importing the necessary Data into python workspace. Converting the normal tabular data like csv files etc. into python understandable data such as “nd-array” object of “numpy” object.

Data Analysis:

Data Analysis means understanding the basics of the data being loaded. To have knowledge of number of row and columns, type of data each column has, their statistics and Graphical Structures. So that we can perform Data pre-processing step easily.

Data pre-processing

Data pre-processing means cleaning and preparing the data for giving it as input to the algorithm etc.

1. Cleaning: removing or handling the empty values in the dataset.
2. Data Manipulation: working with encoders transposing the row into columns vice versa and other pre-processing steps.

Creating: instantiating the multiple algorithms which can accept input and produce output and supplying them with the train data to start the training.

Inference: providing the input to Algorithm to process the output recommendation as it is a un-supervised algorithm.

- **GUI PHASE**

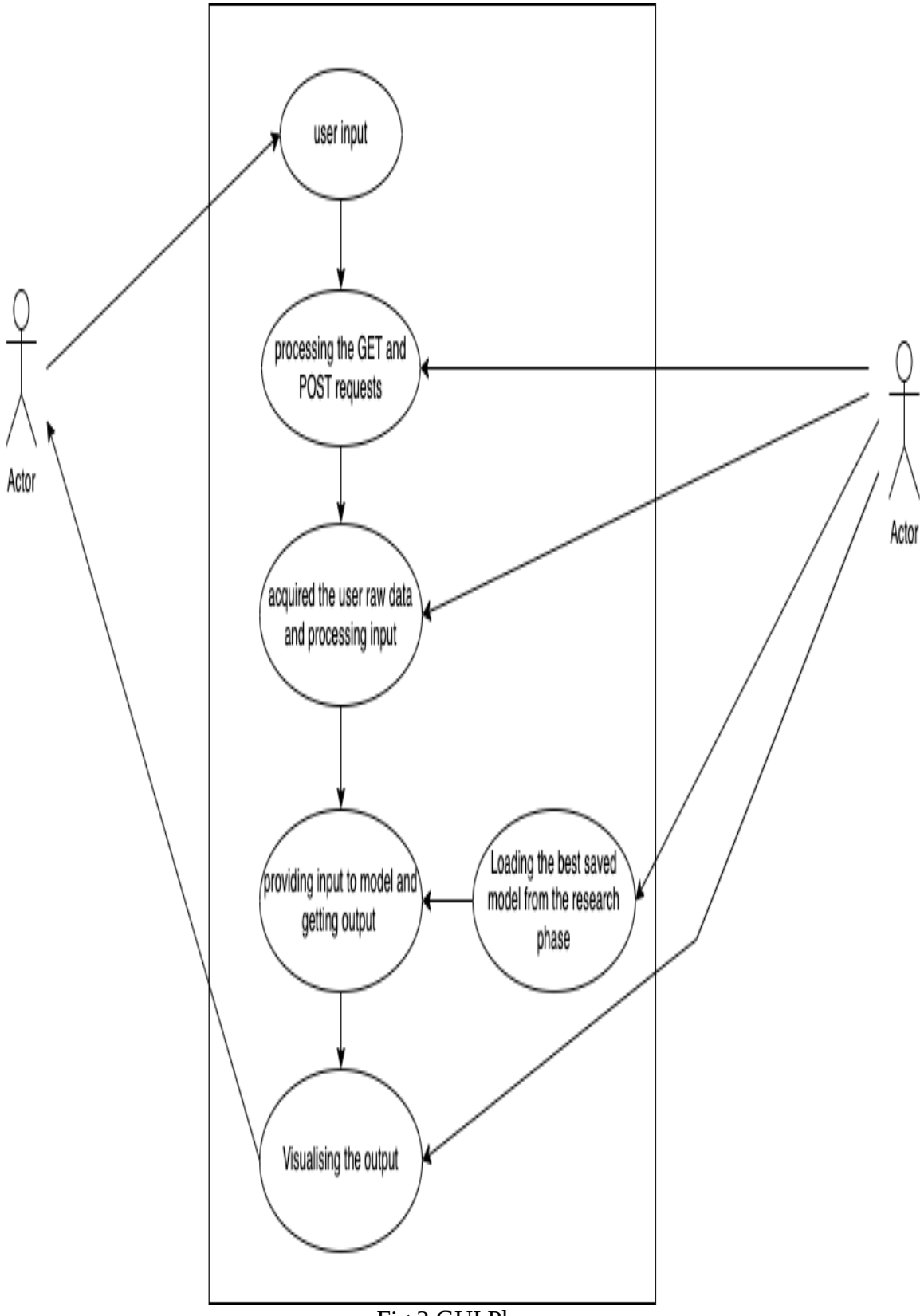


Fig 2 GUI Phase

- **DFD**

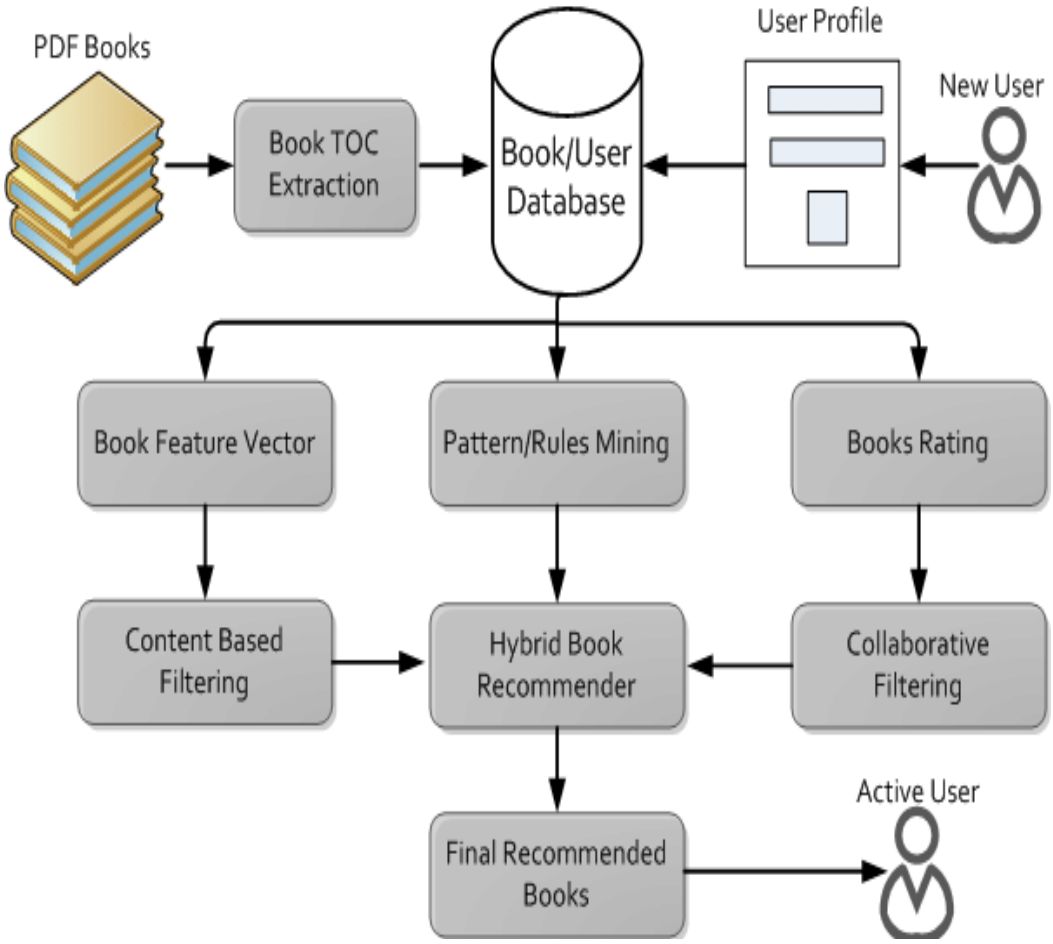


Fig 3 DFD Flowchart

CHAPTER 4

ALGORITHM(S) USED IN THE WORK

4.1 K Nearest Neigbhors

K-NN is a **non-parametric** and **lazy learning algorithm**. Non-parametric means there is no assumption for underlying data distribution i.e. the model structure determined from the dataset.

It is called Lazy algorithm because it does not need any training data points for model generation. All training data is used in the testing phase which makes training faster and testing phase slower and costlier.

K-Nearest Neighbor (K-NN) is a simple algorithm that stores all the available cases and classifies the new data or case based on a similarity measure.

K-NN classification

In K-NN classification, the output is a **class membership**. An object is classified by a plurality vote of its neighbors, with the object being assigned to the **class most common among its k nearest neighbors** (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.

To determine which of the K instances in the training dataset are most similar to a new input, **a distance measure is used**. For real-valued input variables, the most popular distance measure is the Euclidean distance.

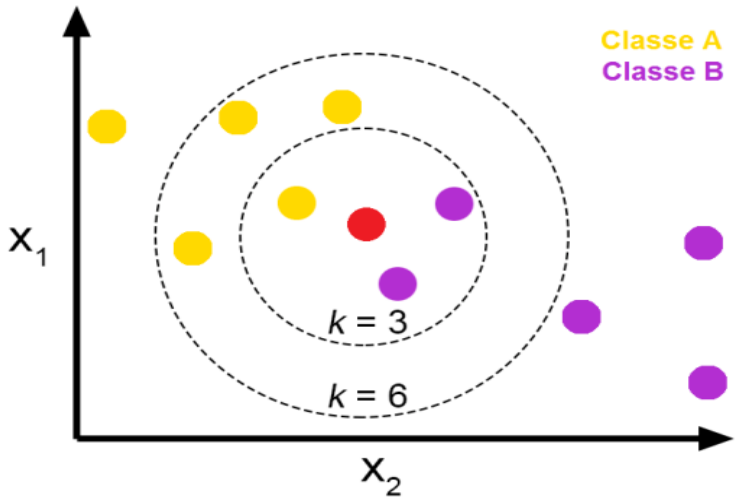


Fig 4 KNN Classification
Source: Towards Data Science

4.2 The Euclidean distance

The Euclidean distance is the most common distance metric used in **low dimensional data sets**. It is also known as the **L2 norm**. The Euclidean distance is the usual manner in which distance is measured in the real world.

Where p and q are n-dimensional vectors and denoted by **p** = (p1, p2,…, pn) and **q** = (q1, q2,…, qn) represent the n attribute values of two records.

While Euclidean distance is useful in low dimensions, it **doesn't work well in high dimensions and for categorical variables**. The drawback of Euclidean distance is that it **ignores the similarity between attributes**. Each attribute is treated as totally different from all of the attributes.

Other popular distance measures :

- **Hamming Distance**: Calculate the distance between binary vectors.
- **Manhattan Distance**: Calculate the distance between real vectors using the sum of their absolute difference. Also called City Block Distance.
- **Minkowski Distance**: Generalization of Euclidean and Manhattan distance.

Steps to be carried out during the K-NN algorithm are as follows:

1. Divide the data into training and test data.
2. Select a value K.
3. Determine which distance function is to be used.
4. Choose a sample from the test data that needs to be classified and compute the distance to its n training samples.
5. Sort the distances obtained and take the k-nearest data samples.
6. Assign the test class to the class based on the majority vote of its k neighbors.

