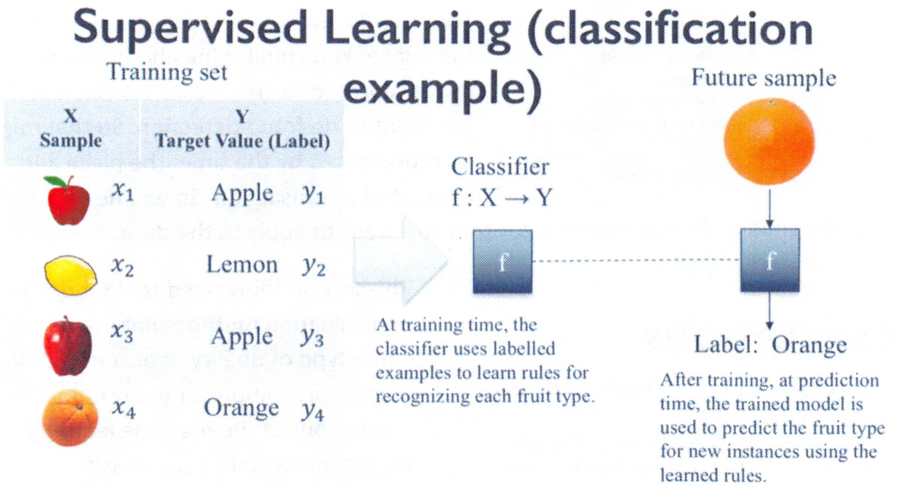
[Applied ML in Python- Week1]

Key types of Machine Learning problems

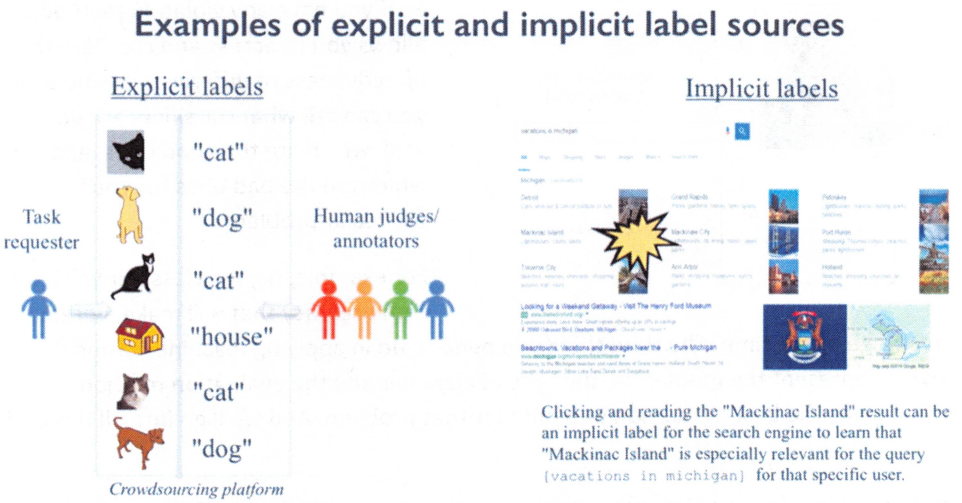
Supervised machine learning: Learn to predict target values from labelled data.

- Classification (target values are discrete classes)
- Regression (target values are continuous values)



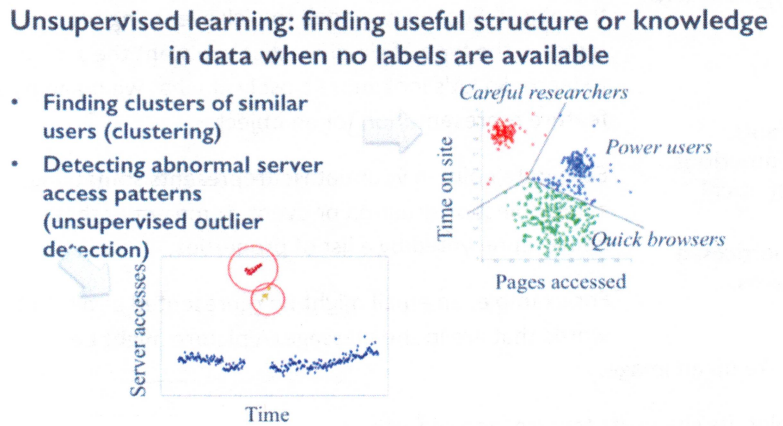
And to do this, the system's given a set of label training examples, of inputs X sub i and outputs Y sub i. And this set of label training examples is what's used to identify the function that best maps the input to the desired output.

