ABSTRACT

Fraud is one of the major moral issues in the credit card industry. The fundamental points are, right off the bat, to distinguish the distinctive kinds of Credit card smart, and, furthermore, to survey elective strategies that have been utilized in fraud recognition. It is critical that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase. The datasets contain exchanges made by charge cards in September 2013 by European cardholders. This dataset presents exchanges that happened in two days, where we have 492 frauds out of 284,807 exchanges. The dataset is profoundly unequal, the positive class (frauds) represent 0.172% all things considered. It contains just numerical input features which are the consequence of a PCA Transformation. Lamentably, because of secrecy issues, they can't give the original data and more background information about the data. Features V1, V2, ... V28 are the foremost parts gotten with PCA, the main features which have not been changed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between every transaction and the primary transaction in the dataset.

We are trying to compare two different algorithms, local outlier, and isolation forest to find the best result for getting the correct ratio of predicting positivity.

# INTRODUCTION

**Credit card fraud** is a wide-ranging term for theft and fraud committed using or involving a payment card, such as a credit card or debit card, as a fraudulent source of funds in a transaction.[1] The purpose may be to obtain goods without paying, or to obtain unauthorized funds from an account. Credit card fraud is also an adjunct to identity theft. According to the United States Federal Trade Commission, while the rate of identity theft had been holding steady during the mid 2000s, it increased by 21 percent in 2008. However, credit card fraud, that crime which most people associate with ID theft, decreased as a percentage of all ID theft complaints for the sixth year in a row.[2]

Although incidences of credit card fraud are limited to about 0.1% of all card transactions, they have resulted in huge financial losses as the fraudulent transactions have been large value transactions. In 1999, out of 12 billion transactions made annually, approximately 10 million—or one out of every 1200 transactions—turned out to be fraudulent.[3] Also, 0.04% (4 out of every 10,000) of all monthly active accounts were fraudulent. Even with tremendous volume and value increase in credit card transactions since then, these proportions have stayed the same or have decreased due to sophisticated fraud detection and prevention systems. Today's fraud detection systems are designed to prevent one-twelfth of one percent of all transactions processed which still translates into billions of dollars in losses.[3]

In the decade to 2008, general credit card losses have been 7 basis points or lower (i.e. losses of $0.07 or less per $100 of transactions).[4] In 2007, fraud in the United Kingdom was estimated at £535 million.

The fundamental points are, right off the bat, to distinguish the distinctive kinds of Credit card smart, and, furthermore, to survey elective strategies that have been utilized in fraud recognition.

# EXISTING SYSTEM

For quite a while, there has been a solid enthusiasm for the morals of banking (Molyneux, 2007; George, 1992), just as the ethical intricacy of fake conduct (Clarke, 1994). Extortion implies getting administrations/products and additionally cash by deceptive methods and is a developing issue everywhere throughout the world these days. Extortion manages cases including criminal purposes that, generally, are hard to recognize. Charge cards are a standout amongst the most well-known focuses of extortion however by all account not the only one; misrepresentation can happen with an acknowledge items, for example, individual advances, home credits, and retail. Besides, the essence of extortion has changed significantly amid the most recent couple of decades as advancements have changed and created. A basic assignment to support organizations and money related establishments including banks is to find a way to anticipate extortion and to manage it productively and adequately when it happens (Anderson, 2007).

Anderson (2007) has identified and explained the different types of fraud, which are as many and varied as the financial institution’s products and technologies.

# DISADVANTAGES

Fraud detection and prevention software that analyzes patterns of normal and unusual behavior as well as individual transactions in order to flag likely fraud. Profiles include such information as IP address. Technologies have existed since the early 1990s to detect potential fraud. One early market entrant was Falcon; other leading software solutions for card fraud include Actimize, SAS, BAE Systems Detica, and IBM. All these does not utilize the modern machine learning techniques which is far more efficient and more.

# PROPOSED SYSTEM

The main aims are, firstly, to identify the credit card fraud and, secondly, to review alternative techniques that have been used in fraud. Here we comparing two different algorithms each of different types of machine learning, one of supervised and another of unsupervised.

# ADVANTAGES

Machine Learning can review large volumes of data and discover specific trends and patterns that would not be apparent to humans. For instance, for an e-commerce website like Amazon, it serves to understand the browsing behaviors and purchase histories of its users to help cater to the right products, deals, and reminders relevant to them. It uses the results to reveal relevant advertisements to them.

ALGORITHMS:

K-MEANS:

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms.

Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes.

[AndreyBu](https://www.liveedu.tv/andreybu/REaxr-machine-learning-model-python-sklearn-kera/), who has more than 5 years of machine learning experience and currently teaches people his skills, says that “the objective of K-means is simple: group similar data points together and discover underlying patterns. To achieve this objective, K-means looks for a fixed number (k) of clusters in a dataset.”

A cluster refers to a collection of data points aggregated together because of certain similarities.

You’ll define a target number k, which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster.

Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares.

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

The ‘means’ in the K-means refers to averaging of the data; that is, finding the centroid.

# **How the K-means algorithm works**

To process the learning data, the K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids

It halts creating and optimizing clusters when either:

* The centroids have stabilized — there is no change in their values because the clustering has been successful.
* The defined number of iterations has been achieved.

# **K-means algorithm example problem**

Let’s see the steps on how the K-means machine learning algorithm works using the Python programming language.

We’ll use the Scikit-learn library and some random data to illustrate a K-means clustering simple explanation.

**Step 1: Import libraries**

import pandas as pdimport numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster import KMeans%matplotlib inline

As you can see from the above code, we’ll import the following libraries in our project:

* Pandas for reading and writing spreadsheets
* Numpy for carrying out efficient computations
* Matplotlib for visualization of data

**Step 2: Generate random data**

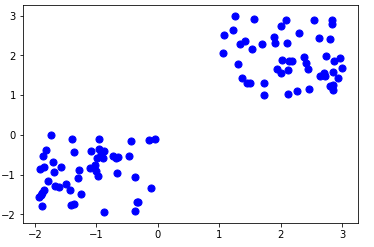
Here is the code for generating some random data in a two-dimensional space:

X= -2 \* np.random.rand(100,2)X1 = 1 + 2 \* np.random.rand(50,2)X[50:100, :] = X1plt.scatter(X[ : , 0], X[ :, 1], s = 50, c = ‘b’)plt.show()

A total of 100 data points has been generated and divided into two groups, of 50 points each.

Here is how the data is displayed on a two-dimensional space:





**Step 3: Use Scikit-Learn**

We’ll use some of the available functions in the [Scikit-learn library](http://scikit-learn.org/stable/) to process the randomly generated data.

Here is the code:

from sklearn.cluster import KMeansKmean = KMeans(n\_clusters=2)Kmean.fit(X)

In this case, we arbitrarily gave k (n\_clusters) an arbitrary value of two.

Here is the output of the K-means parameters we get if we run the code:

KMeans(algorithm=’auto’, copy\_x=True, init=’k-means++’, max\_iter=300  
 n\_clusters=2, n\_init=10, n\_jobs=1, precompute\_distances=’auto’,  
 random\_state=None, tol=0.0001, verbose=0)

**Step 4: Finding the centroid**

Here is the code for finding the center of the clusters:

Kmean.cluster\_centers\_

Here is the result of the value of the centroids:

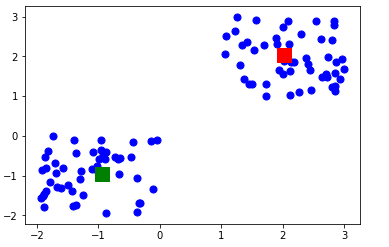
array([[-0.94665068, -0.97138368],  
 [ 2.01559419, 2.02597093]])

Let’s display the cluster centroids (using green and red color).

plt.scatter(X[ : , 0], X[ : , 1], s =50, c=’b’)plt.scatter(-0.94665068, -0.97138368, s=200, c=’g’, marker=’s’)plt.scatter(2.01559419, 2.02597093, s=200, c=’r’, marker=’s’)plt.show()

Here is the output:





**Step 5: Testing the algorithm**

Here is the code for getting the labels property of the K-means clustering example dataset; that is, how the data points are categorized into the two clusters.

Kmean.labels\_

Here is the result of running the above K-means algorithm code:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

As you can see above, 50 data points belong to the **0** cluster while the rest belong to the **1** cluster.

For example, let’s use the code below for predicting the cluster of a data point:

sample\_test=np.array([-3.0,-3.0])second\_test=sample\_test.reshape(1, -1)Kmean.predict(second\_test)

Here is the result:

array([0])

It shows that the test data point belongs to the **0** (green centroid) cluster.

# **Wrapping up**

Here is the entire K-means clustering algorithm code in Python:

import pandas as pdimport numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster import KMeans%matplotlib inlineX= -2 \* np.random.rand(100,2)X1 = 1 + 2 \* np.random.rand(50,2)X[50:100, :] = X1plt.scatter(X[ : , 0], X[ :, 1], s = 50, c = ‘b’)plt.show()from sklearn.cluster import KMeansKmean = KMeans(n\_clusters=2)Kmean.fit(X)Kmean.cluster\_centers\_plt.scatter(X[ : , 0], X[ : , 1], s =50, c=’b’)plt.scatter(-0.94665068, -0.97138368, s=200, c=’g’, marker=’s’)plt.scatter(2.01559419, 2.02597093, s=200, c=’r’, marker=’s’)plt.show()Kmean.labels\_sample\_test=np.array([-3.0,-3.0])second\_test=sample\_test.reshape(1, -1)Kmean.predict(second\_test)

K-means clustering is an extensively used technique for data cluster analysis.

It is easy to understand, especially if you accelerate your learning using a [K-means clustering tutorial](https://www.liveedu.tv/guides/artificial-intelligence/). Furthermore, it delivers training results quickly.

However, its performance is usually not as competitive as those of the other sophisticated clustering techniques because slight variations in the data could lead to high variance.

Furthermore, clusters are assumed to be spherical and evenly sized, something which may reduce the accuracy of the K-means clustering Python results.