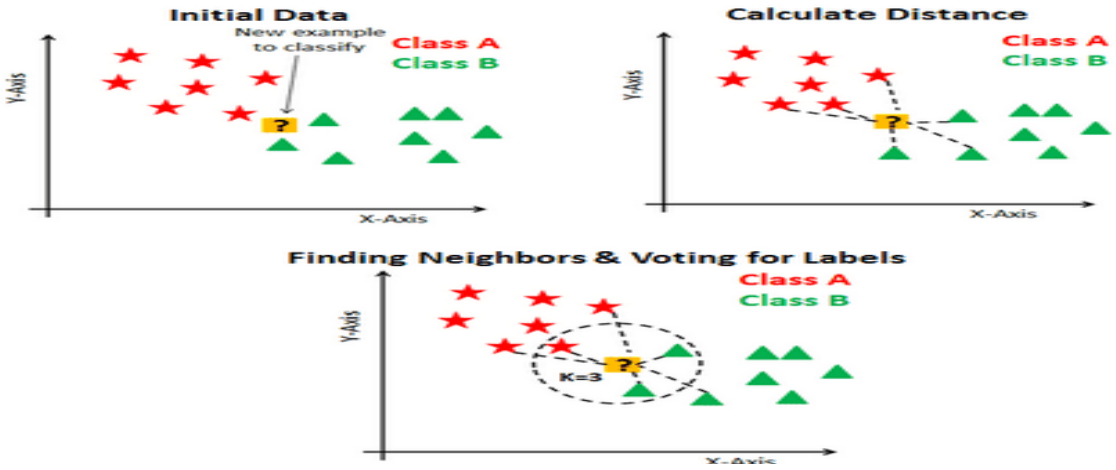


Fig 5 Euclidean Distance

Source: DataCamp

Performance of the K-NN algorithm is influenced by three main factors :

1. The **distance function** or distance metric used to determine the nearest neighbors.
2. The **decision rule used to derive a classification** from the K-nearest neighbors.
3. The **number of neighbors** used to classify the new example.

Advantages of K-NN:

1. The K-NN algorithm is very easy to implement.
2. Nearly optimal in the large sample limit.
3. Uses local information, which can yield highly adaptive behaviour.
4. Lends itself very easily to parallel implementation.

CHAPTER 5

TOOLS USED FOR IMPLEMENTATION

5.1 PYTHON

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales. Van Rossum led the language community until stepping down as leader in July 2018.Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural. It also has a comprehensive standard library. The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https:// www. python .org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

Invoking the Interpreter

The Python interpreter is usually installed as /usr/local/bin/python3.7 on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.7
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

On Windows machines, the Python installation is usually placed in C:\Python37, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python37
```

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support readline. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. If nothing appears to happen, or if ^P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

A second way of starting the interpreter is python -c command [arg] ..., which executes the statement(s) in command, analogous to the shell's -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote command in its entirety with single quotes.

Some Python modules are also useful as scripts. These can be invoked using python -m module [arg] ..., which executes the source file for module as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script.

Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the argv variable in the sys module. You can access this list by executing import sys. The length of the list is at least one; when no script and no arguments are given, sys.argv[0] is an empty string. When the script name is given as '-' (meaning standard input), sys.argv[0] is set to '-'. When -ccommand is used, sys.argv[0] is set to '-c'. When -m module is used, sys.argv[0] is set to the full name of the located module. Options found after -c command or -m module are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

Interactive Mode

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (>>>); for continuation lines it prompts with the secondary prompt, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.7
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>>
```

```
>>>the_world_is_flat = True
>>> ifthe_world_is_flat:
... print("Be careful not to fall off!")
...
Be careful not to fall off!
```

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and …): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1  # and this is the second comment
# ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

Python uses duck typing and has typed objects but untyped variable names. Type constraints are not checked at compile time; rather, operations on an object may fail, signifying that the given object is not of a suitable type. Despite being dynamically typed, Python is strongly typed, forbidding operations that are not well-defined (for example, adding a number to a string) rather than silently attempting to make sense of them.

Python allows programmers to define their own types using classes, which are most often used for object-oriented programming. New instances of classes are constructed by calling the class (for example, `SpamClass()` or `EggsClass()`), and the classes are instances of the metaclass `type` (itself an instance of itself), allowing metaprogramming and reflection.

Before version 3.0, Python had two kinds of classes: old-style and new-style. The syntax of both styles is the same, the difference being whether the class `object` is inherited from, directly or indirectly (all new-style classes inherit from `object` and are instances of `type`). In versions of Python 2 from Python 2.2 onwards, both kinds of classes can be used. Old-style classes were eliminated in Python 3.0.

The long term plan is to support gradual typing[77] and from Python 3.5, the syntax of the language allows specifying static types but they are not checked in the default implementation, CPython. An experimental optional static type checker named mypy supports compile-time type checking.

Python has the usual C language arithmetic operators (`+`, `-`, `*`, `/`, `%`). It also has `**` for exponentiation, e.g. `5**3 == 125` and `9**0.5 == 3.0`, and a new matrix multiply `@` operator is included in version 3.5.[80] Additionally, it has a unary operator (`~`), which essentially inverts all the bits of its one argument. For integers, this means `~x=-x-1`.[81] Other operators include bitwise shift operators `x << y`, which shifts `x` to the left `y` places, the same as `x*(2**y)`, and `x >> y`, which shifts `x` to the right `y` places, the same as `x//(2**y)`.[82]

The behavior of division has changed significantly over time:[83][why?]

- Python 2.1 and earlier use the C division behavior. The `/` operator is integer division if both operands are integers, and floating-point division otherwise. Integer division rounds towards 0, e.g. `7/3 == 2` and `-7/3 == -2`.
- Python 2.2 changes integer division to round towards negative infinity, e.g. `7/3 == 2` and `-7/3 == -3`. The floor division `//` operator is introduced. So `7//3 == 2`, `-7//3 == -3`, `7.5//3 == 2.0` and `-7.5//3 == -3.0`. Adding `from __future__ import division` causes a module to use Python 3.0 rules for division (see next).
- Python 3.0 changes `/` to be always floating-point division. In Python terms, the pre-3.0 `/` is classic division, the version-3.0 `/` is real division, and `//` is floor division.

Rounding towards negative infinity, though different from most languages, adds consistency. For instance, it means that the equation `(a + b)//b == a//b + 1` is always true. It also means that the equation `b*(a//b) + a%b == a` is valid for both positive and negative values of `a`. However, maintaining the validity of this equation means that while the result of `a%b` is, as expected, in the half-open interval [0, b), where `b` is a positive integer, it has to lie in the interval (b, 0] when `b` is negative.

Python provides a `round` function for rounding a float to the nearest integer. For tie-breaking, versions before 3 use round-away-from-zero: `round(0.5)` is 1.0, `round(-0.5)` is −1.0. Python 3 uses round to even: `round(1.5)` is 2, `round(2.5)` is 2.

Python allows boolean expressions with multiple equality relations in a manner that is consistent with general use in mathematics. For example, the expression `a < b < c` tests whether `a` is less than `b` and `b` is less than `c`. C-derived languages interpret this expression differently: in C, the expression would first evaluate `a < b`, resulting in 0 or 1, and that result would then be compared with `c`.

Python has extensive built-in support for arbitrary precision arithmetic. Integers are transparently switched from the machine-supported maximum fixed-precision (usually 32 or 64 bits), belonging to the python type `int`, to arbitrary precision, belonging to the Python type `long`, where needed. The latter have an "L" suffix in their textual representation. (In Python 3, the distinction between the `int` and `long` types was eliminated; this behavior is now entirely contained by the `int` class.) The `Decimal` type/class in module `decimal` (since version 2.4) provides decimal floating point numbers to arbitrary precision and several

rounding modes. The Fraction type in module fractions (since version 2.6) provides arbitrary precision for rational numbers.https://en.wikipedia.org/wiki/Python_(programming_language) - cite_note-91

Due to Python's extensive mathematics library, and the third-party library NumPy that further extends the native capabilities, it is frequently used as a scientific scripting language to aid in problems such as numerical data processing and manipulation.

5.1.1 PYTHON MODULES USED in the RESEARCH

NUMPY

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

The Python programming language was not initially designed for numerical computing, but attracted the attention of the scientific and engineering community early on, so that a special interest group called matrix-sig was founded in 1995 with the aim of defining an array computing package. Among its members was Python designer and maintainer Guido van Rossum, who implemented extensions to Python's syntax (in particular the indexing syntax) to make array computing easier.

An implementation of a matrix package was completed by Jim Fulton, then generalized by Jim Hugunin to become Numeric,[4] also variously called Numerical Python extensions or NumPy. Hugunin, a graduate student at Massachusetts Institute of Technology (MIT), joined the Corporation for National Research Initiatives (CNRI) to work on JPython in 1997 leaving Paul Dubois of Lawrence Livermore National Laboratory (LLNL) to take over as maintainer. Other early contributors include David Ascher, Konrad Hinsen and Travis Oliphant.

A new package called Numarray was written as a more flexible replacement for Numeric. Like Numeric, it is now deprecated. Numarray had faster operations for large arrays, but was slower than Numeric on small ones, so for a time both packages were used for different use cases. The last version of Numeric v24.2 was released on 11 November 2005 and numarray v1.5.2 was released on 24 August 2006.

There was a desire to get Numeric into the Python standard library, but Guido van Rossum decided that the code was not maintainable in its state then.

In early 2005, NumPy developer Travis Oliphant wanted to unify the community around a single array package and ported Numarray's features to Numeric, releasing the result as NumPy 1.0 in 2006.[7] This new project was part of SciPy. To avoid installing the large SciPy package just to get an array object, this new package was separated and called NumPy. Support for Python 3 was added in 2011 with NumPy version 1.5.0.[13]

In 2011, PyPy started development on an implementation of the NumPy API for PyPy. It is not yet fully compatible with NumPy.

NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring rewriting some code, mostly inner loops using NumPy.

Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted,[16] and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars. In comparison, MATLAB boasts a large number of additional toolboxes, notably Simulink, whereas NumPy is intrinsically integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; SciPy is a library that adds more MATLAB-like functionality and Matplotlib is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

Python bindings of the widely used computer vision library OpenCV utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, slicing or masking with other arrays are very efficient ways to access specific pixels of an image. The NumPy array as universal data structure in OpenCV for images, extracted feature points, filter kernels and many more vastly simplifies the programming workflow and debugging.

Traits

NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring rewriting some code, mostly inner loops using NumPy.

Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted,[16] and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars. In comparison, MATLAB boasts a large number of additional toolboxes, notably Simulink, whereas NumPy is intrinsically

integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; SciPy is a library that adds more MATLAB-like functionality and Matplotlib is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

Python bindings of the widely used computer vision library OpenCV utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, slicing or masking with other arrays are very efficient ways to access specific pixels of an image. The NumPy array as universal data structure in OpenCV for images, extracted feature points, filter kernels and many more vastly simplifies the programming workflow and debugging.

The ndarray data structure

The core functionality of NumPy is its "ndarray", for n-dimensional array, data structure. These arrays are strided views on memory.[7] In contrast to Python's built-in list data structure (which, despite the name, is a dynamic array), these arrays are homogeneously typed: all elements of a single array must be of the same type.

Such arrays can also be views into memory buffers allocated by C/C++, Cython, and Fortran extensions to the CPython interpreter without the need to copy data around, giving a degree of compatibility with existing numerical libraries. This functionality is exploited by the SciPy package, which wraps a number of such libraries (notably BLAS and LAPACK). NumPy has built-in support for memory-mapped ndarrays.[7]

Limitations

Inserting or appending entries to an array is not as trivially possible as it is with Python's lists. The np.pad(...) routine to extend arrays actually creates new arrays of the desired shape and padding values, copies the given array into the new one and returns it. NumPy's np.concatenate([a1,a2]) operation does not actually link the two arrays but returns a new one, filled with the entries from both given arrays in sequence. Reshaping the dimensionality of an array with np.reshape(...) is only possible as long as the number of elements in the array does not change. These circumstances originate from the fact that NumPy's arrays must be views on contiguous memory buffers. A replacement package called Blaze attempts to overcome this limitation.[17]

Algorithms that are not expressible as a vectorized operation will typically run slowly because they must be implemented in "pure Python", while vectorization may increase memory complexity of some operations from constant to linear, because temporary arrays must be created that are as large as the inputs. Runtime compilation of numerical code has been implemented by

several groups to avoid these problems; open source solutions that interoperate with NumPy include scipy.weave, numexpr[18] and Numba. Cython and Pythran are static-compiling alternatives to these.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called axes.

For example, the coordinates of a point in 3D space [1, 2, 1] has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

[[1., 0., 0.],
[0., 1., 2.]]

NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

ndarray.ndim:

the number of axes (dimensions) of the array.

ndarray.shape:

the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with nrows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.

ndarray.size

the total number of elements of the array. This is equal to the product of the elements of shape.

ndarray.dtype

an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize

the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray.data

the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>>a.shape
(3, 5)
>>>a.ndim
2
>>>a.dtype.name
'int64'
>>>a.itemsize
8
>>>a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>>
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>>a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>>b.dtype
```

```
dtype('float64')
```

A frequent error consists in calling array with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>>
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>>
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>>
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array isfloat64.