For example, if the supervised learning problem is image recognition. This involves building a classifier where the input X sub i could be a set of pixels that describe a single image. And the desired label Y sub i might be the label of the object in the image. Now there're many algorithms that scientists have developed that can do supervise learning. That could be used to estimate this function F from the training data, and we'll cover a number of those algorithms in the course.



Now obtaining labels for some problems can be easy or difficult, depending on how much labeled data is needed.
The level of human expertise or expert knowledge that is needed to provide an accurate label.
And the complexity of the labeling task among other factors
We can also obtain implicit labels.

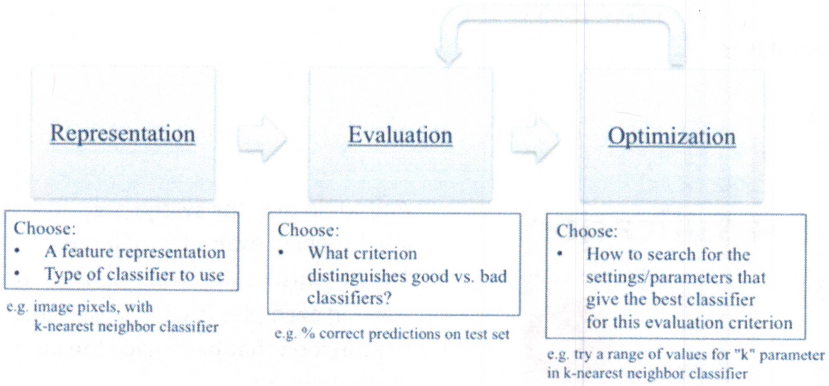
So, for example, if a search engine detects a user clicking on a result link and then sees no more activity for another minute or two before the user comes back to the search engine. The system might use that activity as a kind of an implicit label for that page. In other words, if the user took some time to visit the page, it was more likely that that page was relevant to their query.



The second major class of machine learning algorithms is called unsupervised learning. In many cases we only have input data, we don't have any labels to go with the data. And in those cases the problems we can solve involve taking the input data and trying to find some kind of useful structure in it.

Since there can be many different types of hacking or intrusion attempts to break into the server or exploit in some way. We don't have reliable training labels that

A Basic Machine Learning Workflow

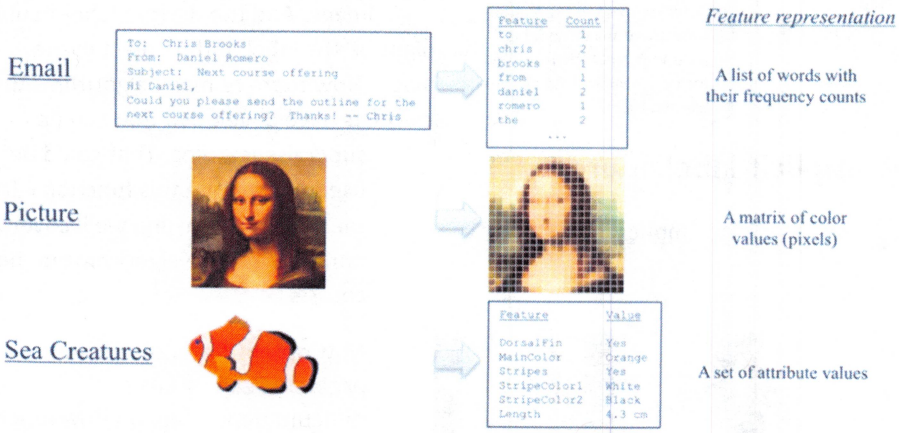


we could use to train the classifier using supervised learning. Instead, we need unsupervised approach that allows us to perform something called outlier detection. That doesn't assume future attacks will be of the same form as previous attacks.

classification, regression, learning algorithm

There could also be metadata associated with the image. You could think about how you might represent a credit card transaction if you want to do fraud detection. So that might be represented by the time, the place and amount of a transaction. So you need some way, some representation of the data you have and the choice of what kind of algorithm you want to apply to the data.