LOCAL OUTLIER FACTOR:

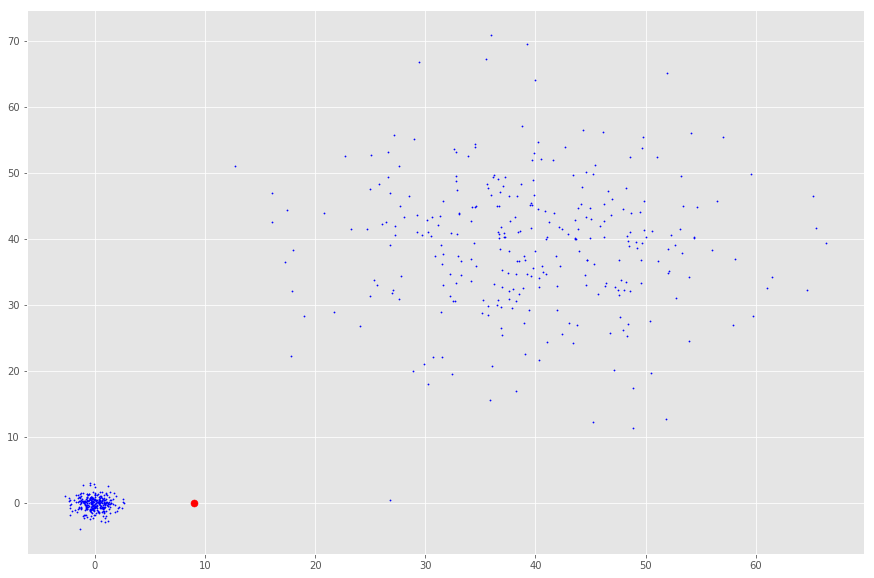
Local Outlier Factor (LOF) is a score that tells how likely a certain data point is an outlier/anomaly.

LOF ≈1 ⇒ no outlier

LOF ≫1 ⇒ outlier

First, I introduce a parameter k which is the number of neighbors the LOF calculation is considering. The LOF is a calculation that looks at the neighbors of a certain point to find out its density and compare this to the density of other points later on. Using a right number k isn’t straight forward. While a small k has a more local focus, i.e. looks only at nearby points, it is more erroneous when having much noise in the data. A large k, however, can miss local outliers.



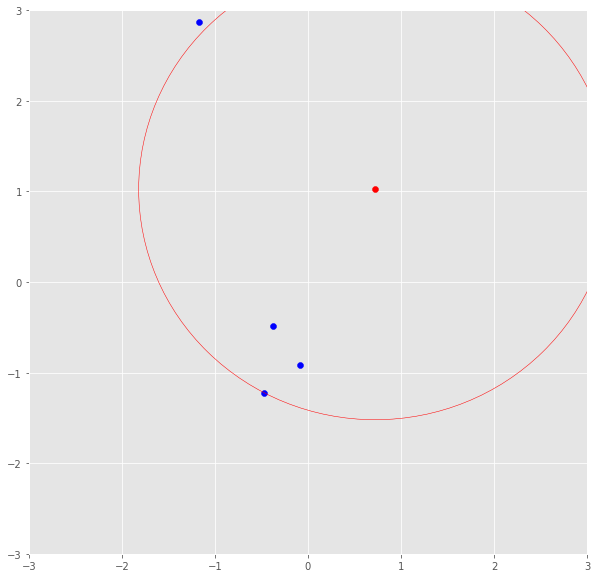


The density of the red point to its nearest neighbors is not different from the density to the cloud in the upper right corner. However, it is probably an outlier compared to the nearest neighbors’ density.

## k-distance

With this k defined, we can introduce the k-distance which is the distance of a point to its kth neighbor. If k was 3, the k-distance would be the distance of a point to the third closest point.





The red point’s k-distance is illustrated by the red line if k=3.

## Reachability distance

The k-distance is now used to calculate the reachability distance. This distance measure is simply the maximum of the distance of two points and the k-distance of the second point.

reach-dist(a,b) = max{k-distance(b), dist(a,b)}

Basically, if point a is within the k neighbors of point b, the reach-dist(a,b) will be the k-distance of b. Otherwise, it will be the real distance of a and b. This is just a “smoothing factor”. For simplicity, consider this the usual distance between two points.

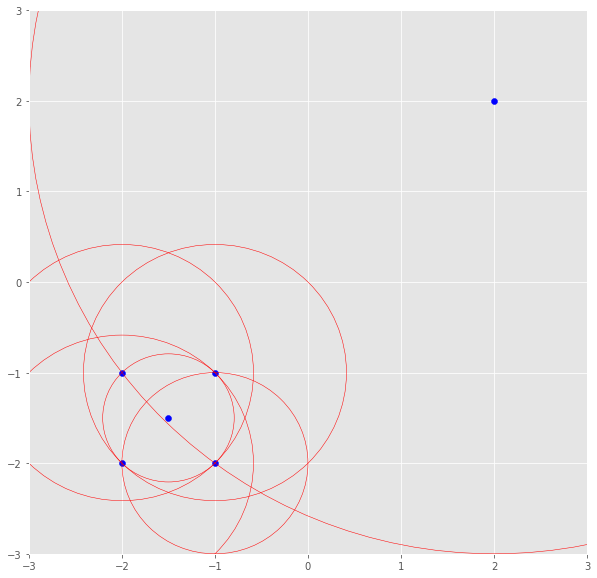
## Local reachability density

The reach-dist is then used to calculate still another concept — the local reachability density (lrd). To get the lrd for a point a, we will first calculate the reachability distance of a to all its k nearest neighbors and take the average of that number. The lrd is then simply the inverse of that average. Remember that we are talking about densities and, therefore, the longer the distance to the next neighbors, the sparser the area the respective point is located in. Hence, the less dense — the inverse.

lrd(a) = 1/(sum(reach-dist(a,n))/k)

By intuition the local reachability density tells how far we have to travel from our point to reach the next point or cluster of points. The lower it is, the less dense it is, the longer we have to travel.





The lrd of the upper right point is the average reachability distance to its nearest neighbors which are points (-1, -1), (-1.5, -1.5) and (-1, -2). These neighbors, however, have other lrds as their nearest neighbors don’t include the upper right point.

## LOF

The lrd of each point will then be compared to the lrd of their k neighbors. More specifically, k ratios of the lrd of each point to its neighboring points will be calculated and averaged. The LOF is basically the average ratio of the lrds of the neighbors of a to the lrd of a. If the ratio is greater than 1, the density of point a is on average smaller than the density of its neighbors and, thus, from point a, we have to travel longer distances to get to the next point or cluster of points than from a’s neighbors to their next neighbors. Keep in mind, the neighbors of a point a may don’t consider a a neighbor as they have points in their reach which are way closer.

In conclusion, the LOF of a point tells the density of this point compared to the density of its neighbors. If the density of a point is much smaller than the densities of its neighbors (LOF ≫1), the point is far from dense areas and, hence, an outlier.

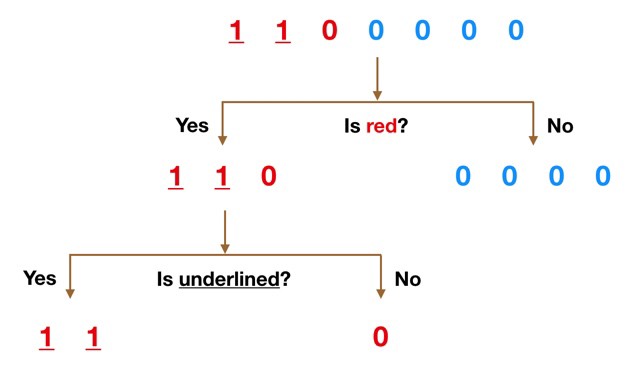
RANDOM FOREST:

Data science provides a plethora of classification algorithms such as logistic regression, support vector machine, naive Bayes classifier, and decision trees. But near the top of the classifier hierarchy is the random forest classifier (there is also the random forest regressor but that is a topic for another day).

In this post, we will examine how basic decision trees work, how individual decisions trees are combined to make a random forest, and ultimately discover why random forests are so good at what they do.

**Decision Trees**

Let’s quickly go over decision trees as they are the building blocks of the random forest model. Fortunately, they are pretty intuitive. I’d be willing to bet that most people have used a decision tree, knowingly or not, at some point in their lives.



Simple Decision Tree Example

It’s probably much easier to understand how a decision tree works through an example.

Imagine that our dataset consists of the numbers at the top of the figure to the left. We have two 1s and five 0s (1s and 0s are our classes) and desire to separate the classes using their features. The features are color (red vs. blue) and whether the observation is underlined or not. So how can we do this?

Color seems like a pretty obvious feature to split by as all but one of the 0s are blue. So we can use the question, “Is it red?” to split our first node. You can think of a node in a tree as the point where the path splits into two — observations that meet the criteria go down the Yes branch and ones that don’t go down the No branch.

The No branch (the blues) is all 0s now so we are done there, but our Yes branch can still be split further. Now we can use the second feature and ask, “Is it underlined?” to make a second split.

The two 1s that are underlined go down the Yes subbranch and the 0 that is not underlined goes down the right subbranch and we are all done. Our decision tree was able to use the two features to split up the data perfectly. Victory!

Obviously in real life our data will not be this clean but the logic that a decision tree employs remains the same. At each node, it will ask —

*What feature will allow me to split the observations at hand in a way that the resulting groups are as different from each other as possible (and the members of each resulting subgroup are as similar to each other as possible)?*

**The Random Forest Classifier**

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model’s prediction (see figure below).



Visualization of a Random Forest Model Making a Prediction

The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science speak, the reason that the random forest model works so well is:

***A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.***

The low correlation between models is the key. Just like how investments with low correlations (like stocks and bonds) come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. **The reason for this wonderful effect is that the trees protect each other from their individual errors** (as long as they don’t constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction. So the prerequisites for random forest to perform well are:

1. There needs to be some actual signal in our features so that models built using those features do better than random guessing.
2. The predictions (and therefore the errors) made by the individual trees need to have low correlations with each other.