```
>>>
>>>np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>>np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
```

```
       [ 1, 1, 1, 1]]], dtype=int16)
>>>np.empty( (2,3) )                    # uninitialized, output may vary
array([[ 3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
[ 5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```
>>>
>>>np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>>np.arange( 0, 2, 0.3 )              # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
>>>
>>> from numpy import pi
>>>np.linspace( 0, 2, 9 )               # 9 numbers from 0 to 2
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )     # useful to evaluate function at lots of
points
>>> f = np.sin(x)
```

5.2 PANDAS:

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.

Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis / manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, **Series** (1-dimensional) and **DataFrame** (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, **DataFrame** provides everything that R's data.frame provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let *Series*, *DataFrame*, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format

- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes :

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cythoncode. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

5.3 DATA STRUCTURES

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

Why more than one data structure?

The best way to think about the pandas data structures is as flexible containers for lower dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using ndarrays to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing

functions; axes are considered more or less equivalent (except when C- or Fortran-contiguousness matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set there is likely to be a "right" way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the DataFrame thus results in more readable code:

for col **in**df.columns:

series = df[col]

 # do something with series

Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to **favor immutability** where sensible.

Intro to Data Structures

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

In [1]: importnumpyasnp

In [2]: importpandasaspd

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

>>>s = pd.Series(data, index=index)

Here, data can be many different things:

- a Python dict
- anndarray
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data is**:

From ndarray

If data is anndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values [0, ..., len(data) - 1].

In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [4]: s

Out[4]:

a 0.469112

b -0.282863

c -1.509059

d -1.135632

e 1.212112

dtype: float64

In [5]: s.index

Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))

Out[6]:

0 -0.173215

1 0.119209

2 -1.044236

3 -0.861849

4 -2.104569

dtype: float64

Note

pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

In [7]: d = {'b': 1, 'a': 0, 'c': 2}

In [8]: pd.Series(d)

Out[8]:

b 1

a 0

c 2

dtype: int64

Note

When the data is a dict, and an index is not passed, the Series index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and Pandas version >= 0.23.

If you're using Python < 3.6 or Pandas < 0.23, and an index is not passed, the Series index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a Pandas version lower than 0.23, the Series would be ordered by the lexical order of the dict keys (i.e. ['a', 'b', 'c'] rather than ['b', 'a', 'c']).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

In [9]: d = {'a': 0., 'b': 1., 'c': 2.}

In [10]: pd.Series(d)

Out[10]:

a 0.0

b 1.0

c 2.0

dtype: float64

In [11]: pd.Series(d, index=['b', 'c', 'd', 'a'])

Out[11]:

b 1.0

c 2.0

d NaN

a 0.0

dtype: float64

Note

NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

In [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])

Out[12]:

a 5.0

b 5.0

c 5.0

d 5.0

e 5.0

dtype: float64

Series is ndarray-like

Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

In [13]: s[0]

Out[13]: 0.46911229990718628

In [14]: s[:3]

Out[14]:

a 0.469112

b -0.282863

c -1.509059

dtype: float64

In [15]: s[s >s.median()]

Out[15]:

a 0.469112

e 1.212112

dtype: float64

In [16]: s[[4, 3, 1]]

Out[16]:

e 1.212112

d -1.135632

b -0.282863

dtype: float64

In [17]: np.exp(s)

Out[17]:

a 1.598575

b 0.753623

c 0.221118

d 0.321219

e 3.360575

dtype: float64

Note

We will address array-based indexing like s[[4, 3, 1]] in section.

Like a NumPy array, a pandas Series has a **dtype**.

In [18]: s.dtype

Out[18]: dtype('float64')

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy's type system in a few places, in which case the dtype would be a **ExtensionDtype**. Some examples within pandas are Categorical Data and Nullable Integer Data Type. See dtypes for more.

If you need the actual array backing a Series, use **Series.array**.

In [19]: s.array

Out[19]:

<PandasArray>

[0.46911229990718628, -0.28286334432866328, -1.5090585031735124,

-1.1356323710171934, 1.2121120250208506]

Length: 5, dtype: float64

Accessing the array can be useful when you need to do some operation without the index (to disable automatic alignment, for example).

Series.array will always be an **ExtensionArray**. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a **numpy.ndarray**. Pandas knows how to take an ExtensionArray and store it in a Series or a column of a DataFrame. See dtypes for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use **Series.to_numpy()**.

In [20]: s.to_numpy()

Out[20]: array([0.4691, -0.2829, -1.5091, -1.1356, 1.2121])

Even if the Series is backed by a **ExtensionArray**, **Series.to_numpy()** will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

In [21]: s['a']

Out[21]: 0.46911229990718628

In [22]: s['e'] = 12.

In [23]: s

Out[23]:

a 0.469112

b -0.282863

c -1.509059

d -1.135632

e 12.000000

dtype: float64

In [24]: 'e' **in** s

Out[24]: True

In [25]: 'f' **in** s

Out[25]: False

If a label is not contained, an exception is raised:

>>>s['f']

KeyError: 'f'

Using the get method, a missing label will return None or specified default:

In [26]: s.get('f')

In [27]: s.get('f', np.nan)

Out[27]: nan

See also the section on attribute access.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting anndarray.

In [28]: s + s

Out[28]:

a 0.938225

b -0.565727

c -3.018117

d -2.271265

e 24.000000

dtype: float64

In [29]: s * 2

Out[29]:

a 0.938225

b -0.565727

c -3.018117

d -2.271265

e 24.000000

dtype: float64

In [30]: np.exp(s)

Out[30]:

a 1.598575

b 0.753623

c 0.221118

d 0.321219

e 162754.791419

dtype: float64

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

In [31]: s[1:] + s[:-1]

Out[31]:

a NaN

b -0.565727

c -3.018117

d -2.271265

e NaN

dtype: float64

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note

In general, we chose to make the default result of operations between differently indexed objects yield the**union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the **dropna** function.

Name attribute

Series can also have a name attribute:

In [32]: s = pd.Series(np.random.randn(5), name='something')

In [33]: s

Out[33]:

0 -0.494929

1 1.071804

2 0.721555

3 -0.706771

4 -1.039575

Name: something, dtype: float64

In [34]: s.name

Out[34]: 'something'

The Series name will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

New in version 0.18.0.

You can rename a Series with the **pandas.Series.rename()** method.

In [35]: s2 = s.rename("different")

In [36]: s2.name

Out[36]: 'different'

Note that s and s2 refer to different objects.

5.4 DataFrame

DataFrame is a 2-dimensional 45 abelled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note

When the data is a dict, and columns is not specified, the DataFrame columns will be ordered by the dict's insertion order, if you are using Python version >= 3.6 and Pandas >= 0.23.

If you are using Python < 3.6 or Pandas < 0.23, and columns is not specified, the DataFrame columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

In [37]: d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),

….: 'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}

….:

In [38]: df = pd.DataFrame(d)

In [39]: df

Out[39]:

one two

a 1.0 1.0

b 2.0 2.0

c 3.0 3.0

d NaN 4.0

In [40]: pd.DataFrame(d, index=[‘d’, ‘b’, ‘a’])

Out[40]:

one two

d NaN 4.0

b 2.0 2.0

a 1.0 1.0

In [41]: pd.DataFrame(d, index=[‘d’, ‘b’, ‘a’], columns=[‘two’, ‘three’])

Out[41]:

two three

d 4.0NaN

b 2.0NaN

a 1.0NaN

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

Note

When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

In [42]: df.index

Out[42]: Index([‘a’, ‘b’, ‘c’, ‘d’], dtype=’object’)

In [43]: df.columns

Out[43]: Index([‘one’, ‘two’], dtype=’object’)

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be range(n), where n is the array length.

In [44]: d = {‘one’: [1., 2., 3., 4.],

….: ‘two’: [4., 3., 2., 1.]}

….:

In [45]: pd.DataFrame(d)

Out[45]:

one two

2 1.0 4.0
3 2.0 3.0

2 3.0 2.0

3 4.0 1.0

In [46]: pd.DataFrame(d, index=[‘a’, ‘b’, ‘c’, ‘d’])

Out[46]:

one two

a 1.0 4.0

b 2.0 3.0

c 3.0 2.0

d 4.0 1.0

From structured or record array

This case is handled identically to a dict of arrays.

In [47]: data = np.zeros((2,), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

In [48]: data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

In [49]: pd.DataFrame(data)

Out[49]:

A B C

4 1 2.0b'Hello'
5 2 3.0b'World'

In [50]: pd.DataFrame(data, index=['first', 'second'])

Out[50]:

A B C

first 1 2.0b'Hello'

second 2 3.0 b'World'

In [51]: pd.DataFrame(data, columns=['C', 'A', 'B'])

Out[51]:

C A B

6 b’Hello’ 1 2.0
7 b’World’ 2 3.0

Note

DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

From a list of dicts

In [52]: data2 = [{‘a’: 1, ‘b’: 2}, {‘a’: 5, ‘b’: 10, ‘c’: 20}]

In [53]: pd.DataFrame(data2)

Out[53]:

a b c

0 1 2 NaN

8 5 10 20.0

In [54]: pd.DataFrame(data2, index=[‘first’, ‘second’])

Out[54]:

a b c

first 1 2 NaN

second 5 10 20.0

In [55]: pd.DataFrame(data2, columns=[‘a’, ‘b’])

Out[55]:

a b

0 1 2

9 5 10

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

In [56]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},

….: ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},

….: ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},

….: ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},

….: ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})

….:

Out[56]:

a b

b a c a b

A B 1.0 4.0 5.0 8.0 10.0

C 2.0 3.0 6.0 7.0 NaN

D NaNNaNNaNNaN 9.0

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Missing Data

Much more will be said on this topic in the Missing data section. To construct a DataFrame with missing data, we use np.nan to represent missing values. Alternatively, you may pass a numpy.MaskedArray as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate Constructors

DataFrame.from_dict

DataFrame.from_dict takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the DataFrame constructor except for the orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

In [57]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]))

Out[57]:

A B

10 1 4
11 2 5

2 3 6

If you pass orient='index', the keys will be the row labels. In this case, you can also pass the desired column names:

In [58]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]),

….: orient='index', columns=['one', 'two', 'three'])

….:

Out[58]: one two three

A 1 2 3

B 4 5 6

DataFrame.from_records

DataFrame.from_records takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal DataFrame constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

In [59]: data

Out[59]:

array([(1, 2., b'Hello'), (2, 3., b'World')],

dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [60]: pd.DataFrame.from_records(data, index='C')

Out[60]:

	A	BC
b'Hello'	1	2.0
b'World'	2	3.0

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

In [61]: df['one']

Out[61]:

a 1.0

b 2.0

c 3.0

d NaN

Name: one, dtype: float64

In [62]: df[‘three’] = df[‘one’] * df[‘two’]

In [63]: df[‘flag’] = df[‘one’] > 2

In [64]: df

Out[64]:

one two three flag

a 1.0 1.0 1.0 False

b 2.0 2.0 4.0 False

c 3.0 3.0 9.0 True

d NaN 4.0 NaN False

Columns can be deleted or popped like with a dict:

In [65]: deldf[‘two’]

In [66]: three = df.pop(‘three’)

In [67]: df

Out[67]:

one flag

a 1.0 False

b 2.0 False

c 3.0 True

d NaN False

When inserting a scalar value, it will naturally be propagated to fill the column:

In [68]: df[‘foo’] = ‘bar’

In [69]: df

Out[69]:

one flag foo

a 1.0 False bar

b 2.0 False bar

c 3.0 True bar

d NaN False bar

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:

In [70]: df[‘one_trunc’] = df[‘one’][:2]

In [71]: df

Out[71]:

one flag fooone_trunc

a 1.0 False bar 1.0

b 2.0 False bar 2.0

c 3.0 True bar NaN

d NaN False bar NaN

You can insert raw ndarrays but their length must match the length of the DataFrame’s index.

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns:

In [72]: df.insert(1, 'bar', df['one'])

In [73]: df

Out[73]:

one bar flag foo one_trunc

a 1.0 1.0 False bar 1.0

b 2.0 2.0 False bar 2.0

c 3.0 3.0 True bar NaN

d NaNNaN False bar NaN

Assigning New Columns in Method Chains

Inspired by dplyr's mutate verb, DataFrame has an **assign()** method that allows you to easily create new columns that are potentially derived from existing columns.

In [74]: iris = pd.read_csv('data/iris.data')

In [75]: iris.head()

Out[75]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa

4 5.0 3.6 1.4 0.2 Iris-setosa

In [76]: (iris.assign(sepal_ratio=iris[‘SepalWidth’] / iris[‘SepalLength’])

….: .head())

….:

Out[76]:

SepalLengthSepalWidthPetalLengthPetalWidth Name sepal_ratio

0 5.1 3.5 1.4 0.2 Iris-setosa 0.686275

1 4.9 3.0 1.4 0.2 Iris-setosa 0.612245

2 4.7 3.2 1.3 0.2 Iris-setosa 0.680851

3 4.6 3.1 1.5 0.2 Iris-setosa 0.673913

4 5.0 3.6 1.4 0.2 Iris-setosa 0.720000

In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

In [77]: iris.assign(sepal_ratio=**lambda** x: (x[‘SepalWidth’] / x[‘SepalLength’])).head()

Out[77]:

SepalLengthSepalWidthPetalLengthPetalWidth Name sepal_ratio

0 5.1 3.5 1.4 0.2 Iris-setosa 0.686275

1 4.9 3.0 1.4 0.2 Iris-setosa 0.612245

2 4.7 3.2 1.3 0.2 Iris-setosa 0.680851

3 4.6 3.1 1.5 0.2 Iris-setosa 0.673913

4 5.0 3.6 1.4 0.2 Iris-setosa 0.720000

assign **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand. This is common when using assign in a chain of operations. For example, we can limit the DataFrameto just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

In [78]: (iris.query('SepalLength> 5')

….: .assign(SepalRatio=**lambda** x: x.SepalWidth / x.SepalLength,

….: PetalRatio=**lambda** x: x.PetalWidth / x.PetalLength)

….: .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))

….:

Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2b527b1a58>

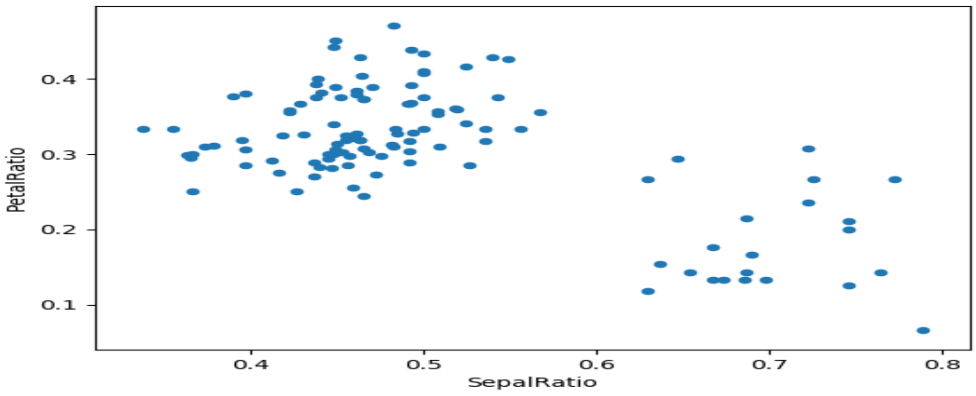


Fig 6 Matplot Axes

Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFramethat's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for assign is simply **kwargs. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a Series or NumPy array), or a function of one argument to be called on the DataFrame. A *copy* of the original DataFrame is returned, with the new values inserted.

Changed in version 0.23.0.

Starting with Python 3.6 the order of **kwargs is preserved. This allows for *dependent* assignment, where an expression later in **kwargs can refer to a column created earlier in the same **assign()**.

In [79]: dfa = pd.DataFrame({"A": [1, 2, 3],

….: "B": [4, 5, 6]})

….:

In [80]: dfa.assign(C=**lambda** x: x['A'] + x['B'],

….: D=**lambda** x: x['A'] + x['C'])

….:

Out[80]:

A B C D

0 1 4 5 6

1 2 5 7 9

2 3 6 9 12

In the second expression, x['C'] will refer to the newly created column, that's equal to dfa['A'] + dfa['B'].

To write code compatible with all versions of Python, split the assignment in two.

In [81]: dependent = pd.DataFrame({“A”: [1, 1, 1]})

In [82]: (dependent.assign(A=**lambda** x: x[‘A’] + 1)

….: .assign(B=**lambda** x: x[‘A’] + 2))

….:

Out[82]:

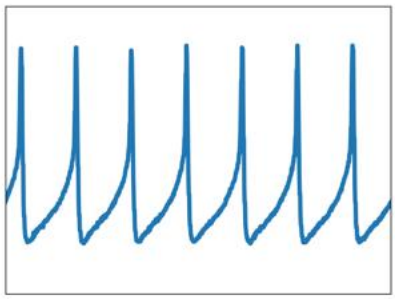
A B

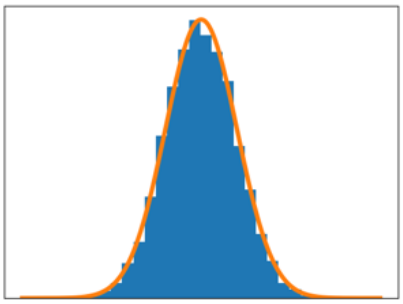
12 2 4
13 2 4

2 2 4

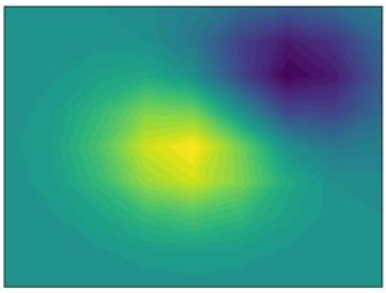
5.4MATPLOTLIB:

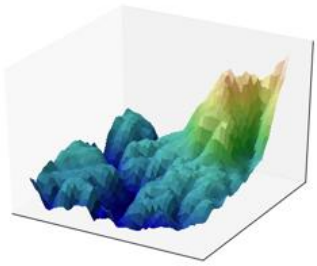
Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.





OBJ





OBJ

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Installing an official release

Matplotlib and its dependencies are available as wheel packages for macOS, Windows and Linux distributions:

python -m pip install -U pip
python -m pip install -U matplotlib

Although not required, we suggest also installing IPython for interactive use. To easily install a complete Scientific Python stack, see Scientific Python Distributions below.

macOS

To use the native OSX backend you will need a framework build build of Python.

Test data

The wheels (*.whl) on the PyPI download page do not contain test data or example code.

If you want to try the many demos that come in the Matplotlib source distribution, download the *.tar.gz file and look in the examples subdirectory.

To run the test suite:

- extract the lib/matplotlib/tests or lib/mpl_toolkits/tests directories from the source distribution;
- install test dependencies: pytest, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, ImageMagick, and Inkscape;
- run python -mpytest.

Third-party distributions of Matplotlib:
Scientific Python Distributions:

Anaconda and Canopy and ActiveState are excellent choices that "just work" out of the box for Windows, macOS and common Linux platforms. WinPython is an option for Windows users. All of these distributions include Matplotlib and *lots* of other useful (data) science tools.

Linux: using your package manager

If you are on Linux, you might prefer to use your package manager. Matplotlib is packaged for almost every major Linux distribution.

- Debian / Ubuntu: sudo apt-get install python3-matplotlib
- Fedora: sudo dnf install python3-matplotlib
- Red Hat: sudo yum install python3-matplotlib
- Arch: sudo pacman -S python-matplotlib

Installing from source

If you are interested in contributing to Matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build Matplotlib from source. Grab the latest *tar.gz* release file from the PyPI files page, or if you want to develop Matplotlib or just need the latest bugfixed version, grab the latest git version Install from source.

The standard environment variables CC, CXX, PKG_CONFIG are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