Feature Representations



The second thing need to do is decide on an evaluation method that provides some type of quality or accuracy score. For the predictions or the output that is coming out of the machine learning algorithm typically I say classifier.

So if you have any violation method this allows you to access and compare the effectiveness of different classifiers. So you can tell what classifiers are doing well, which are the good ones, and which are the bad ones for your particular problem.

For example, a good classifier will have high accuracy that will make a prediction that matches the correct true label a high percentage of the time. The third thing you need to do in applying machine learning to solve a problem is. Once we've decided on how to represent the input data, the type of classifier and the evaluation method. We need to then search for the optimal classifier that gives the best evaluation outcome for that problem. And we'll go into all three of these areas in this course.

Let's first talk about what it means to convert the problem into a representation that a computer can deal with. This involves two things. You need to convert each input object, which we often call a sample, into a set of features that describe the object. Second, we need to pick a learning model, typically the type of classifier that you want the system to learn. So let's look more closely at what we mean by a feature representation for an object.

SciPy Library: Scientific Computing Tools

http://www.scipy.org/

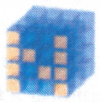
- **Provides a variety of useful scientific computing tools, including statistical distributions, optimization of functions, linear algebra, and a variety of specialized mathematical functions.**
- **With scikit-learn, it provides support for *sparse matrices*, a way to store large tables that consist mostly of zeros.**
- **Example import: `import scipy as sp`**

Each data point in your dataset represents something, some object or situation or event. Some entity that's being represented by a list of properties.

For example, an email might be represented as a list of words that are in the message. A picture might be represented by a matrix of color values for the pixels that make up an image.

A piece of fruit, like this apple could be represented by its color, its shape, its texture, and so forth.

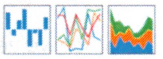
NumPy: Scientific Computing Library



http://www.numpy.org/

- Provides fundamental data structures used by scikit-learn, particularly multi-dimensional arrays.
- Typically, data that is input to scikit-learn will be in the form of a NumPy array.
- Example import: `import numpy as np`

Pandas: Data Manipulation and Analysis

pandas  http://pandas.pydata.org/

- Provides key data structures like `DataFrame`
- Also, support for reading/writing data in different formats
- Example import: `import pandas as pd`

matplotlib and other plotting libraries

matplotlib http://matplotlib.org/

- We typically use matplotlib's pyplot module:
  `import matplotlib.pyplot as plt`
- We also sometimes use the seaborn visualization library (http://seaborn.pydata.org/)
  `import seaborn as sn`
- And sometimes the graphviz plotting library:
  `import graphviz`

The other key part of representing a machine learning problem is choosing the type of classifier that's appropriate for the problem. And we'll cover many different types of classifiers in this course. They all have different trade-offs in terms of their accuracy, their interpretability, and their speed, and so forth.

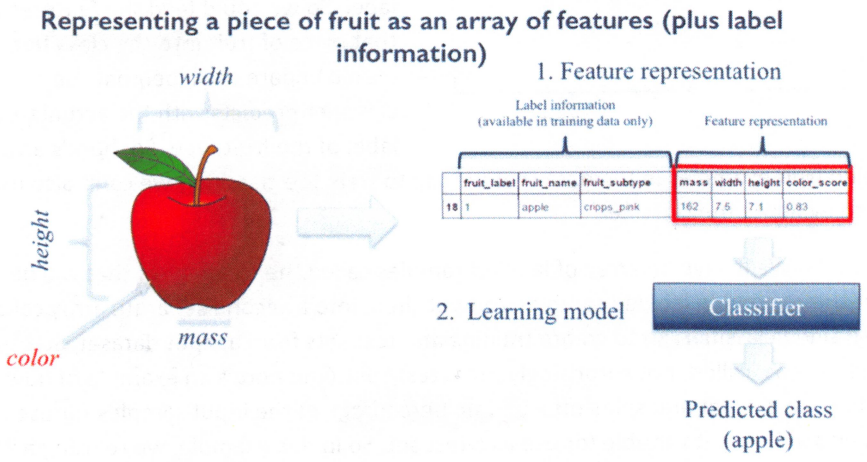
[Python tools]

The most important library we'll be using for machine learning is called scikit-learn. Scikit-learn is the most widely used Python library for machine learning and it will be the basis for this course. So one valuable reference that we recommend you use while you take this course is the scikit-learn User Guide and the API documentation which contains further details on the different algorithms that are in the library, along with details on the various options that these algorithms support. Scikit-SciPy is a Python library that supports data manipulation and analysis methods that are commonly used in scientific computing. So this includes support for statistical distributions, optimization of functions, linear algebra, and a variety of specialized mathematical functions.learn makes use of two other Python scientific computing libraries called SciPy and NumPy.