**An Example of Why Uncorrelated Outcomes are So Great**

The wonderful effects of having many uncorrelated models is such a critical concept that I want to show you an example to help it really sink in. Imagine that we are playing the following game:

* I use a uniformly distributed random number generator to produce a number.
* If the number I generate is greater than or equal to 40, you win (so you have a 60% chance of victory) and I pay you some money. If it is below 40, I win and you pay me the same amount.
* Now I offer you the the following choices. We can either:

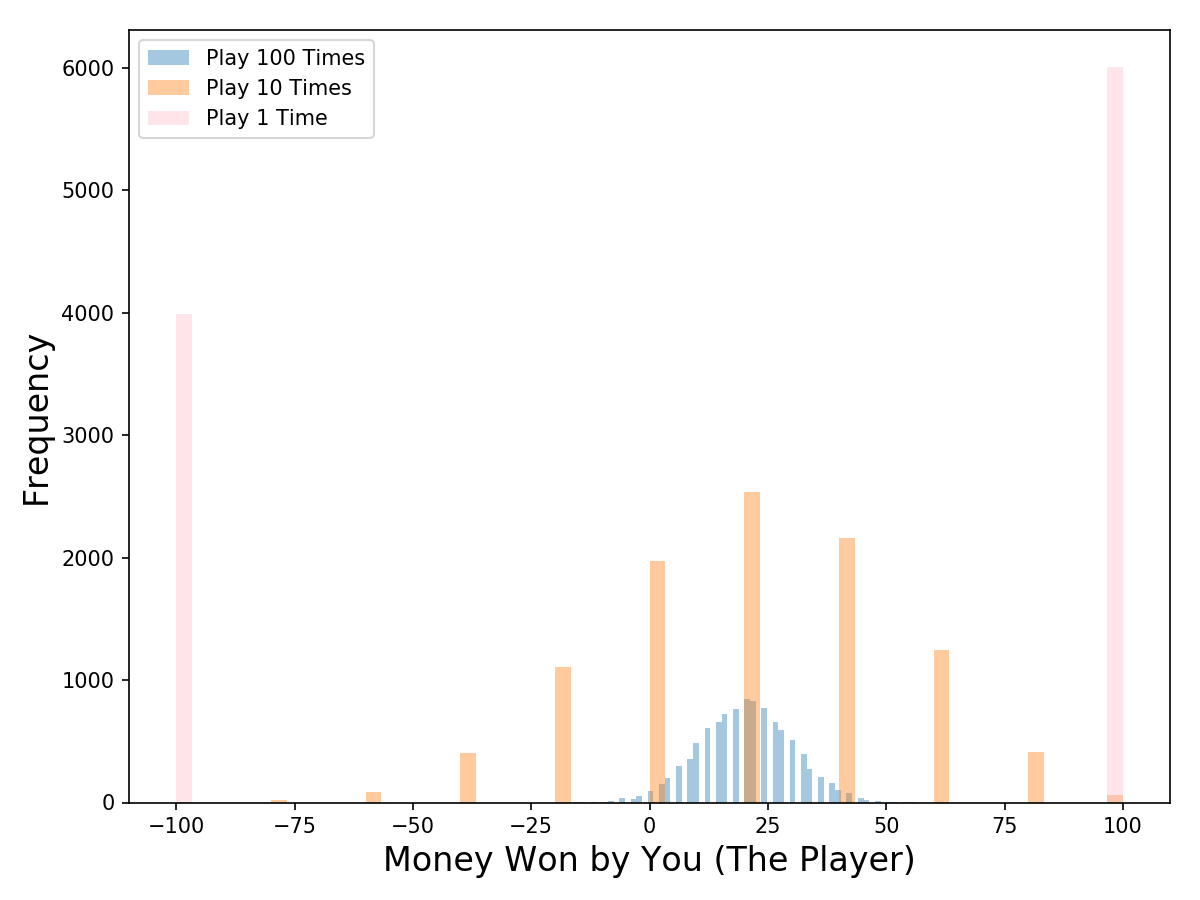
1. **Game 1** — play 100 times, betting $1 each time.
2. **Game 2**— play 10 times, betting $10 each time.
3. **Game 3**— play one time, betting $100.

Which would you pick? The expected value of each game is the same:

*Expected Value Game 1 = (0.60\*1 + 0.40\*-1)\*100 = 20*

*Expected Value Game 2= (0.60\*10 + 0.40\*-10)\*10 = 20*

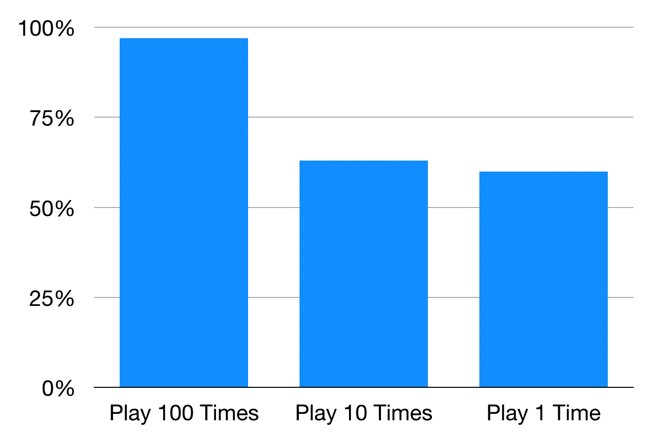
*Expected Value Game 3= 0.60\*100 + 0.40\*-100 = 20*



Outcome Distribution of 10,000 Simulations for each Game

What about the distributions? Let’s visualize the results with a Monte Carlo simulation (we will run 10,000 simulations of each game type; **for example, we will simulate 10,000 times the 100 plays of Game 1**). Take a look at the chart on the left — now which game would you pick? Even though the expected values are the same, **the outcome distributions are vastly different going from positive and narrow (blue) to binary (pink).**

Game 1 (where we play 100 times) offers up the best chance of making some money — **out of the 10,000 simulations that I ran, you make money in 97% of them!** For Game 2 (where we play 10 times) you make money in 63% of the simulations, a drastic decline (and a drastic increase in your probability of losing money). And Game 3 that we only play once, you make money in 60% of the simulations, as expected.



Probability of Making Money for Each Game

So even though the games share the same expected value, their outcome distributions are completely different. The more we split up our $100 bet into different plays, the more confident we can be that we will make money. As mentioned previously, this works because each play is independent of the other ones.

Random forest is the same — each tree is like one play in our game earlier. We just saw how our chances of making money increased the more times we played. Similarly, with a random forest model, our chances of making correct predictions increase with the number of uncorrelated trees in our model.

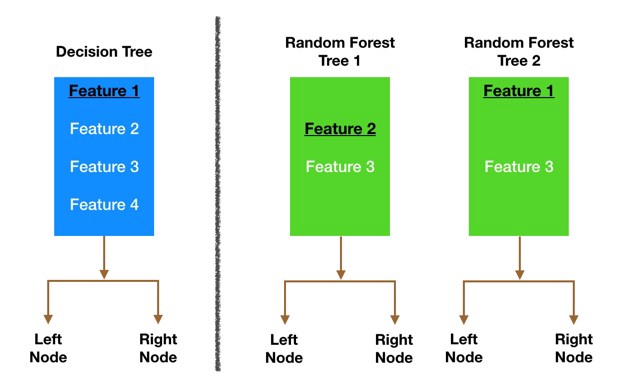
If you would like to run the code for simulating the game yourself you can find it on my GitHub here.

**Ensuring that the Models Diversify Each Other**

So how does random forest ensure that the behavior of each individual tree is not too correlated with the behavior of any of the other trees in the model? It uses the following two methods:

**Bagging (Bootstrap Aggregation) — Decisions trees are very sensitive to the data they are trained on — small changes to the training set can result in significantly different tree structures.** Random forest takes advantage of this by allowing each individual tree to randomly sample from the dataset with replacement, resulting in different trees. This process is known as bagging.

Notice that with bagging we are not subsetting the training data into smaller chunks and training each tree on a different chunk. Rather, if we have a sample of size N, we are still feeding each tree a training set of size N (unless specified otherwise). But instead of the original training data, we take a random sample of size N with replacement. For example, if our training data was [1, 2, 3, 4, 5, 6] then we might give one of our trees the following list [1, 2, 2, 3, 6, 6]. Notice that both lists are of length six and that “2” and “6” are both repeated in the randomly selected training data we give to our tree (because we sample with replacement).



Node splitting in a random forest model is based on a random subset of features for each tree.

**Feature Randomness —**In a normal decision tree, when it is time to split a node, we consider every possible feature and pick the one that produces the most separation between the observations in the left node vs. those in the right node. In contrast, each tree in a random forest can pick only from a random subset of features. This forces even more variation amongst the trees in the model and ultimately results in lower correlation across trees and more diversification.

Let’s go through a visual example — in the picture above, the traditional decision tree (in blue) can select from all four features when deciding how to split the node. It decides to go with Feature 1 (black and underlined) as it splits the data into groups that are as separated as possible.

Now let’s take a look at our random forest. We will just examine two of the forest’s trees in this example. When we check out random forest Tree 1, we find that it it can only consider Features 2 and 3 (selected randomly) for its node splitting decision. We know from our traditional decision tree (in blue) that Feature 1 is the best feature for splitting, but Tree 1 cannot see Feature 1 so it is forced to go with Feature 2 (black and underlined). Tree 2, on the other hand, can only see Features 1 and 3 so it is able to pick Feature 1.

***So in our random forest, we end up with trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions.***

And that, my dear reader, creates uncorrelated trees that buffer and protect each other from their errors.

DECISION TREE:

Classification is a two-step process, learning step and prediction step, in machine learning. In the learning step, the model is developed based on given training data. In the prediction step, the model is used to predict the response for given data. Decision Tree is one of the easiest and popular classification algorithms to understand and interpret.

**Decision Tree Algorithm**

Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, the decision tree algorithm can be used for solving **regression and classification problems** too.

The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by **learning simple decision rules** inferred from prior data(training data).

In Decision Trees, for predicting a class label for a record we start from the **root** of the tree. We compare the values of the root attribute with the record’s attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

**Types of Decision Trees**

Types of decision trees are based on the type of target variable we have. It can be of two types:

1. **Categorical Variable Decision Tree:**Decision Tree which has a categorical target variable then it called a **Categorical variable decision tree.**
2. **Continuous Variable Decision Tree:**Decision Tree has a continuous target variable then it is called **Continuous Variable Decision Tree.**

**Example: -** Let’s say we have a problem to predict whether a customer will pay his renewal premium with an insurance company (yes/ no). Here we know that the income of customers is a significant variable but the insurance company does not have income details for all customers. Now, as we know this is an important variable, then we can build a decision tree to predict customer income based on occupation, product, and various other variables. In this case, we are predicting values for the continuous variables.

**Important Terminology related to Decision Trees**

1. **Root Node:**It represents the entire population or sample and this further gets divided into two or more homogeneous sets.
2. **Splitting:**It is a process of dividing a node into two or more sub-nodes.
3. **Decision Node:**When a sub-node splits into further sub-nodes, then it is called the decision node.
4. **Leaf / Terminal Node:**Nodes do not split is called Leaf or Terminal node.
5. **Pruning:**When we remove sub-nodes of a decision node, this process is called pruning. You can say the opposite process of splitting.
6. **Branch / Sub-Tree:**A subsection of the entire tree is called branch or sub-tree.
7. **Parent and Child Node:**A node, which is divided into sub-nodes is called a parent node of sub-nodes whereas sub-nodes are the child of a parent node.