```
export CC=x86_64-pc-linux-gnu-gcc
export CXX=x86_64-pc-linux-gnu-g++
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

Once you have satisfied the requirements detailed below (mainly Python, NumPy, libpng and FreeType), you can build Matplotlib.

```
cd matplotlib
python -mpipinstall .
```

We provide a setup.cfg file which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that Matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging Matplotlib.

If you have installed prerequisites to nonstandard places and need to inform Matplotlib where they are, edit setupext.py and add the base dirs to the basedir dictionary entry for your sys.platform; e.g., if the header of some required library is in /some/path/include/someheader.h, put /some/path in the basedir list for your platform.

Dependencies

Matplotlib requires the following dependencies:

- Python (>= 3.5)
- FreeType (>= 2.3)
- libpng (>= 1.2)
- NumPy (>= 1.10.0)
- setuptools
- cycler (>= 0.10.0)
- dateutil (>= 2.1)
- kiwisolver (>= 1.0.0)
- pyparsing

Optionally, you can also install a number of packages to enable better user interface toolkits. See What is a backend? for more details on the optional Matplotlib backends and the capabilities they provide.

- tk (>= 8.3, != 8.6.0 or 8.6.1): for the Tk-based backends;
- PyQt4 (>= 4.6) or PySide (>= 1.0.3): for the Qt4-based backends;
- PyQt5: for the Qt5-based backends;
- PyGObject or pgi (>= 0.0.11.2): for the GTK3-based backends;
- wxpython (>= 4): for the WX-based backends;
- cairocffi (>= 0.8) or pycairo: for the cairo-based backends;
- Tornado: for the WebAgg backend;

For better support of animation output format and image file formats, LaTeX, etc., you can install the following:

- ffmpeg/avconv: for saving movies;
- ImageMagick: for saving animated gifs;

- Pillow (>= 3.4): for a larger selection of image file formats: JPEG, BMP, and TIFF image files;
- LaTeX and GhostScript (>=9.0) : for rendering text with LaTeX.

Building on Linux

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build Matplotlib with:

sudo apt-get build-dep python-matplotlib

If you are on Fedora, you can get all the dependencies required to build Matplotlib with:

sudodnfbuilddep python-matplotlib

If you are on RedHat, you can get all the dependencies required to build Matplotlib by first installing yum-builddep and then running:

su -c "yum-builddep python-matplotlib"

These commands do not build Matplotlib, but instead get and install the build dependencies, which will make building from source easier.

Building on macOS

The build situation on macOS is complicated by the various places one can get the libpng and FreeType requirements (MacPorts, Fink, /usr/X11R6), the different architectures (e.g., x86, ppc, universal), and the different macOS versions (e.g., 10.4 and 10.5). We recommend that you build the way we do for the macOS release: get the source from the tarball or the git repository and install the required dependencies through a third-party package manager. Two widely used package managers are Homebrew, and MacPorts. The following example illustrates how to install libpng and FreeType using brew:

brew install libpngfreetypepkg-config

If you are using MacPorts, execute the following instead:

port install libpngfreetypepkgconfig

After installing the above requirements, install Matplotlib from source by executing:

python -mpipinstall .

Note that your environment is somewhat important. Some conda users have found that, to run the tests, their PYTHONPATH must include /path/to/anaconda/.../site-packages and their DYLD_FALLBACK_LIBRARY_PATH must include /path/to/anaconda/lib.

Building on Windows

The Python shipped from https://www.python.org is compiled with Visual Studio 2015 for 3.5+. Python extensions should be compiled with the same compiler, see e.g. https://packaging.python.org/guides/packaging-binary-extensions/#setting-up-a-build-environment-on-windows for how to set up a build environment.

Since there is no canonical Windows package manager, the methods for building FreeType, zlib, and libpng from source code are documented as a build script at matplotlib-winbuild.

There are a few possibilities to build Matplotlib on Windows:

- Wheels via matplotlib-winbuild
- Wheels by using conda packages (see below)
- Conda packages (see below)

Wheel builds using conda packages

This is a wheel build, but we use conda packages to get all the requirements. The binary requirements (png, FreeType,...) are statically linked and therefore not needed during the wheel install.

Set up the conda environment. Note, if you want a qt backend, add pyqt to the list of conda packages.

conda create -n "matplotlib_build" python=3.7 numpy python-dateutilpyparsing tornado cycler tklibpngzlibfreetypemsinttypes
conda activate matplotlib_build

For building, call the script build_alllocal.cmd in the root folder of the repository:

build_alllocal.cmd

General Concepts

matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., 'contour this 2-D array') to the most specific (e.g., 'color this screen pixel red'). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control -- that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib "state-machine environment" which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

Note

Pyplot's state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

For even more control -- which is essential for things like embedding matplotlib plots in GUI applications -- the pyplot level may be dropped completely, leaving a purely object-oriented approach.