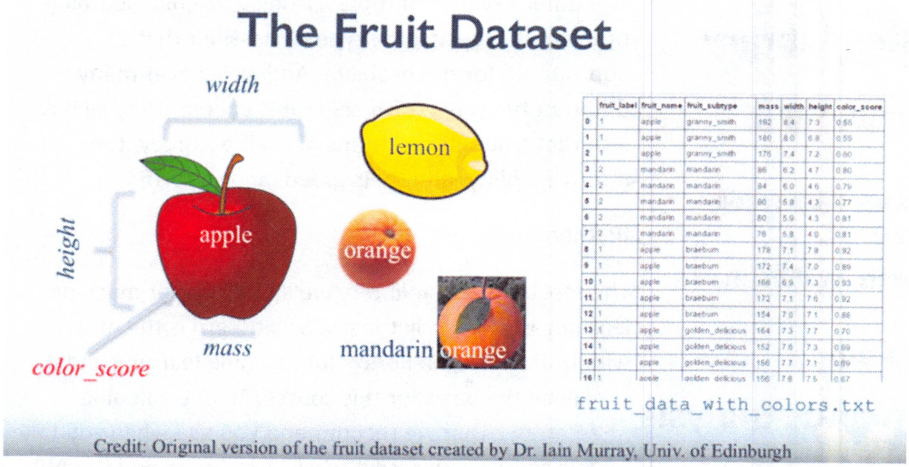
SciPy is a Python library that supports data manipulation and analysis methods that are commonly used in scientific computing. So this includes support for statistical distributions, optimization of functions, linear algebra, and a variety of specialized mathematical functions.

In some parts of this course, we'll make use of SciPy's ability to provide what are called sparse matrices, which are a way to store large tables that contain mostly zeros, so more on that later. NumPy is a Python library for scientific computing that contains support for some fundamental data structures used by scikit-learn, such as multi-dimensional arrays. Typically, data that's input to scikit-learn will be in the form of a NumPy array.

Pandas is a Python library for data manipulation and analysis. And if you took course one in this series, you've already had experience with what pandas can do.



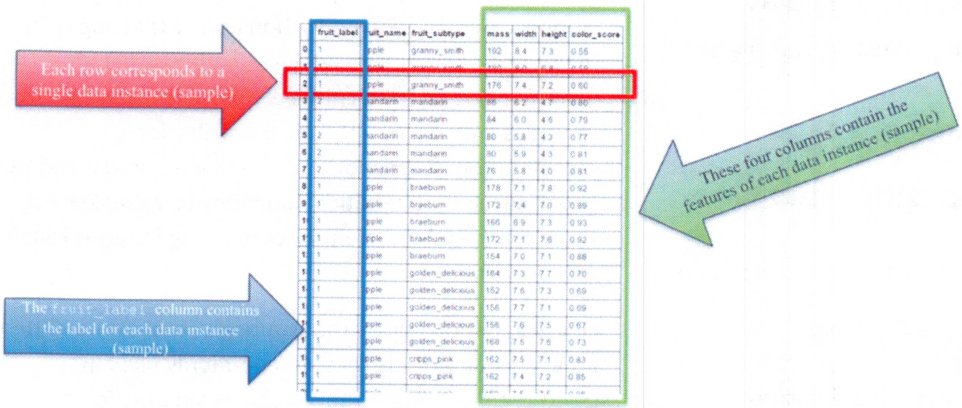
The key data structure pandas supports that we'll be using is called a DataFrame, which is basically like a spreadsheet table with rows and named columns.



So unlike the arrays supported by NumPy, the columns in a DataFrame can be of all different types. You can have one column holding string values, another column holding a date, another column holding a floating point number and so on.