Decision trees classify the examples by sorting them down the tree from the root to some leaf/terminal node, with the leaf/terminal node providing the classification of the example.

Each node in the tree acts as a test case for some attribute, and each edge descending from the node corresponds to the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new node.

**Assumptions while creating Decision Tree**

Below are some of the assumptions we make while using Decision tree:

* In the beginning, the whole training set is considered as the **root.**
* Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
* Records are **distributed recursively** on the basis of attribute values.
* Order to placing attributes as root or internal node of the tree is done by using some statistical approach.

Decision Trees follow **Sum of Product (SOP) r**epresentation. The Sum of product (SOP) is also known as **Disjunctive Normal Form**. For a class, every branch from the root of the tree to a leaf node having the same class is conjunction (product) of values, different branches ending in that class form a disjunction (sum).

The primary challenge in the decision tree implementation is to identify which attributes do we need to consider as the root node and each level. Handling this is to know as the attributes selection. We have different attributes selection measures to identify the attribute which can be considered as the root note at each level.

**How do Decision Trees work?**

The decision of making strategic splits heavily affects a tree’s accuracy. The decision criteria are different for classification and regression trees.

Decision trees use multiple algorithms to decide to split a node into two or more sub-nodes. The creation of sub-nodes increases the homogeneity of resultant sub-nodes. In other words, we can say that the purity of the node increases with respect to the target variable. The decision tree splits the nodes on all available variables and then selects the split which results in most homogeneous sub-nodes.

The algorithm selection is also based on the type of target variables. Let us look at some algorithms used in Decision Trees:

**ID3** → (extension of D3)  
**C4.5** → (successor of ID3)  
**CART** → (Classification And Regression Tree)  
**CHAID** → (Chi-square automatic interaction detection Performs multi-level splits when computing classification trees)  
**MARS** → (multivariate adaptive regression splines)

The ID3 algorithm builds decision trees using a top-down greedy search approach through the space of possible branches with no backtracking. A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment.

**Steps in ID3 algorithm:**

1. It begins with the original set S as the root node.
2. On each iteration of the algorithm, it iterates through the very unused attribute of the set S and calculates **Entropy(H)** and **Information gain(IG)**of this attribute.
3. It then selects the attribute which has the smallest Entropy or Largest Information gain.
4. The set S is then split by the selected attribute to produce a subset of the data.
5. The algorithm continues to recur on each subset, considering only attributes never selected before.

**Attribute Selection Measures**

If the dataset consists of **N** attributes then deciding which attribute to place at the root or at different levels of the tree as internal nodes is a complicated step. By just randomly selecting any node to be the root can’t solve the issue. If we follow a random approach, it may give us bad results with low accuracy.

For solving this attribute selection problem, researchers worked and devised some solutions. They suggested using some *criteria* like :

**Entropy**,

I**nformation gain,**

**Gini index,**

**Gain Ratio,**

**Reduction in Variance**

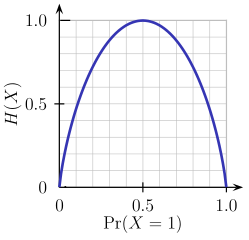
**Chi-Square**

These criterions will calculate values for every attribute. The values are sorted, and attributes are placed in the tree by following the order i.e., the attribute with a high value (in case of information gain) is placed at the root.

While using Information Gain as a criterion, we assume attributes to be categorical, and for the Gini index, attributes are assumed to be continuous.

**Entropy**

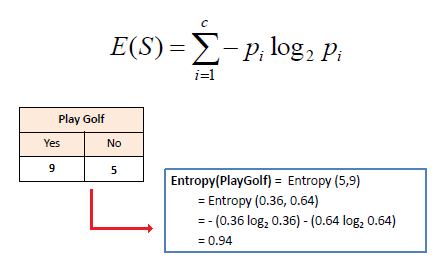
Entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information. Flipping a coin is an example of an action that provides information that is random.



From the above graph, it is quite evident that the entropy H(X) is zero when the probability is either 0 or 1. The Entropy is maximum when the probability is 0.5 because it projects perfect randomness in the data and there is no chance if perfectly determining the outcome.

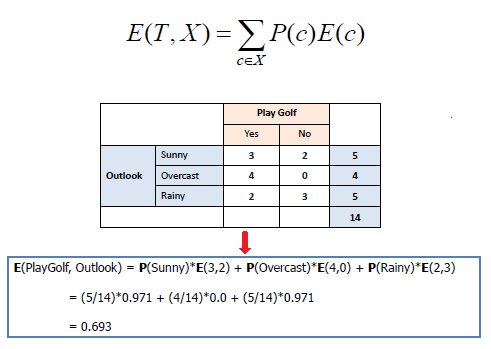
***ID3 follows the rule — A branch with an entropy of zero is a leaf node and A brach with entropy more than zero needs further splitting.***

Mathematically Entropy for 1 attribute is represented as:



Where **S → Current state, and Pi → Probability of an event *i*of state S or Percentage of class *i* in a node of state S.**

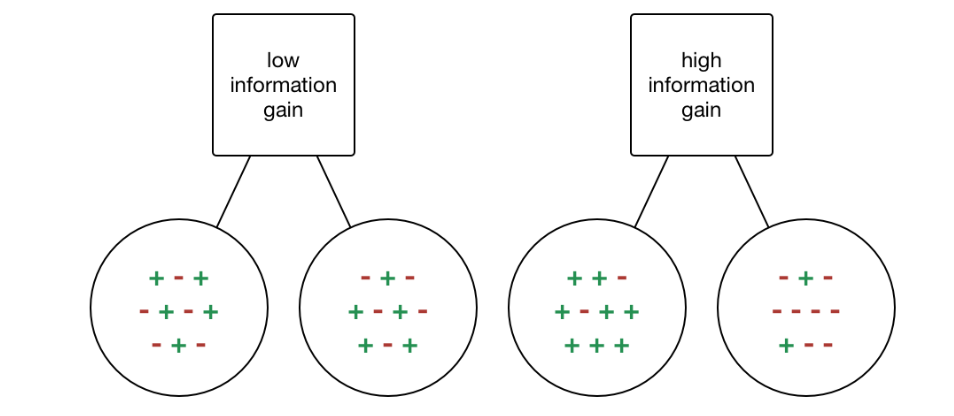
Mathematically Entropy for multiple attributes is represented as:



where**T→ Current state and X → Selected attribute**

**Information Gain**

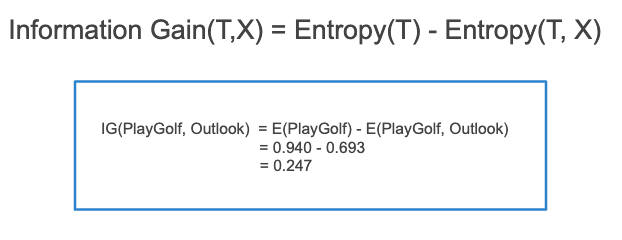
Information gain or **IG**is a statistical property that measures how well a given attribute separates the training examples according to their target classification. Constructing a decision tree is all about finding an attribute that returns the highest information gain and the smallest entropy.



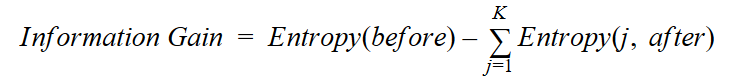
Information Gain

Information gain is a decrease in entropy. It computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values. ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

Mathematically, IG is represented as:



In a much simpler way, we can conclude that:

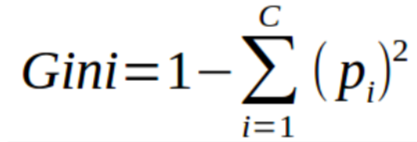


Information Gain

Where “before” is the dataset before the split, K is the number of subsets generated by the split, and (j, after) is subset j after the split.

**Gini Index**

You can understand the Gini index as a cost function used to evaluate splits in the dataset. It is calculated by subtracting the sum of the squared probabilities of each class from one. It favors larger partitions and easy to implement whereas information gain favors smaller partitions with distinct values.



Gini Index

Gini Index works with the categorical target variable “Success” or “Failure”. It performs only Binary splits.

*Higher the value of Gini index higher the homogeneity.*

**Steps to Calculate Gini index for a split**

1. Calculate Gini for sub-nodes, using the above formula for success(p) and failure(q) (p²+q²).
2. Calculate the Gini index for split using the weighted Gini score of each node of that split.

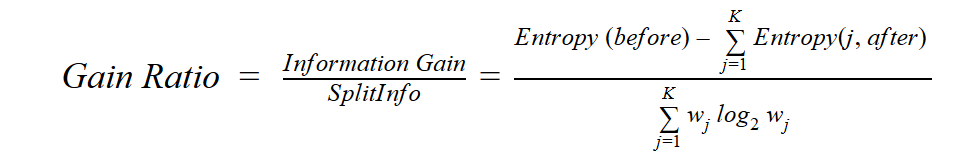
CART (Classification and Regression Tree) uses the Gini index method to create split points.

**Gain ratio**

Information gain is biased towards choosing attributes with a large number of values as root nodes. It means it prefers the attribute with a large number of distinct values.

C4.5, an improvement of ID3, uses Gain ratio which is a modification of Information gain that reduces its bias and is usually the best option. Gain ratio overcomes the problem with information gain by taking into account the number of branches that would result before making the split. It corrects information gain by taking the intrinsic information of a split into account.

*Let us consider if we have a dataset that has users and their movie genre preferences based on variables like gender, group of age, rating, blah, blah. With the help of information gain, you split at ‘Gender’ (assuming it has the highest information gain) and now the variables ‘Group of Age’ and ‘Rating’ could be equally important and with the help of gain ratio, it will penalize a variable with more distinct values which will help us decide the split at the next level.*

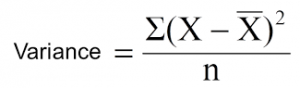


Gain Ratio

Where “before” is the dataset before the split, K is the number of subsets generated by the split, and (j, after) is subset j after the split.

**Reduction in Variance**

**Reduction in variance** is an algorithm used for continuous target variables (regression problems). This algorithm uses the standard formula of variance to choose the best split. The split with lower variance is selected as the criteria to split the population:



Above X-bar is the mean of the values, X is actual and n is the number of values.