 # sphinx_gallery_thumbnail_number = 3
importmatplotlib.pyplotasplt
importnumpyasnp

Parts of a Figure

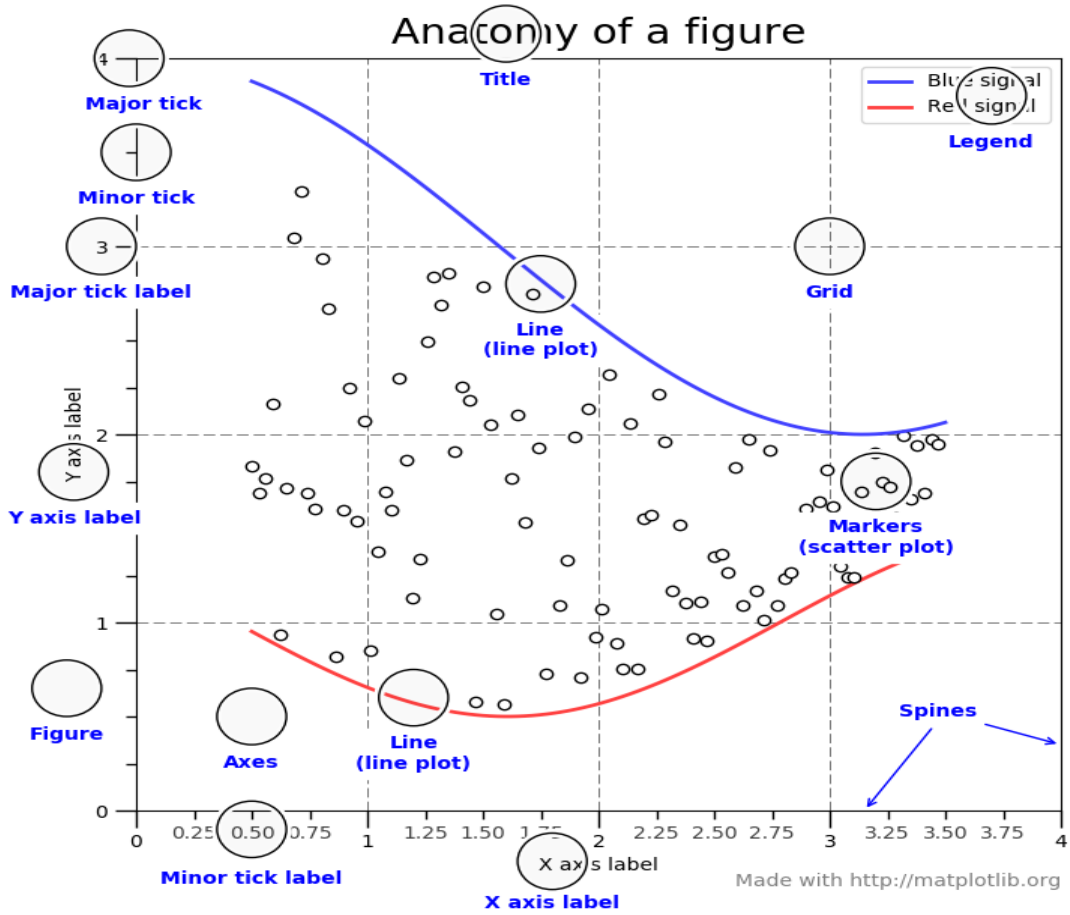


Fig 7 Anatomy of Matplotlib

The **whole** figure. The figure keeps track of all the child Axes, a smattering of 'special' artists (titles, figure legends, etc), and the **canvas**. (Don't worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of Axes, but to be useful should have at least one.

5.5 SEABORN:

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

An introduction to **seaborn**

Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with pandas data structures.

Here is some of the functionality that seaborn offers:

- A dataset-oriented API for examining relationships between multiple variables
- Specialized support for using categorical variables to show observations or aggregate statistics
- Options for visualizing univariate or bivariate distributions and for comparing them between subsets of data
- Automatic estimation and plotting of linear regression models for different kinds dependent variables
- Convenient views onto the overall structure of complex datasets
- High-level abstractions for structuring multi-plot grids that let you easily build complex visualizations
- Concise control over matplotlib figure styling with several built-in themes
- Tools for choosing color palettes that faithfully reveal patterns in your data

Seaborn aims to make visualization a central part of exploring and understanding data. Its dataset-oriented plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots.

Here's an example of what this means:

importseabornassns

```
sns.set()

tips = sns.load_dataset("tips")

sns.relplot(x="total_bill", y="tip", col="time",

        hue="smoker", style="smoker", size="size",

        data=tips);
```

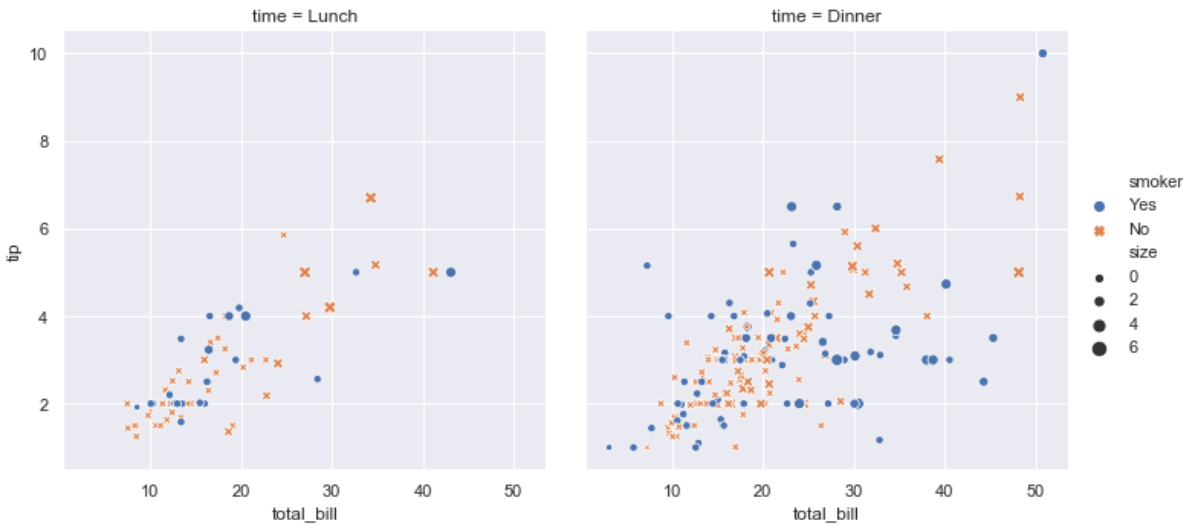


Fig 8 Seaborn plot

A few things have happened here. Let's go through them one by one:

1. We import seaborn, which is the only library necessary for this simple example.

importseabornassns

Behind the scenes, seaborn uses matplotlib to draw plots. Many tasks can be accomplished with only seaborn functions, but further customization might require using matplotlib directly. This is explained in more detail below. For interactive work, it's recommended to use a Jupyter/IPython interface in matplotlib mode, or else you'll have to call matplotlib.pyplot.show when you want to see the plot.

2. We apply the default default seaborn theme, scaling, and color palette.

sns.set()

This uses the matplotlib rcParam system and will affect how all matplotlib plots look, even if you don't make them with seaborn. Beyond the default theme, there are several other options, and you can independently control the style and scaling of the plot to quickly translate your work between presentation contexts (e.g., making a plot that will have readable fonts when projected during a talk). If you like the matplotlib defaults or prefer a different theme, you can skip this step and still use the seaborn plotting functions.

3. We load one of the example datasets.

tips = sns.load_dataset("tips")

Most code in the docs will use the **load_dataset()** function to get quick access to an example dataset. There's nothing particularly special about these datasets; they are just pandas dataframes, and we could have loaded them with pandas.read_csv or build them by hand. Many examples use the "tips" dataset, which is very boring but quite useful for demonstration. The tips dataset illustrates the "tidy" approach to organizing a dataset. You'll get the most out of seaborn if your datasets are organized this way, and it is explained in more detail below.

4. We draw a faceted scatter plot with multiple semantic variables.

```
sns.relplot(x="total_bill", y="tip", col="time",
        hue="smoker", style="smoker", size="size",
        data=tips)
```

This particular plot shows the relationship between five variables in the tips dataset. Three are numeric, and two are categorical. Two numeric variables (total_bill and tip) determined the position of each point on the axes, and the third (size) determined the size of each point. One categorical variable split the dataset onto two different axes (facets), and the other determined the color and shape of each point.

All of this was accomplished using a single call to the seaborn function **relplot()**. Notice how we only provided the names of the variables in the dataset and the roles that we wanted them to play in the plot. Unlike when using matplotlib directly, it wasn't necessary to translate the variables into parameters of the visualization (e.g., the specific color or marker to use for each category). That translation was done automatically by seaborn. This lets the user stay focused on the question they want the plot to answer.

API abstraction across visualizations

There is no universal best way to visualize data. Different questions are best answered by different kinds of visualizations. Seaborn tries to make it easy to switch between different visual representations that can be parameterized with the same dataset-oriented API.

The function **relplot()** is named that way because it is designed to visualize many different statistical *relationships*. While scatter plots are a highly effective way of doing this, relationships where one variable represents a measure of time are better represented by a line. The **relplot()** function has a convenient kind parameter to let you easily switch to this alternate representation:

```
dots = sns.load_dataset("dots")
sns.relplot(x="time", y="firing_rate", col="align",
            hue="choice", size="coherence", style="choice",
facet_kws=dict(sharex=False),
            kind="line", legend="full", data=dots);
```

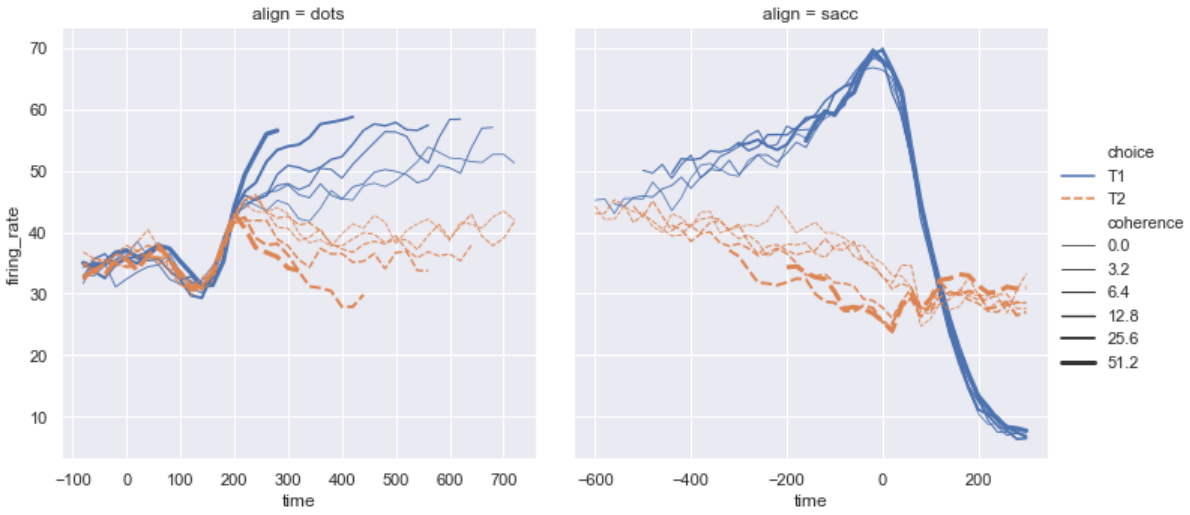


Fig 9 Seaborn Scatter Plot

Notice how the size and style parameters are shared across the scatter and line plots, but they affect the two visualizations differently (changing marker area and symbol vs line width and dashing). We did not need to keep those details in mind, letting us focus on the overall structure of the plot and the information we want it to convey.