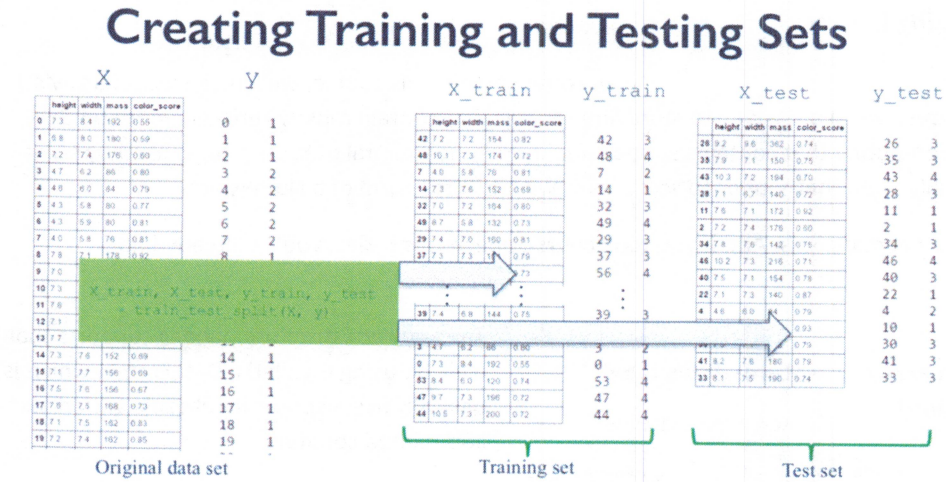
Pandas also has great support for reading and writing data in a variety of formats, everything from comma separated CSV files to SQL, the Structured Query Language used by databases and more.

[**An example ML problem**]



So, we've formatted his original data slightly and added one or two extra simulated features such as a color score for instructional purposes. This dataset is called fruit_data_with_colors.txt. The fruit shipping companies that screen for rotten oranges during processing.

Anyway, to solve machine learning problems, you can thing of the input data as a table. Where each object, so in our case a piece of fruit, is represented by a row, and the attributes of the object, the measurement, the color, the size, and so forth in our case for a piece of fruit, the features of the fruit are represented by the values that you see across the columns.



Now, assuming for the moment that we already we had a classifier ready to go, how would we know if its predictions were likely to be accurate? Well, we could choose a fruit sample, called a test sample, for which we already had a label. So we could feed the features of that piece of fruit **into the classifier**, and then compare the label that the classifier predicts with the actual true label of the fruit type. So, here's a very important point though, if we use one of our labeled fruit examples in the data that we use to train the classifier, we can't also use that same fruit sample later as a test sample to also evaluate the classifier.

what we'll do is split the original dataset into two parts. We'll have an array of labeled samples called the training set that will be used to train the classifier. And then we'll hold out the remaining labeled samples and put them into a second separate array called the test set that will be used to then evaluate the trained classifier. So to create training and test sets from a input dataset, Scikit-learn provides a handy function that will do this split for us, called, not surprisingly, train test split. And here's an example of how we're going to use it. This function randomly shuffles the dataset and splits off a certain percentage of the input samples for use as a training set, and then puts the remaining samples into a different variable for use as a test set. So in this example, we're using a 75-

```
In [88]: fruits.shape
Out[88]: (59, 7)

In [92]: X_train.shape
Out[92]: (44, 4)

In [93]: X_test.shape
Out[93]: (15, 4)

In [94]: y_train.shape
Out[94]: (44,)

In [95]: y_test.shape
Out[95]: (15,)
```

In [96]: X_train

Out[96]:

	height	width	mass	color_score
42	7.2	7.2	154	0.82
48	10.1	7.3	174	0.72
7	4.0	5.8	76	0.81
14	7.3	7.6	152	0.69
32	7.0	7.2	164	0.80
49	8.7	5.8	132	0.73
29	7.4	7.0	160	0.81
37	7.3	7.3	154	0.79
56	8.1	5.9	116	0.73
18	7.1	7.5	162	0.83
55	7.7	6.3	116	0.72
27	9.2	7.5	204	0.77
15	7.1	7.7	156	0.69
5	4.3	5.8	80	0.77
31	8.0	7.8	210	0.82
16	7.5	7.6	156	0.67