**Steps to calculate Variance:**

1. Calculate variance for each node.
2. Calculate variance for each split as the weighted average of each node variance.

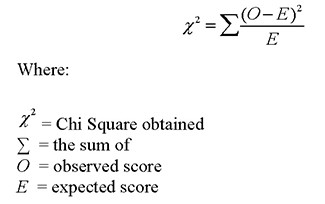
**Chi-Square**

The acronym CHAID stands for *Chi*-squared Automatic Interaction Detector. It is one of the oldest tree classification methods. It finds out the statistical significance between the differences between sub-nodes and parent node. We measure it by the sum of squares of standardized differences between observed and expected frequencies of the target variable.

It works with the categorical target variable “Success” or “Failure”. It can perform two or more splits. Higher the value of Chi-Square higher the statistical significance of differences between sub-node and Parent node.

It generates a tree called CHAID (Chi-square Automatic Interaction Detector).

Mathematically, Chi-squared is represented as:



**Steps to Calculate Chi-square for a split:**

1. Calculate Chi-square for an individual node by calculating the deviation for Success and Failure both
2. Calculated Chi-square of Split using Sum of all Chi-square of success and Failure of each node of the split

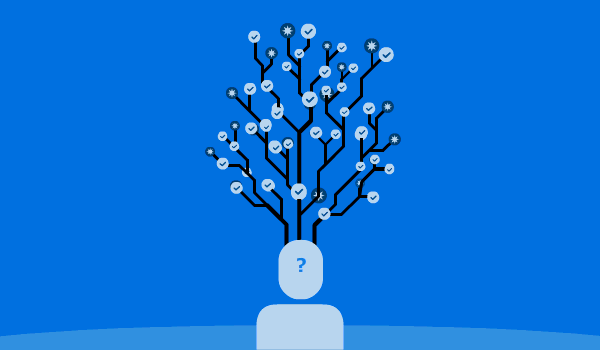
**How to avoid/counter Overfitting in Decision Trees?**

The common problem with Decision trees, especially having a table full of columns, they fit a lot. Sometimes it looks like the tree memorized the training data set. If there is no limit set on a decision tree, it will give you 100% accuracy on the training data set because in the worse case it will end up making 1 leaf for each observation. Thus this affects the accuracy when predicting samples that are not part of the training set.

Here are two ways to remove overfitting:

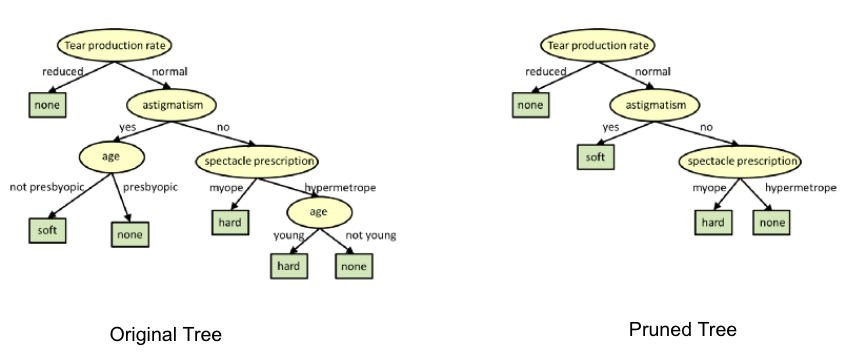
1. Pruning Decision Trees.
2. Random Forest

**Pruning Decision Trees**The splitting process results in fully grown trees until the stopping criteria are reached. But, the fully grown tree is likely to overfit the data, leading to poor accuracy on unseen data.



Pruning in action

In **pruning**, you trim off the branches of the tree, i.e., remove the decision nodes starting from the leaf node such that the overall accuracy is not disturbed. This is done by segregating the actual training set into two sets: training data set, D and validation data set, V. Prepare the decision tree using the segregated training data set, D. Then continue trimming the tree accordingly to optimize the accuracy of the validation data set, V.



Pruning

In the above diagram, the ‘Age’ attribute in the left-hand side of the tree has been pruned as it has more importance on the right-hand side of the tree, hence removing overfitting.

K-NN:

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.

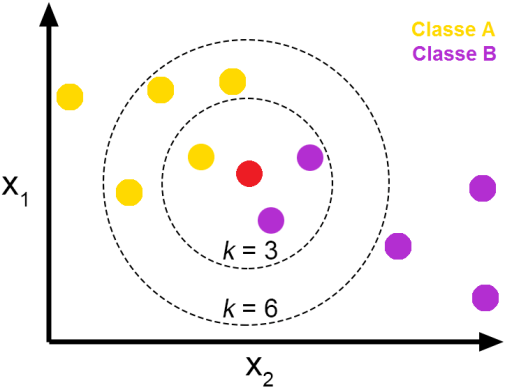
It is called Lazy algorithm because it does not need any training data points for model generation. All training data is used in the testing phase which makes training faster and testing phase slower and costlier.

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.



Source: Towards Data Science

**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** = (*p*1, *p*2,…, *pn*) and **q** = (*q*1, *q*2,…, *qn*) represent the n attribute values of two records.

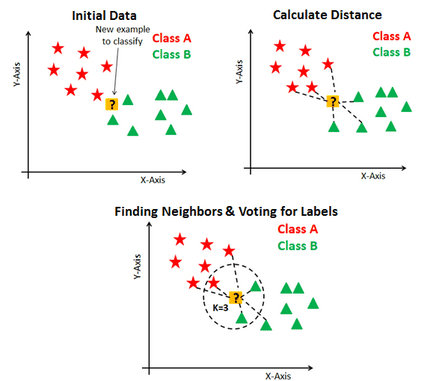
* While Euclidean distance is useful in low dimensions, it **doesn’t work well in high dimensions and for categorical variables**. The drawback of Euclidean distance is that it **ignores the similarity between attributes**. Each attribute is treated as totally different from all of the attributes.

**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.



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 behavior.
4. Lends itself very easily to parallel implementation.

MODULES

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](https://en.wikipedia.org/wiki/CPython) reference [implementation](https://en.wikipedia.org/wiki/Programming_language_implementation) of Python, which is a non-optimizing [bytecode](https://en.wikipedia.org/wiki/Bytecode) interpreter. Mathematical algorithms written for this version of Python often run much slower than [compiled](https://en.wikipedia.org/wiki/Compiler) 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](https://en.wikipedia.org/wiki/MATLAB) since they are both interpreted,[[16]](https://en.wikipedia.org/wiki/NumPy#cite_note-16) and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of [scalars](https://en.wikipedia.org/wiki/Scalar_(computing)). In comparison, MATLAB boasts a large number of additional toolboxes, notably [Simulink](https://en.wikipedia.org/wiki/Simulink), whereas NumPy is intrinsically integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; [SciPy](https://en.wikipedia.org/wiki/SciPy) is a library that adds more MATLAB-like functionality and [Matplotlib](https://en.wikipedia.org/wiki/Matplotlib) is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on [BLAS](https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms) and [LAPACK](https://en.wikipedia.org/wiki/LAPACK) for efficient linear algebra computations.

Python [bindings](https://en.wikipedia.org/wiki/Language_binding) of the widely used [computer vision](https://en.wikipedia.org/wiki/Computer_vision) library [OpenCV](https://en.wikipedia.org/wiki/OpenCV) utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, [slicing](https://en.wikipedia.org/wiki/Array_slicing#1991:_Python) or [masking](https://en.wikipedia.org/wiki/Mask_(computing)#Image_masks) 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](https://en.wikipedia.org/wiki/Interest_point_detection), [filter kernels](https://en.wikipedia.org/wiki/Kernel_(image_processing)) and many more vastly simplifies the programming workflow and [debugging](https://en.wikipedia.org/wiki/Debugger).

**The ndarray data structure**

The core functionality of NumPy is its "ndarray", for *n*-dimensional array, data structure. These arrays are [strided](https://en.wikipedia.org/wiki/Stride_of_an_array) views on memory.[[7]](https://en.wikipedia.org/wiki/NumPy#cite_note-cise-7) In contrast to Python's built-in list data structure (which, despite the name, is a [dynamic array](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/C_Programming_Language)/[C++](https://en.wikipedia.org/wiki/C%2B%2B), [Cython](https://en.wikipedia.org/wiki/Cython), and [Fortran](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/SciPy) package, which wraps a number of such libraries (notably [BLAS](https://en.wikipedia.org/wiki/BLAS) and [LAPACK](https://en.wikipedia.org/wiki/LAPACK)). NumPy has built-in support for [memory-mapped](https://en.wikipedia.org/wiki/Memory-mapped_file) ndarrays.[[7]](https://en.wikipedia.org/wiki/NumPy#cite_note-cise-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]](https://en.wikipedia.org/wiki/NumPy#cite_note-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]](https://en.wikipedia.org/wiki/NumPy#cite_note-18) and [Numba](https://en.wikipedia.org/wiki/Numba).[[19]](https://en.wikipedia.org/wiki/NumPy#cite_note-19) [Cython](https://en.wikipedia.org/wiki/Cython) and [Pythran](https://en.wikipedia.org/w/index.php?title=Pythran&action=edit&redlink=1) 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 *n*rows 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)

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](https://www.python.org/) 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**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html#pandas.Series) (1-dimensional) and [**DataFrame**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html#pandas.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**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html#pandas.DataFrame) provides everything that R’s data.frame provides and much more. pandas is built on top of [NumPy](https://www.numpy.org/) 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 [Cython](https://cython.org/)code. 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](https://www.statsmodels.org/stable/index.html), making it an important part of the statistical computing ecosystem in Python.
* pandas has been used extensively in production in financial applications.

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]: import** **numpy** **as** **np**

**In [2]: import** **pandas** **as** **pd**

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**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html#pandas.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
* an ndarray
* 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 an ndarray, **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](http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing).

Like a NumPy array, a pandas Series has a [**dtype**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.dtype.html#pandas.Series.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**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.api.extensions.ExtensionDtype.html#pandas.api.extensions.ExtensionDtype). Some examples within pandas are [Categorical Data](http://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html#categorical) and [Nullable Integer Data Type](http://pandas.pydata.org/pandas-docs/stable/user_guide/integer_na.html#integer-na). See [dtypes](http://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html#basics-dtypes) for more.

If you need the actual array backing a Series, use [**Series.array**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.array.html#pandas.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](http://pandas.pydata.org/pandas-docs/stable/getting_started/dsintro.html#dsintro-alignment), for example).