Statistical estimation and error bars

Often we are interested in the average value of one variable as a function of other variables. Many seaborn functions can automatically perform the statistical estimation that is neccesary to answer these questions:

```
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal", col="region",
            hue="event", style="event",
            kind="line", data=fmri);
```

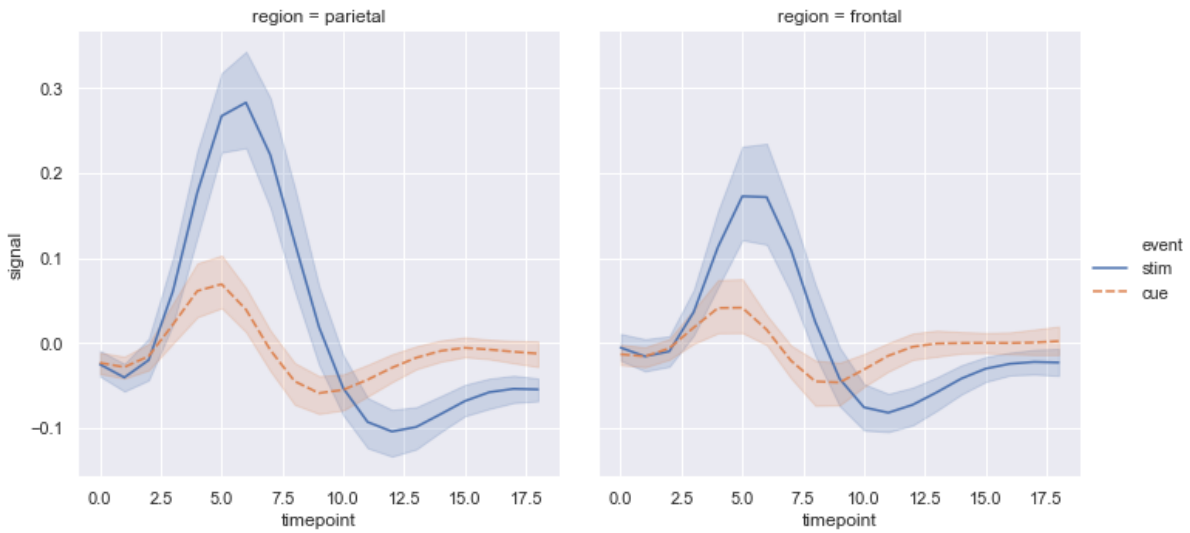


Fig 10 Implot

When statistical values are estimated, seaborn will use bootstrapping to compute confidence intervals and draw error bars representing the uncertainty of the estimate.

Statistical estimation in seaborn goes beyond descriptive statisitics. For example, it is also possible to enhance a scatterplot to include a linear regression model (and its uncertainty) using **lmplot()**:

```
sns.lmplot(x="total_bill", y="tip", col="time", hue="smoker",
        data=tips);
```

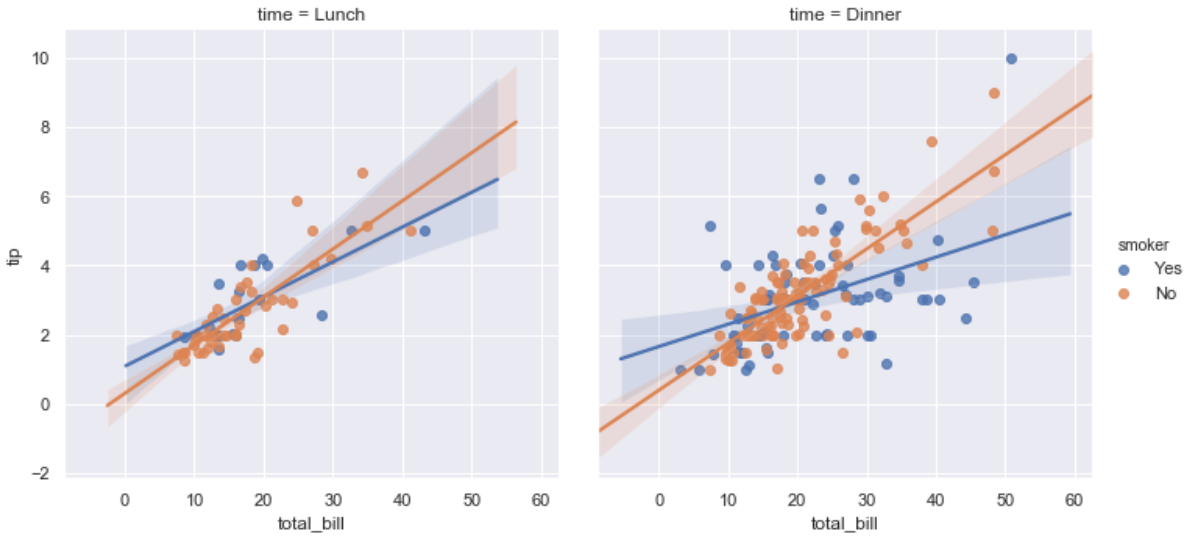


Fig 11 Specialized Categorical Plots

Specialized categorical plots

Standard scatter and line plots visualize relationships between numerical variables, but many data analyses involve categorical variables. There are

several specialized plot types in seaborn that are optimized for visualizing this kind of data. They can be accessed through **catplot()**. Similar to **relplot()**, the idea of **catplot()** is that it exposes a common dataset-oriented API that generalizes over different representations of the relationship between one numeric variable and one (or more) categorical variables.

These representations offer different levels of granularity in their presentation of the underlying data. At the finest level, you may wish to see every observation by drawing a scatter plot that adjusts the positions of the points along the categorical axis so that they don't overlap:

```
sns.catplot(x="day", y="total_bill", hue="smoker",
        kind="swarm", data=tips);
```

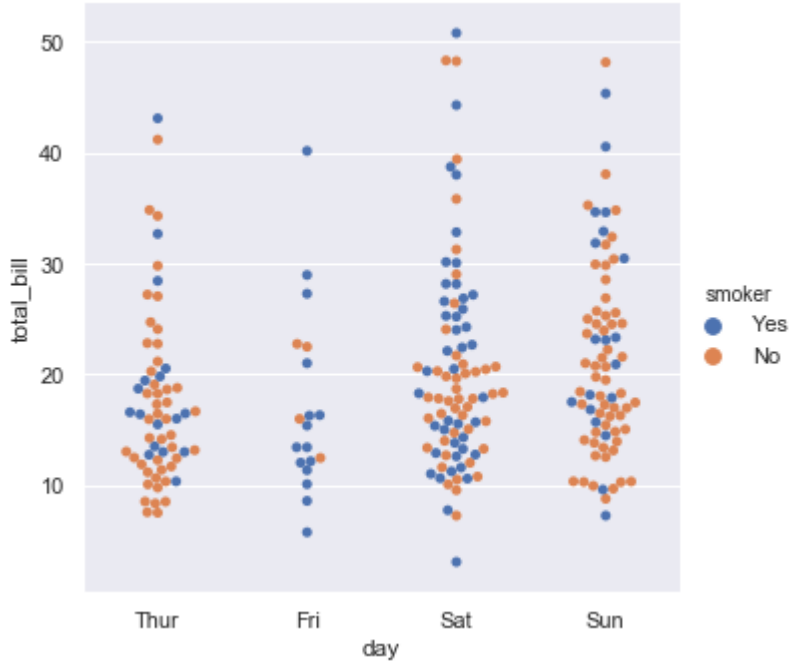


Fig 12 Kernal Density Plot

Alternately, you could use kernel density estimation to represent the underlying distribution that the points are sampled from:

```
sns.catplot(x="day", y="total_bill", hue="smoker",
        kind="violin", split=True, data=tips);
```

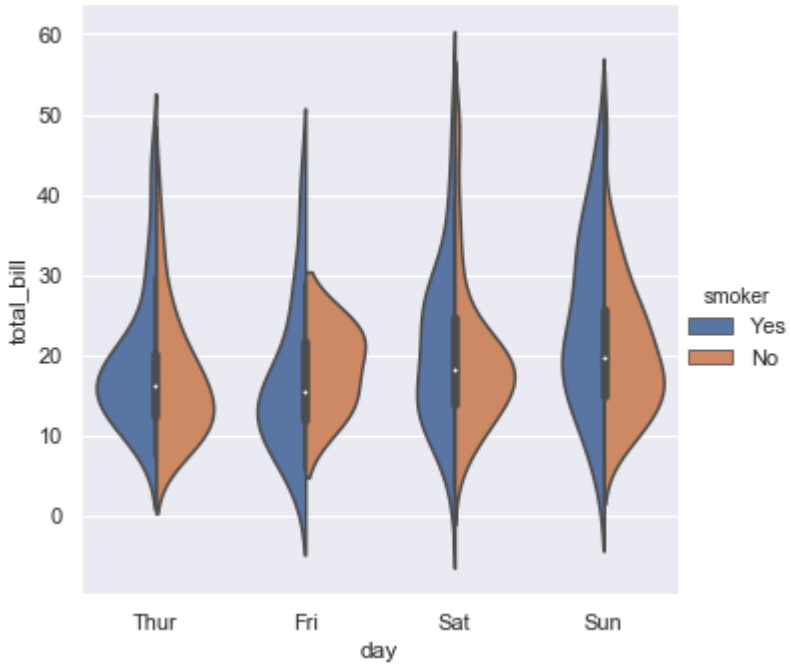


Fig 13 Mean Value plot

Or you could show the only mean value and its confidence interval within each nested category:

```
sns.catplot(x="day", y="total_bill", hue="smoker",
        kind="bar", data=tips);
```

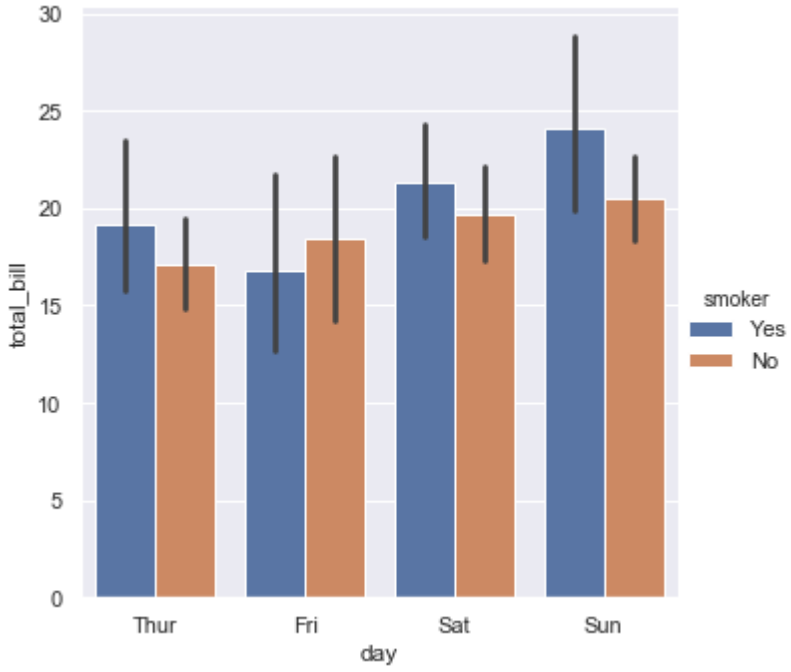


Fig 14 Bar plot

Figure-level and axes-level functions

How do these tools work? It's important to know about a major distinction between seaborn plotting functions. All of the plots shown so far have been made with "figure-level" functions. These are optimized for exploratory analysis because they set up the matplotlib figure containing the plot(s) and make it easy to spread out the visualization across multiple axes. They also handle some tricky business like putting the legend outside the axes. To do these things, they use a seaborn **FacetGrid**.

Each different figure-level plot kind combines a particular "axes-level" function with the **FacetGrid** object. For example, the scatter plots are drawn using the **scatterplot()** function, and the bar plots are drawn using the **barplot()** function. These functions are called "axes-level" because they draw onto a single matplotlib axes and don't otherwise affect the rest of the figure.