In [98]: y_train

Out[98]:

```
42    3
48    4
7     2
14    1
32    3
49    4
29    3
37    3
56    4
18    1
55    4
27    3
15    1
5     2
31    3
16    1
50    4
20    1
51    4
8     1
13    1
25    3
17    1
58    4
57    4
52    4
38    3
1     1
12    1
45    4
24    3
6     2
```

In [97]: X_test

Out[97]:

	height	width	mass	color_score
26	9.2	9.6	362	0.74
35	7.9	7.1	150	0.75
43	10.3	7.2	194	0.70
28	7.1	6.7	140	0.72
11	7.6	7.1	172	0.92
2	7.2	7.4	176	0.60
34	7.8	7.6	142	0.75
46	10.2	7.3	216	0.71
40	7.5	7.1	154	0.78
22	7.1	7.3	140	0.87
4	4.6	6.0	84	0.79
10	7.3	6.9	166	0.93
30	7.5	7.1	158	0.79
41	8.2	7.6	180	0.79
33	8.1	7.5	190	0.74

In [99]: y_test

Out[99]:

```
26    3
35    3
43    4
28    3
11    1
2     1
34    3
46    4
40    3
22    1
4     2
10    1
30    3
41    3
33    3
```

25% split of training versus test data. And that's a pretty standard relative split that's used. It's a good rule of thumb to use in deciding what proportion of training versus test might be helpful.

As a reminder, when we're using Scikit-learn, we'll denote the data that we have using different flavors of the variable X, capital X, which is typically a two dimensional array or data frame. And the notation we'll use for labels will be typically based on lowercase y, which is usually a one dimensional array, or a scalar. Now, note the use of the random state parameter in the train_test_split function. So this random state parameter provides a seed value to the function's internal random number generator.

If we choose different values for that seed value, that will result in different randomized splits for training and test. So, if we want to get the same training and test split each time, we just make sure to pass in the same value of the random state parameter. And so here, we're going to set that parameter to zero for all our examples. The training test split function will put the training set here into X_train, the test set into X_test, the training labels into y_train, and the test labels into y_test. So this is a 75-25 partitioning of the original data into these two parts.

We'll put the rows of the data without the label, the training instances into this capital X variable, and the list of corresponding labels for those rows into a variable called lowercase y. When using the training set and the test set, we'll then use X_train that holds the training instances to train the classifier, and X_test to evaluate the classifier after it's been trained. And we're going to show a short snippet 작은 정보, 소식 of code here now that illustrates how to do that. So here we can see the results of the function, applying this train_test_split function, you can see that it has indeed split our fruit dataset into training and test sets with the correct proportion of samples.

Some reasons why looking at the data initially is important

- Inspecting feature values may help identify what cleaning or preprocessing still needs to be done once you can see the range or distribution of values that is typical for each attribute.
- You might notice missing or noisy data, or inconsistencies such as the wrong data type being used for a column, incorrect units of measurements for a particular column, or that there aren't enough examples of a particular class.
- You may realize that your problem is actually solvable without machine learning.

Examples of incorrect or missing feature values

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	192
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79
5	2	mandarin	apple	80	5.8	4.3	0.77
6	2	mandarin	mandarin	80	5.9	4.3	0.81
7	2	mandarin	mandarin	76	5.8	4.0	0.81
8	1	apple	braeburn	178	7.1	7.8	0.92
9	1	apple	braeburn	172	7.4	7.0	0.89
10	1	apple	braeburn	166	6.9	7.3	0.93
11	1	apple	braeburn	172	7.1	7.6	0.92
12	1	apple	braeburn	154	7.0	7.1	0.88
13	1	apple	golden_delicious	164	7.3	7.7	0.70
14	1	apple	golden_delicious	152	7.6	7.3	0.69