[**Series.array**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.array.html#pandas.Series.array) will always be an [**ExtensionArray**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.api.extensions.ExtensionArray.html#pandas.api.extensions.ExtensionArray). Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a [**numpy.ndarray**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray). Pandas knows how to take an ExtensionArray and store it in a Series or a column of a DataFrame. See [dtypes](http://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html#basics-dtypes) for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use [**Series.to\_numpy()**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.to_numpy.html#pandas.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**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.api.extensions.ExtensionArray.html#pandas.api.extensions.ExtensionArray), [**Series.to\_numpy()**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.to_numpy.html#pandas.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](http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing-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 an ndarray.

**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()**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.rename.html#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.

DataFrame

**DataFrame** is a 2-dimensional labeled 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](https://docs.scipy.org/doc/numpy/user/basics.rec.html) 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 4.0 NaN

b 2.0 NaN

a 1.0 NaN

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

0 1.0 4.0

1 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

0 1 2.0 b'Hello'

1 2 3.0 b'World'

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

Out[50]:

A B C

first 1 2.0 b'Hello'

second 2 3.0 b'World'

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

Out[51]:

C A B

0 b'Hello' 1 2.0

1 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

1 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

1 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 NaN NaN NaN NaN 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](http://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#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

0 1 4

1 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 B

C

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]: del** df['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 foo one\_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 NaN NaN False bar NaN

Assigning New Columns in Method Chains

Inspired by [dplyr’s](https://dplyr.tidyverse.org/reference/mutate.html) mutate verb, DataFrame has an [**assign()**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.assign.html#pandas.DataFrame.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]:

SepalLength SepalWidth PetalLength PetalWidth 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]:**

SepalLength SepalWidth PetalLength PetalWidth 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 DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

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

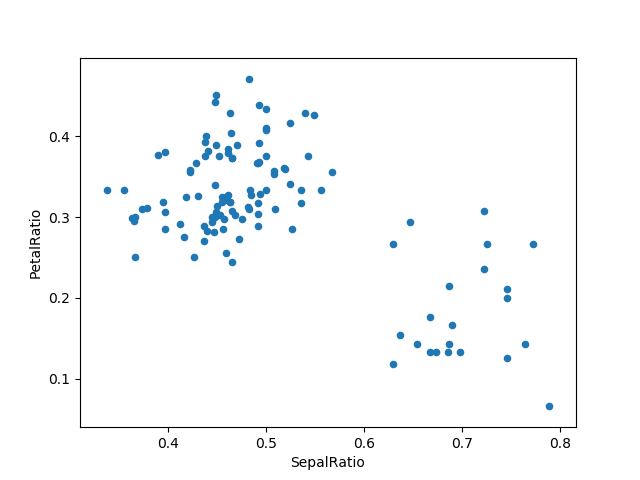
**....:**  .assign(SepalRatio=**lambda** x: x.SepalWidth / x.SepalLength,

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

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

**....:**

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



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that’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()**](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.assign.html#pandas.DataFrame.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

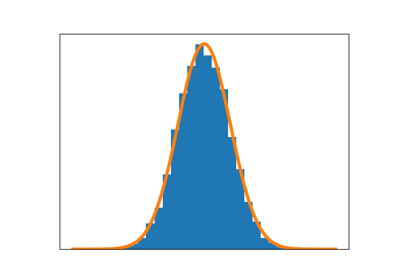
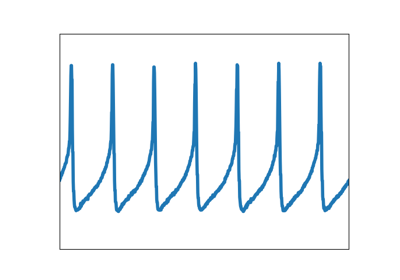
0 2 4

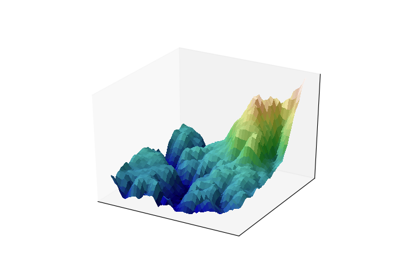
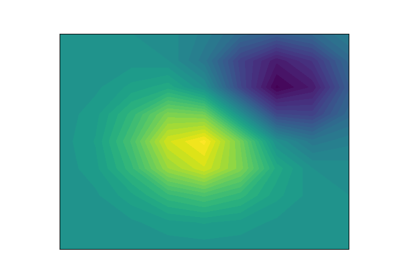
1 2 4

2 2 4

Matplotlib:

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](http://ipython.org/) shells, the [Jupyter](http://jupyter.org/) notebook, web application servers, and four graphical user interface toolkits.

[[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

[[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

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](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html) and [thumbnail gallery](https://matplotlib.org/3.0.3/gallery/index.html).

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**](https://matplotlib.org/3.0.3/users/installing.html#id3)

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](https://matplotlib.org/3.0.3/users/installing.html#install-scipy-dists) below.

[**macOS**](https://matplotlib.org/3.0.3/users/installing.html#id4)

To use the native OSX backend you will need [a framework build](https://matplotlib.org/3.0.3/faq/osx_framework.html#osxframework-faq) build of Python.

[**Test data**](https://matplotlib.org/3.0.3/users/installing.html#id5)

The wheels (\*.whl) on the [PyPI download page](https://pypi.python.org/pypi/matplotlib/) 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](https://pypi.python.org/pypi/pytest), Pillow, MiKTeX, GhostScript, ffmpeg, avconv, ImageMagick, and [Inkscape](https://inkscape.org/);
* run python -mpytest.

[**Third-party distributions of Matplotlib**](https://matplotlib.org/3.0.3/users/installing.html#id6)

[**Scientific Python Distributions**](https://matplotlib.org/3.0.3/users/installing.html#id7)

[Anaconda](https://www.continuum.io/downloads/) and [Canopy](https://www.enthought.com/products/canopy/) and [ActiveState](https://www.activestate.com/activepython/downloads) are excellent choices that "just work" out of the box for Windows, macOS and common Linux platforms. [WinPython](https://winpython.github.io/) 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**](https://matplotlib.org/3.0.3/users/installing.html#id8)

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**](https://matplotlib.org/3.0.3/users/installing.html#id9)

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](https://pypi.python.org/pypi/matplotlib/), or if you want to develop Matplotlib or just need the latest bugfixed version, grab the latest git version [Install from source](https://matplotlib.org/3.0.3/faq/installing_faq.html#install-from-git).

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 -mpip install .

We provide a [setup.cfg](https://raw.githubusercontent.com/matplotlib/matplotlib/master/setup.cfg.template) 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**](https://matplotlib.org/3.0.3/users/installing.html#id10)

Matplotlib requires the following dependencies:

* [Python](https://www.python.org/downloads/) (>= 3.5)
* [FreeType](https://www.freetype.org/) (>= 2.3)
* [libpng](http://www.libpng.org/) (>= 1.2)
* [NumPy](http://www.numpy.org/) (>= 1.10.0)
* [setuptools](https://setuptools.readthedocs.io/en/latest/)
* [cycler](http://matplotlib.org/cycler/) (>= 0.10.0)
* [dateutil](https://pypi.python.org/pypi/python-dateutil) (>= 2.1)
* [kiwisolver](https://github.com/nucleic/kiwi) (>= 1.0.0)
* [pyparsing](https://pyparsing.wikispaces.com/)

Optionally, you can also install a number of packages to enable better user interface toolkits. See [What is a backend?](https://matplotlib.org/3.0.3/tutorials/introductory/usage.html#what-is-a-backend) for more details on the optional Matplotlib backends and the capabilities they provide.

* [tk](https://matplotlib.org/3.0.3/glossary/index.html#term-tk) (>= 8.3, != 8.6.0 or 8.6.1): for the Tk-based backends;
* [PyQt4](https://pypi.python.org/pypi/PyQt4) (>= 4.6) or [PySide](https://pypi.python.org/pypi/PySide) (>= 1.0.3): for the Qt4-based backends;
* [PyQt5](https://pypi.python.org/pypi/PyQt5): for the Qt5-based backends;
* [PyGObject](https://pypi.org/project/PyGObject/) or [pgi](https://pypi.org/project/pgi/) (>= 0.0.11.2): for the GTK3-based backends;
* [wxpython](https://matplotlib.org/3.0.3/glossary/index.html#term-wxpython) (>= 4): for the WX-based backends;
* [cairocffi](https://cairocffi.readthedocs.io/en/latest/) (>= 0.8) or [pycairo](https://pypi.python.org/pypi/pycairo): for the cairo-based backends;
* [Tornado](https://pypi.python.org/pypi/tornado): for the WebAgg backend;

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

* [ffmpeg](https://www.ffmpeg.org/)/[avconv](https://libav.org/avconv.html): for saving movies;
* [ImageMagick](https://www.imagemagick.org/script/index.php): for saving animated gifs;
* [Pillow](https://pillow.readthedocs.io/en/latest/) (>= 3.4): for a larger selection of image file formats: JPEG, BMP, and TIFF image files;
* [LaTeX](https://miktex.org/) and [GhostScript (>=9.0)](https://ghostscript.com/download/) : for rendering text with LaTeX.

[**Building on Linux**](https://matplotlib.org/3.0.3/users/installing.html#id11)

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:

sudo dnf builddep 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**](https://matplotlib.org/3.0.3/users/installing.html#id12)

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 libpng freetype pkg-config

If you are using MacPorts, execute the following instead:

port install libpng freetype pkgconfig

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

python -mpip install .

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**](https://matplotlib.org/3.0.3/users/installing.html#id13)

The Python shipped from [https://www.python.org](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](https://github.com/jbmohler/matplotlib-winbuild).

There are a few possibilities to build Matplotlib on Windows:

* Wheels via [matplotlib-winbuild](https://github.com/jbmohler/matplotlib-winbuild)
* Wheels by using conda packages (see below)
* Conda packages (see below)

[**Wheel builds using conda packages**](https://matplotlib.org/3.0.3/users/installing.html#id14)

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-dateutil pyparsing tornado cycler tk libpng zlib freetype msinttypes

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](https://matplotlib.org/3.0.3/api/_as_gen/matplotlib.pyplot.html#module-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.