The upshot is that the figure-level function needs to control the figure it lives in, while axes-level functions can be combined into a more complex matplotlib figure with other axes that may or may not have seaborn plots on them:

importmatplotlib.pyplotasplt

f, axes = plt.subplots(1, 2, sharey=**True**, figsize=(6, 4))

sns.boxplot(x="day", y="tip", data=tips, ax=axes[0])

sns.scatterplot(x="total_bill", y="tip", hue="day", data=tips, ax=axes[1]);

Controling the size of the figure-level functions works a little bit differently than it does for other matplotlib figures. Instead of setting the overall figure size, the figure-level functions are parameterized by the size of each facet. And instead of setting the height and width of each facet, you control the height and *aspect* ratio (ratio of width to height). This parameterization makes it easy to control the size of the graphic without thinking about exactly how many rows and columns it will have, although it can be a source of confusion:

sns.relplot(x="time", y="firing_rate", col="align",

hue="choice", size="coherence", style="choice",

height=4.5, aspect=2 / 3,

facet_kws=dict(sharex=**False**),

kind="line", legend="full", data=dots);

The way you can tell whether a function is “figure-level” or “axes-level” is whether it takes an ax= parameter. You can also distinguish the two classes by their output type: axes-level functions return the matplotlib axes, while figure-level functions return the **FacetGrid**.

Visualizing dataset structure

There are two other kinds of figure-level functions in seaborn that can be used to make visualizations with multiple plots. They are each oriented towards illuminating the structure of a dataset. One, **jointplot()**, focuses on a single relationship:

```
iris = sns.load_dataset("iris")

sns.jointplot(x="sepal_length", y="petal_length", data=iris);
```

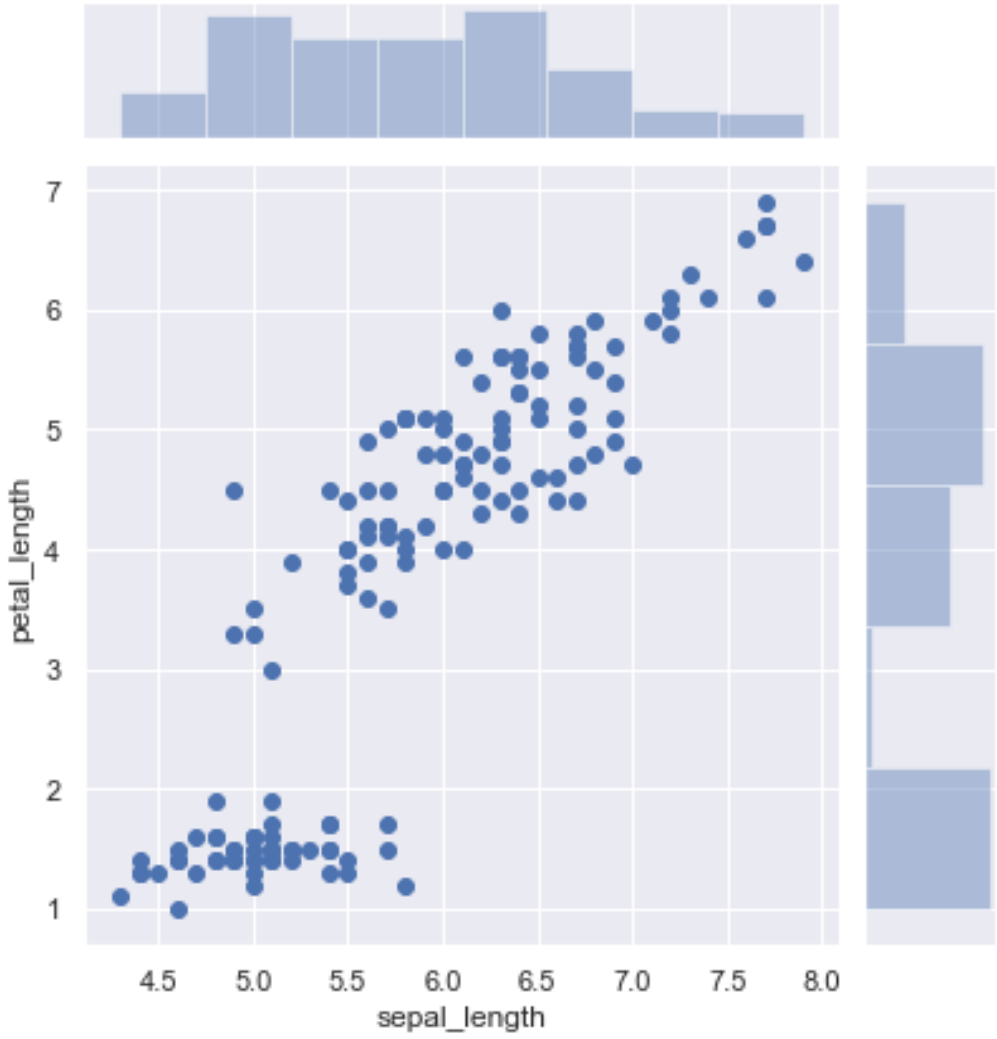


Fig 15 Visualizing dataset structure

The other, **pairplot()**, takes a broader view, showing all pairwise relationships and the marginal distributions, optionally conditioned on a categorical variable :

sns.pairplot(data=iris, hue="species");

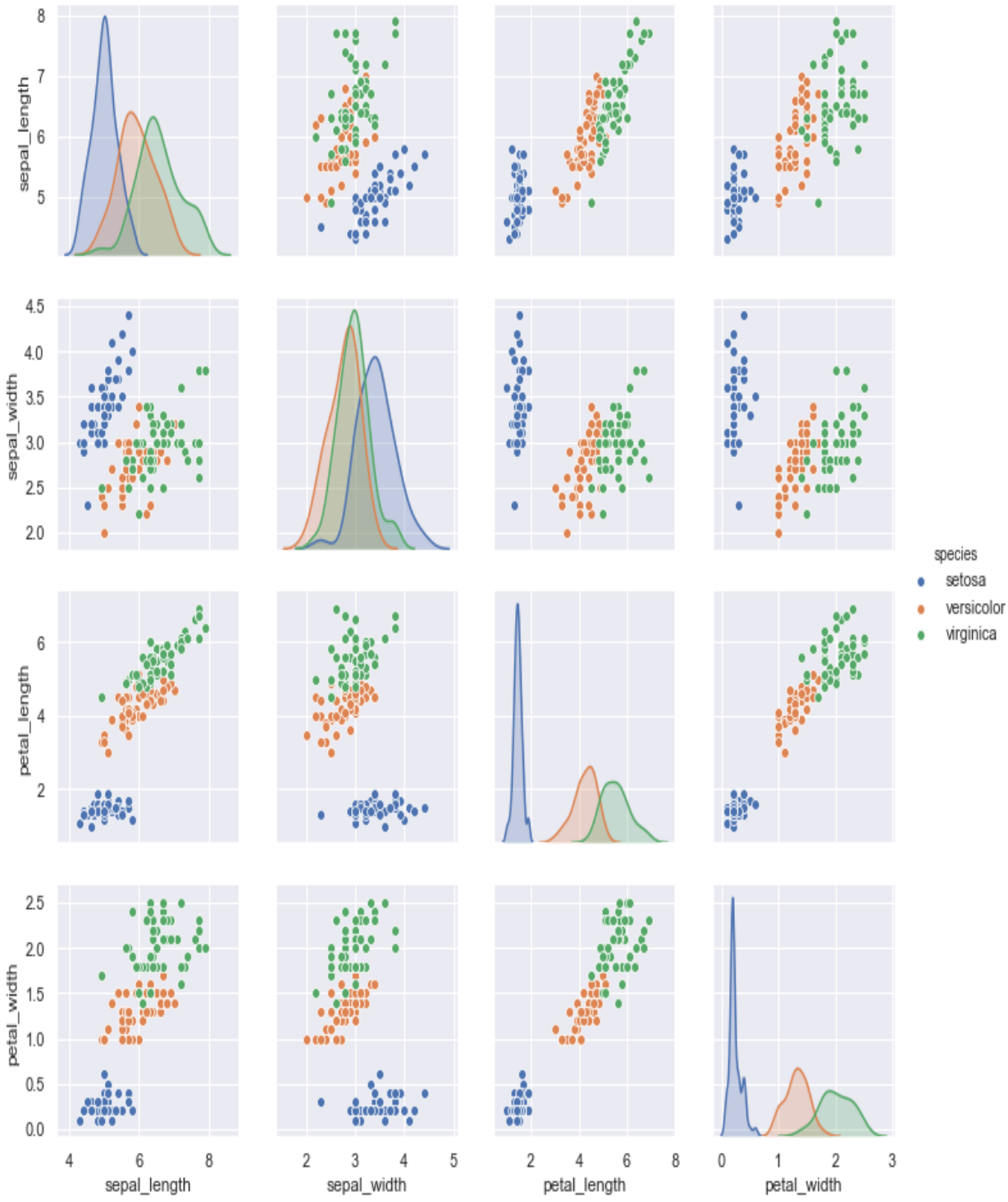


Fig 16 Pairplot

Both **jointplot()** and **pairplot()** have a few different options for visual representation, and they are built on top of classes that allow more thoroughly customized multi-plot figures (**JointGrid** and **PairGrid**, respectively).

Customizing plot appearance

The plotting functions try to use good default aesthetics and add informative labels so that their output is immediately useful. But defaults can only go so far, and creating a fully-polished custom plot will require additional steps. Several levels of additional customization are possible.

The first way is to use one of the alternate seaborn themes to give your plots a different look. Setting a different theme or color palette will make it take effect for all plots:

```
sns.set(style="ticks", palette="muted")

sns.relplot(x="total_bill", y="tip", col="time",
        hue="smoker", style="smoker", size="size",
        data=tips);
```

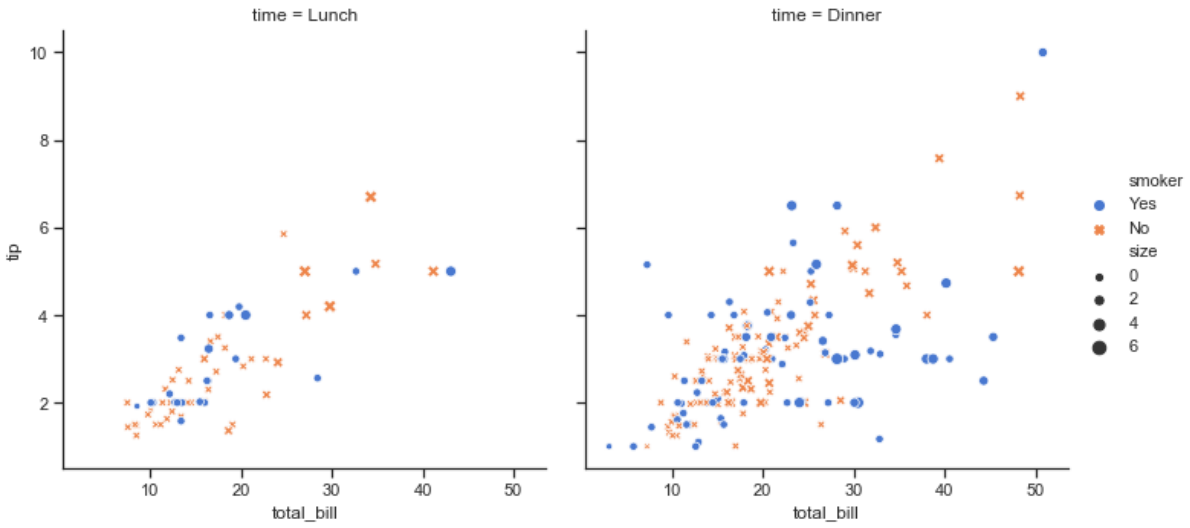


Fig 17 Smoker plot

For figure-specific customization, all seaborn functions accept a number of optional parameters for switching to non-default semantic mappings, such as different colors. (Appropriate use of color is critical for effective data visualization, and seaborn has extensive support for customizing color palettes).