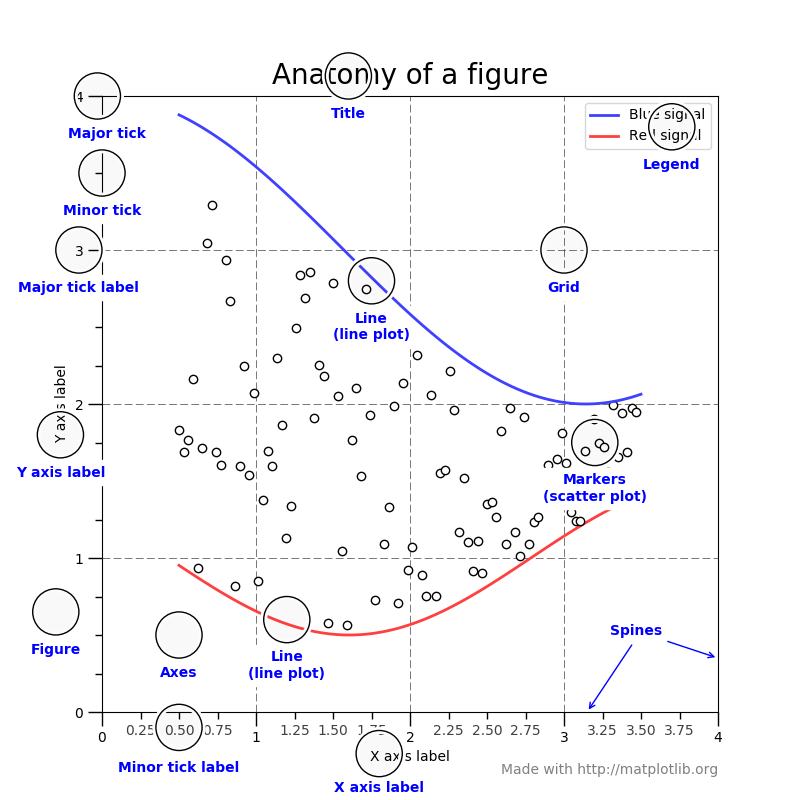
For even more control -- which is essential for things like embedding matplotlib plots in GUI applications -- the pyplot level may be dropped completely, leaving a purely object-oriented approach.

*# sphinx\_gallery\_thumbnail\_number = 3*

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

## Parts of a Figure



### [Figure](https://matplotlib.org/3.0.3/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure)

The **whole** figure. The figure keeps track of all the child [Axes](https://matplotlib.org/3.0.3/api/axes_api.html#matplotlib.axes.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](https://matplotlib.org/3.0.3/api/axes_api.html#matplotlib.axes.Axes), but to be useful should have at least one.

SEABORN:

Seaborn is a Python data visualization library based on [matplotlib](https://matplotlib.org/). 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](https://matplotlib.org/) and closely integrated with [pandas](https://pandas.pydata.org/) data structures.

Here is some of the functionality that seaborn offers:

* A dataset-oriented API for examining [relationships](https://seaborn.pydata.org/examples/scatter_bubbles.html#scatter-bubbles) between [multiple variables](https://seaborn.pydata.org/examples/faceted_lineplot.html#faceted-lineplot)
* Specialized support for using categorical variables to show [observations](https://seaborn.pydata.org/examples/jitter_stripplot.html#jitter-stripplot) or [aggregate statistics](https://seaborn.pydata.org/examples/pointplot_anova.html#pointplot-anova)
* Options for visualizing [univariate](https://seaborn.pydata.org/examples/distplot_options.html#distplot-options) or [bivariate](https://seaborn.pydata.org/examples/joint_kde.html#joint-kde) distributions and for [comparing](https://seaborn.pydata.org/examples/horizontal_boxplot.html#horizontal-boxplot) them between subsets of data
* Automatic estimation and plotting of [linear regression](https://seaborn.pydata.org/examples/anscombes_quartet.html#anscombes-quartet) models for different kinds [dependent](https://seaborn.pydata.org/examples/logistic_regression.html#logistic-regression) variables
* Convenient views onto the overall [structure](https://seaborn.pydata.org/examples/scatterplot_matrix.html#scatterplot-matrix) of complex datasets
* High-level abstractions for structuring [multi-plot grids](https://seaborn.pydata.org/examples/faceted_histogram.html#faceted-histogram) that let you easily build [complex](https://seaborn.pydata.org/examples/pair_grid_with_kde.html#pair-grid-with-kde) visualizations
* Concise control over matplotlib figure styling with several [built-in themes](https://seaborn.pydata.org/tutorial/aesthetics.html#aesthetics-tutorial)
* Tools for choosing [color palettes](https://seaborn.pydata.org/tutorial/color_palettes.html#palette-tutorial) 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:

**import** **seaborn** **as** **sns**

sns.set()

tips = sns.load\_dataset("tips")

sns.relplot(x="total\_bill", y="tip", col="time",

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

data=tips);



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.

**import** **seaborn** **as** **sns**

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](https://seaborn.pydata.org/introduction.html#intro-plot-customization). For interactive work, it’s recommended to use a Jupyter/IPython interface in [matplotlib mode](https://ipython.readthedocs.io/en/stable/interactive/plotting.html), or else you’ll have to call matplotlib.pyplot.show when you want to see the plot.

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

sns.set()

This uses the [matplotlib rcParam system](https://matplotlib.org/users/customizing.html) 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](https://seaborn.pydata.org/tutorial/aesthetics.html#aesthetics-tutorial), 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.

1. We load one of the example datasets.

tips = sns.load\_dataset("tips")

Most code in the docs will use the [**load\_dataset()**](https://seaborn.pydata.org/generated/seaborn.load_dataset.html#seaborn.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](https://seaborn.pydata.org/introduction.html#intro-tidy-data).

1. 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()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.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()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.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()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) function has a convenient kind parameter to let you easily switch to this alternate representation:

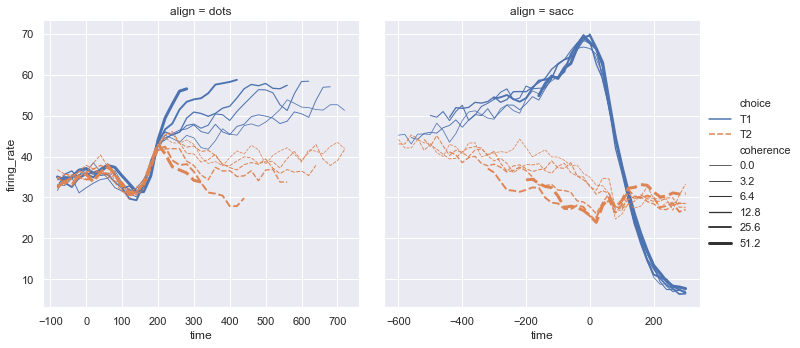
dots = sns.load\_dataset("dots")

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);



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

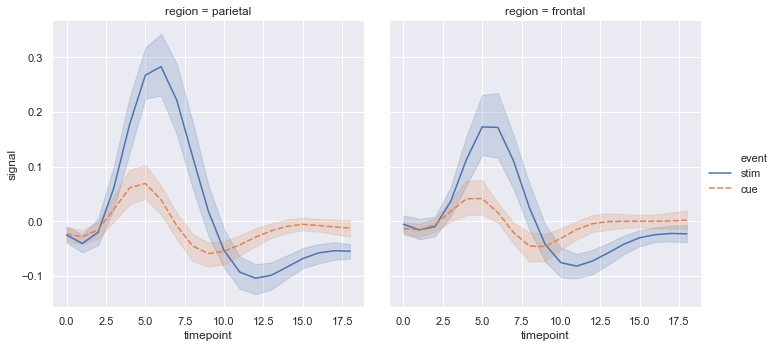
Often we are interested in the average value of one variable as a function of other variables. Many seaborn functions can automatically perform the statistical estimation that is neccesary to answer these questions:

fmri = sns.load\_dataset("fmri")

sns.relplot(x="timepoint", y="signal", col="region",

hue="event", style="event",

kind="line", data=fmri);

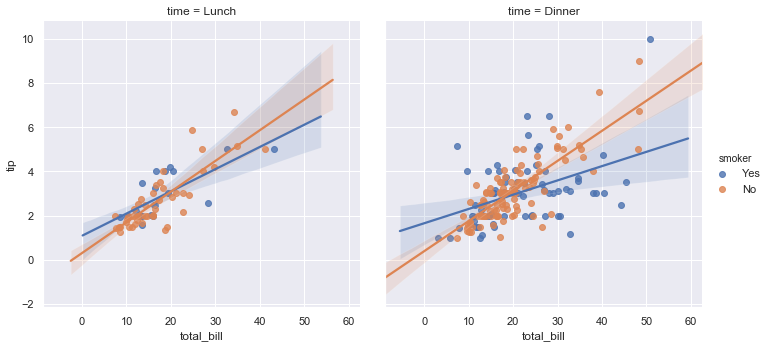


When statistical values are estimated, seaborn will use bootstrapping to compute confidence intervals and draw error bars representing the uncertainty of the estimate.

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()**](https://seaborn.pydata.org/generated/seaborn.lmplot.html#seaborn.lmplot):

sns.lmplot(x="total\_bill", y="tip", col="time", hue="smoker",

data=tips);



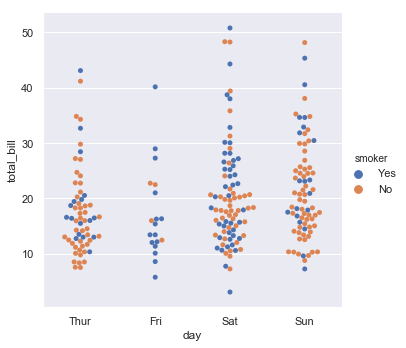
Specialized categorical plots

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()**](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot). Similar to [**relplot()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot), the idea of [**catplot()**](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.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",

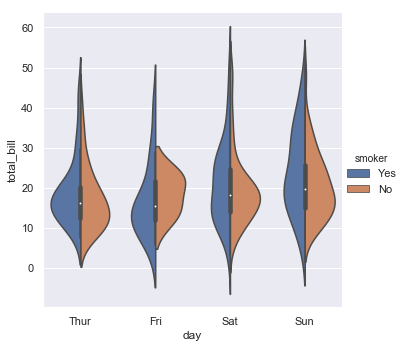
kind="swarm", data=tips);



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);



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);

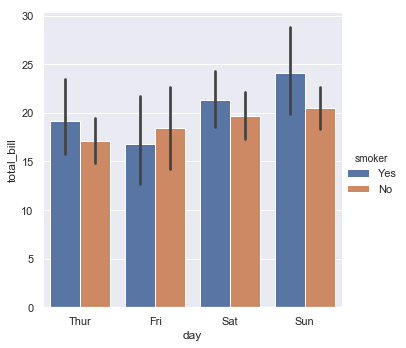


Figure-level and axes-level functions

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**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid).

Each different figure-level plot kind combines a particular “axes-level” function with the [**FacetGrid**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid) object. For example, the scatter plots are drawn using the [**scatterplot()**](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) function, and the bar plots are drawn using the [**barplot()**](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.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.

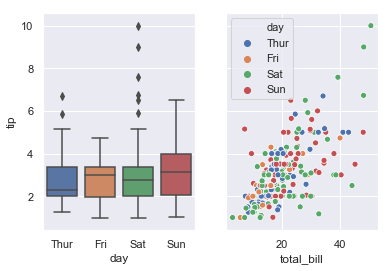
The upshot is that the figure-level function needs to control the figure it lives in, while axes-level functions can be combined into a more complex matplotlib figure with other axes that may or may not have seaborn plots on them:

**import** **matplotlib.pyplot** **as** **plt**

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:

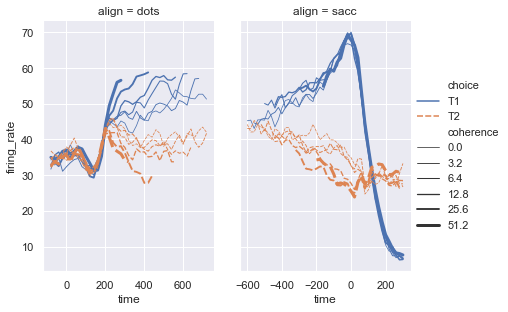
sns.relplot(x="time", y="firing\_rate", col="align",

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

height=4.5, aspect=2 / 3,

facet\_kws=dict(sharex=**False**),

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



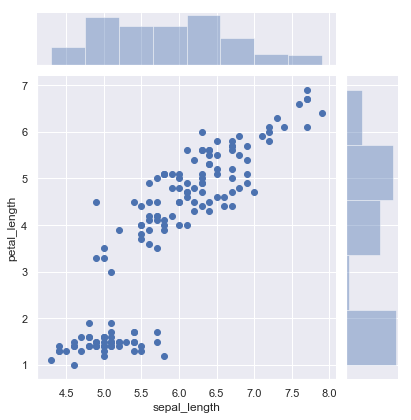
The way you can tell whether a function is “figure-level” or “axes-level” is whether it takes an ax= parameter. You can also distinguish the two classes by their output type: axes-level functions return the matplotlib axes, while figure-level functions return the [**FacetGrid**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid).

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()**](https://seaborn.pydata.org/generated/seaborn.jointplot.html#seaborn.jointplot), focuses on a single relationship:

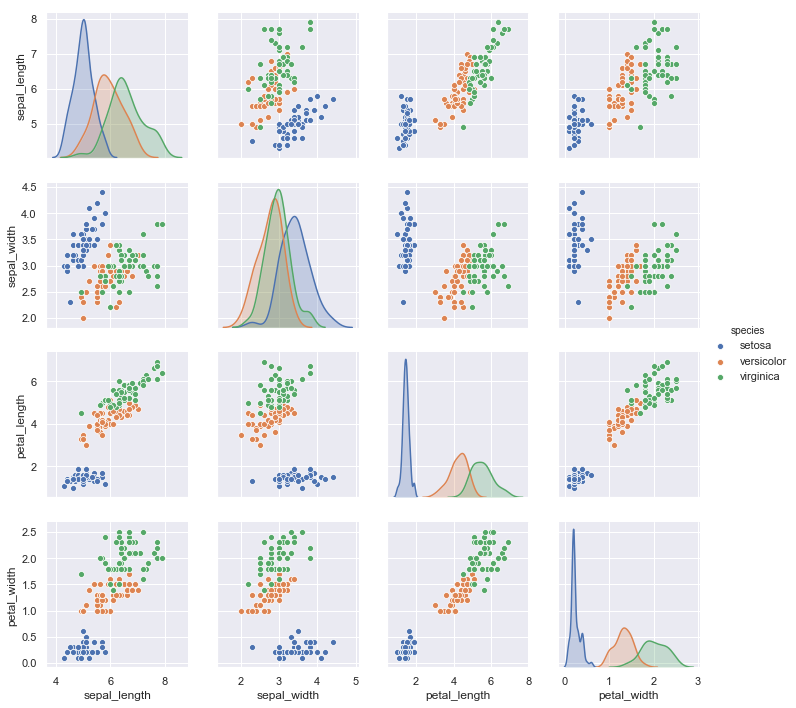
iris = sns.load\_dataset("iris")

sns.jointplot(x="sepal\_length", y="petal\_length", data=iris);



The other, [**pairplot()**](https://seaborn.pydata.org/generated/seaborn.pairplot.html#seaborn.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");



Both [**jointplot()**](https://seaborn.pydata.org/generated/seaborn.jointplot.html#seaborn.jointplot) and [**pairplot()**](https://seaborn.pydata.org/generated/seaborn.pairplot.html#seaborn.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**](https://seaborn.pydata.org/generated/seaborn.JointGrid.html#seaborn.JointGrid) and [**PairGrid**](https://seaborn.pydata.org/generated/seaborn.PairGrid.html#seaborn.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.

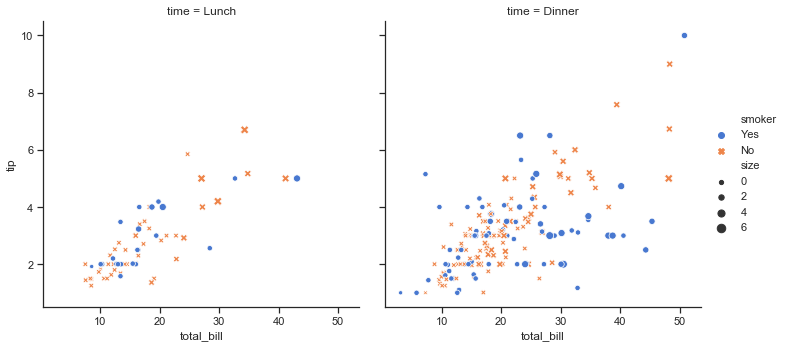
The first way is to use one of the alternate seaborn themes to give your plots a different look. Setting a different theme or color palette will make it take effect for all plots:

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

sns.relplot(x="total\_bill", y="tip", col="time",

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

data=tips);



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](https://seaborn.pydata.org/tutorial/color_palettes.html#palette-tutorial) for customizing color palettes).

Finally, where there is a direct correspondence with an underlying matplotlib function (like [**scatterplot()**](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) and plt.scatter), additional keyword arguments will be passed through to the matplotlib layer:

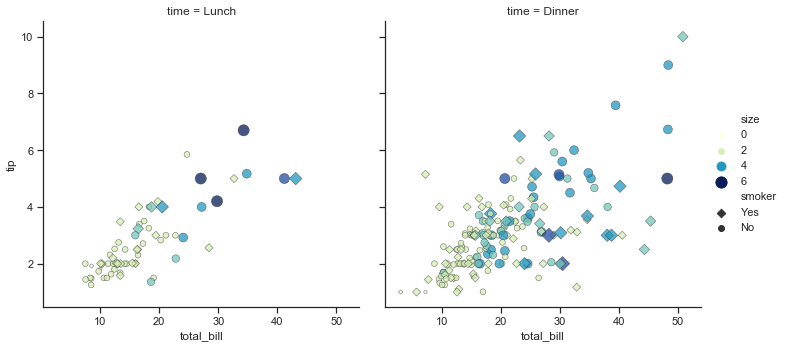
sns.relplot(x="total\_bill", y="tip", col="time",

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

palette="YlGnBu", markers=["D", "o"], sizes=(10, 125),

edgecolor=".2", linewidth=.5, alpha=.75,

data=tips);



In the case of [**relplot()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) and other figure-level functions, that means there are a few levels of indirection because [**relplot()**](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.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**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.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 ($)", "")

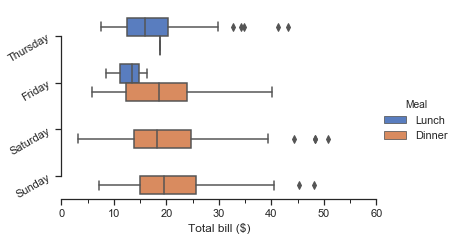
g.set(xlim=(0, 60), yticklabels=["Thursday", "Friday", "Saturday", "Sunday"])

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](https://github.com/wagnerlabpapers/Waskom_PNAS_2017) 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](http://vita.had.co.nz/papers/tidy-data.html). 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 |

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](https://seaborn.pydata.org/introduction.html#intro-stat-estimation) 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 |

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.

Sklearn:

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 Cournapeau 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, Tinyclues and the Python Software Foundation.

In general, a learning problem considers a set of n [samples](https://en.wikipedia.org/wiki/Sample_(statistics)) 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](https://en.wikipedia.org/wiki/Multivariate_random_variable) data), it is said to have several attributes or **features**.

Learning problems fall into a few categories:

* [supervised learning](https://en.wikipedia.org/wiki/Supervised_learning), in which the data comes with additional attributes that we want to predict ([Click here](https://scikit-learn.org/stable/supervised_learning.html#supervised-learning) to go to the scikit-learn supervised learning page).This problem can be either:
  + [classification](https://en.wikipedia.org/wiki/Classification_in_machine_learning): 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](https://en.wikipedia.org/wiki/Regression_analysis): 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Cluster_analysis), or to determine the distribution of data within the input space, known as [density estimation](https://en.wikipedia.org/wiki/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](https://scikit-learn.org/stable/unsupervised_learning.html#unsupervised-learning) 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](https://en.wikipedia.org/wiki/Iris_flower_data_set) and [digits](http://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits) datasets for classification and the [boston house prices dataset](https://archive.ics.uci.edu/ml/machine-learning-databases/housing/) 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](https://scikit-learn.org/stable/datasets/index.html#datasets).

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](https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html#sphx-glr-auto-examples-classification-plot-digits-classification-py) 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](https://scikit-learn.org/stable/datasets/index.html#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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Support_vector_machine). 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](https://scikit-learn.org/stable/modules/grid_search.html#grid-search) and [cross validation](https://scikit-learn.org/stable/modules/cross_validation.html#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)

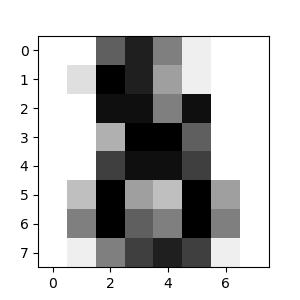
Now you can *predict* new values. In this case, you’ll predict using the last image from digits.data. By predicting, you’ll determine the image from the training set that best matches the last image.

>>>

**>>>** clf.predict(digits.data[-1:])

array([8])

The corresponding image is:

[](https://scikit-learn.org/stable/auto_examples/datasets/plot_digits_last_image.html)