Finally, where there is a direct correspondence with an underlying matplotlib function (like **scatterplot()** and plt.scatter), additional keyword arguments will be passed through to the matplotlib layer:

```
sns.relplot(x="total_bill", y="tip", col="time",
        hue="size", style="smoker", size="size",
        palette="YlGnBu", markers=["D", "o"], sizes=(10, 125),
edgecolor=".2", linewidth=.5, alpha=.75,
        data=tips);
```

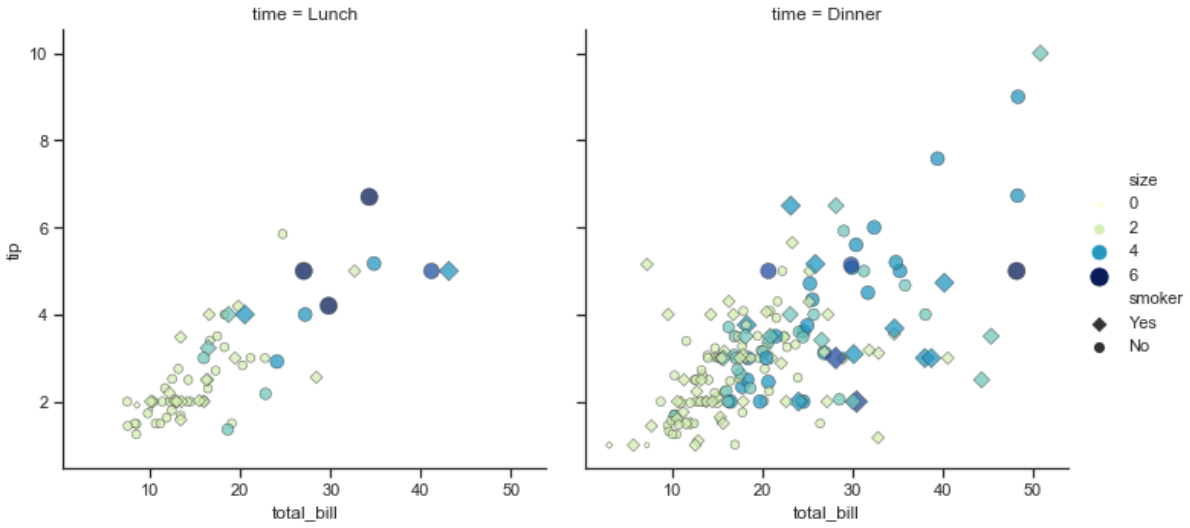


Fig 18 Scatter plot

In the case of **relplot()** and other figure-level functions, that means there are a few levels of indirection because **relplot()** passes its exta keyword arguments to the underlying seaborn axes-level function, which passes *its* extra keyword arguments to the underlying matplotlib function. So it might take some effort to find the right documentation for the parameters you'll need to use, but in principle an extremely high level of customization is possible.

Some customization of figure-level functions can be accomplished through additional parameters that get passed to **FacetGrid**, and you can use the methods on that object to control many other properties of the figure. For even more tweaking, you can access the matplotlib objects that the plot is drawn onto, which are stored as attributes:

```
g = sns.catplot(x="total_bill", y="day", hue="time",
        height=3.5, aspect=1.5,
        kind="box", legend=False, data=tips);
g.add_legend(title="Meal")
```

```
g.set_axis_labels("Total bill ($)", "")
g.set(xlim=(0, 60), yticklabels=["Thursday", "Friday", "Saturday", "Sunday"])
g.despine(trim=True)
g.fig.set_size_inches(6.5, 3.5)
g.ax.set_xticks([5, 15, 25, 35, 45, 55], minor=True);
plt.setp(g.ax.get_yticklabels(), rotation=30);
```

Because the figure-level functions are oriented towards efficient exploration, using them to manage a figure that you need to be precisely sized and organized may take more effort than setting up the figure directly in matplotlib and using the corresponding axes-level seaborn function. Matplotlib has a comprehensive and powerful API; just about any attribute of the figure can be changed to your liking. The hope is that a combination of seaborn's high-level interface and matplotlib's deep customizability will allow you to quickly explore your data and create graphics that can be tailored into a publication quality final product.

Organizing datasets

As mentioned above, seaborn will be most powerful when your datasets have a particular organization. This format ia alternately called "long-form" or "tidy" data and is described in detail by Hadley Wickham in this academic paper. The rules can be simply stated:

1. Each variable is a column
2. Each observation is a row

A helpful mindset for determining whether your data are tidy is to think backwards from the plot you want to draw. From this perspective, a "variable" is something that will be assigned a role in the plot. It may be useful to look at the example datasets and see how they are structured. For example, the first five rows of the "tips" dataset look like this:

```
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Table 1 Tips Dataset

In some domains, the tidy format might feel awkward at first. Timeseries data, for example, are sometimes stored with every timepoint as part of the same observational unit and appearing in the columns. The "fmri" dataset that we used above illustrates how a tidy timeseries dataset has each timepoint in a different row:

fmri.head()

	subject	timepoint	event	region	signal
0	s13	18	stim	parietal	-0.017552
1	s5	14	stim	parietal	-0.080883
2	s12	18	stim	parietal	-0.081033
3	s11	18	stim	parietal	-0.046134
4	s10	18	stim	parietal	-0.037970

Table 2 FRMI Dataset

Many seaborn functions can plot wide-form data, but only with limited functionality. To take advantage of the features that depend on tidy-formatted data, you'll likely find the pandas.melt function useful for "un-pivoting" a wide-form dataframe.

5.6 SCIKIT-LEARN

Defining scikit learn, it is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Scikit-learn was initially developed by David Cornopean as a Google summer of code project in 2007.Later Matthieu Brucher joined the project and started to use it as a part of his thesis work. In 2010 INRIA got involved and the first public release (v0.1 beta) was published in late January 2010.The project now has more than 30 active contributors and has had paid sponsorship from INRIA, Google, Tiny clues and the Python Software Foundation.

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), it is said to have several attributes or features.

Learning problems fall into a few categories:

- supervised learning, in which the data comes with additional attributes that we want to predict (Click here to go to the scikit-learn supervised learning page).This problem can be either:
    - classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of a classification problem would be handwritten digit recognition, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
    - regression: if the desired output consists of one or more continuous variables, then the task is called regression. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
- unsupervised learning, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of visualization (Click here to go to the Scikit-Learn unsupervised learning page).

Training set and testing set

Machine learning is about learning some properties of a data set and then testing those properties against another data set. A common practice in machine learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the training set, on which we learn some properties; we call the other set the testing set, on which we test the learned properties.

Loading an example dataset

scikit-learn comes with a few standard datasets, for instance the iris and digits datasets for classification and the boston house prices dataset for regression.

In the following, we start a Python interpreter from our shell and then load the iris and digits datasets. Our notational convention is that $ denotes the shell prompt while >>> denotes the Python interpreter prompt:

```
$ python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some metadata about the data. This data is stored in the .data member, which is a n_samples, n_features array. In the case of supervised problem, one or more response variables are stored in the .target member. More details on the different datasets can be found in the dedicated section.
For instance, in the case of the digits dataset, digits.data gives access to the features that can be used to classify the digits samples:

```
>>>
>>> print(digits.data)
[[ 0.   0.   5. ...   0.   0.   0.]
 [ 0.   0.   0. ...  10.   0.   0.]
 [ 0.   0.   0. ...  16.   9.   0.]
 ...
 [ 0.   0.   1. ...   6.   0.   0.]
 [ 0.   0.   2. ...  12.   0.   0.]
 [ 0.   0.  10. ...  12.   1.   0.]]
```

and digits.target gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
>>>
>>>digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

Shape of the data arrays
The data is always a 2D array, shape (n_samples, n_features), although the original data may have had a different shape. In the case of the digits, each original sample is an image of shape (8, 8) and can be accessed using:

```
>>>
>>>digits.images[0]
array([[ 0.,  0.,  5.,  13.,  9.,  1.,  0.,  0.],
[  0.,  0.,  13.,  15.,  10.,  15.,  5.,  0.],
[  0.,  3.,  15.,  2.,  0.,  11.,  8.,  0.],
[  0.,  4.,  12.,  0.,  0.,  8.,  8.,  0.],
[  0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
[  0.,  4.,  11.,  0.,  1.,  12.,  7.,  0.],
[  0.,  2.,  14.,  5.,  10.,  12.,  0.,  0.],
[  0.,  0.,  6.,  13.,  10.,  0.,  0.,  0.]])
```

The simple example on this dataset illustrates how starting from the original problem one can shape the data for consumption in scikit-learn.

Loading from external datasets

To load from an external dataset, please refer to loading external datasets.
Learning and predicting
In the case of the digits dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the

digits zero through nine) on which we fit an estimator to be able to predict the classes to which unseen samples belong.
In scikit-learn, an estimator for classification is a Python object that implements the methods fit(X, y) and predict(T).
An example of an estimator is the class sklearn.svm.SVC, which implements support vector classification. The estimator's constructor takes as arguments the model's parameters.
For now, we will consider the estimator as a black box:

```
>>>
>>> from sklearn import svm
>>>clf = svm.SVC(gamma=0.001, C=100.)
```

Choosing the parameters of the model

In this example, we set the value of gamma manually. To find good values for these parameters, we can use tools such as grid search and cross validation.
The clf (for classifier) estimator instance is first fitted to the model; that is, it must learn from the model. This is done by passing our training set to the fit method. For the training set, we'll use all the images from our dataset, except for the last image, which we'll reserve for our predicting. We select the training set with the [:-1] Python syntax, which produces a new array that contains all but the last item from digits.data:

```
>>>
>>>clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
```

Now you can predict new values. In this case, you'll predict using the last image from digits.data. By predicting, you'll determine the image from the training set that best matches the last image.

```
>>>
>>>clf.predict(digits.data[-1:])
array([8])
```

The corresponding image is:

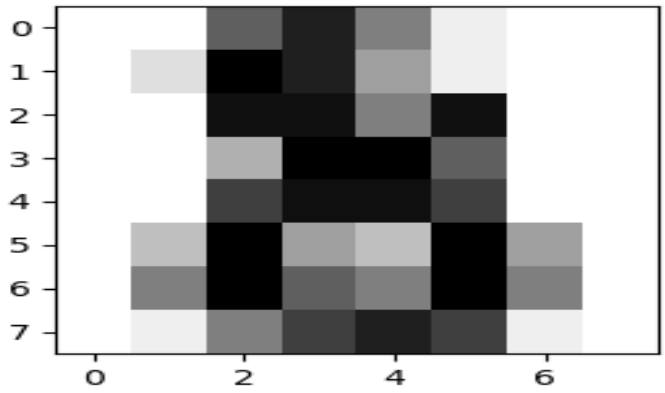


Fig 19 Training Set

Fig 20 Flask logo

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy.

```
from flask import Flask, escape, request


app = Flask(__name__)


@app.route('/')
def hello():
    name =request.args.get("name", "World")
return f'Hello, {escape(name)}!'
```

```
$ env FLASK_APP=hello.py flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.

Applications that use the Flask framework include Pinterest and LinkedIn.

CHAPTER 6

IMPLEMENTATION & RESULTS

Modules are files that contain Python definitions and declarations. Modules can define functions, classes, and variables. Modules can also include executable code. Grouping related code in the module can make it easier to understand and use the code. It also makes the code logical. In Python programming, treat modules as the same as code libraries.

The Designed system needs to be implemented. We are using Python Programming for the implementation. A chatbot is a computer program that allows people to interact with technology using various input methods (such as voice, text, gestures, and touch, 24 hours a day, 7 days a week, 365 days a year).

CHAPTER 7

CONCLUSION

CONCLUSION:

The Project is being implemented. The Proposed Book Recommender System will use Content based filtering technique using cosine similarity algorithm. This methodology depends on making a plenty of parameters to describe a particular product.. Thinking about an Book as an model the potential parameters could be Author, Publisher, Year Published etc.. The bigger the parameter set the better and simpler it is to coordinate examples with customer's profile and his online impression.

CHAPTER 8

REFERENCES

- https://iopscience.iop.org/article/10.1088/1742-6596/1362/1/012130/pdf
- https://webpages.charlotte.edu/nmatta1/cloudproject/Project_Report.pdf
- https://thesai.org/Downloads/Volume12No1/Paper_26Personalized_Book_Recommendation_System.pdf
- http://www.jotr.in/text.asp?20131/6/1/1/118718
- http://dlib.net/landmark_recommendationex.cpp.html
- http://ethesis.nitrkl.ac.in/80857/1/2016_BT_CSahoo_112EI0563_Driver.pdf
- https://www.ijitee.org/wp-content/uploads/papers/v8i6s4/F11640486S419.pdf

Engpaper Journal

https://www.engpaper.com