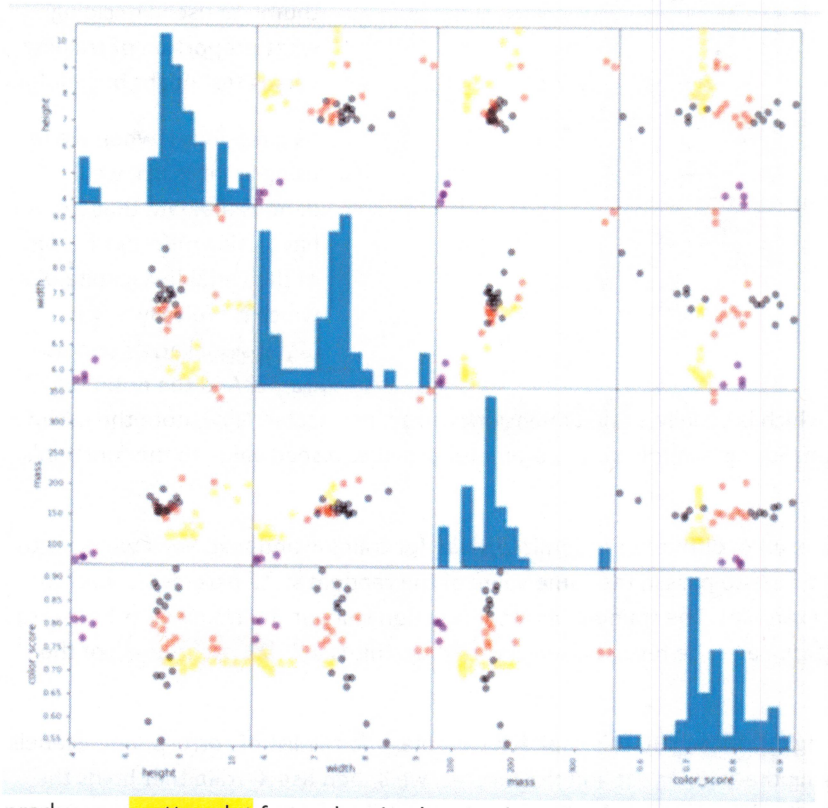
[Examining the Data]

First, it's helpful to simply get a sense for what's actually in the data set. because it may be that in inspecting the features of each object, you might get a better idea of what type of cleaning or preprocessing still needs to be done to the data. And of the range of values or the distribution of values that is typical for each attribute or each feature.

This initial exploration can be especially valuable when you're dealing with complex objects like text that may be represented by many features that are extracted using a series of several pre-processing steps. For example, you might discover that the data set you got has a single column with person's name that still needs to be split into two separate first and last name columns. For example, if you're using the name as one of the prediction feature, that might be important. Second, you might notice missing or noisy data. Or maybe some specific inconsistencies, such as the wrong data type being used for a column. Incorrect or inconsistent units of measurement for a particular column, particular feature. Or maybe you'll notice that there aren't enough examples of a particular labeled class

So now that we have a training set selected, let's create some simple visualizations to look at how the features in the objects in the training set, in our case different fruits, relate to each other and to the labels. So with these visualizations, we get at least two major benefits. First, we can get an idea of the range of values that each feature takes on. And we could immediately see any unusual outliers that are very different from other points. And that might indicate noise or a missing feature or other problem with the data

set. And second, we may be able to get a better idea how likely it is that a machine learning algorithm could do well at predicting the different classes. By seeing how well clustered and well separated the different types of objects are in feature space. So feature space refers to the representation of an object using specific features that are in certain columns of the data that we have.



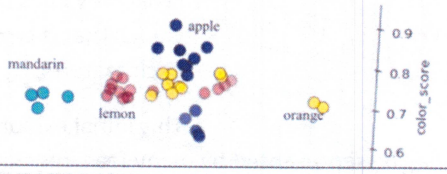
So for example, if the features for objects with the same label, for example, all the lemons have similar feature values, we should see a well-defined cluster appear in the visualization. While if the features for objects that have different labels tend to be quite different, we should see these well separated into visibly different areas of the plot. So having objects whose classes are well defined and well separated in feature space is a good indication that suggests the classifier is likely to be able to predict the class label from the features with good accuracy. Now the visualization techniques that I'll be showing here work well when you have a relatively small number of features, let's say less than 20.

Later, when we cover unsupervised learning, you'll learn how to create visualizations of data sets that use a very large number of feature dimensions, so hundreds or even thousands or millions, to represent each object. But for now, the first visualization tool we'll use is called a **feature pair plot** and that's shown here.

So this plot shows all possible pairs of features and produces a scatter plot for each pair, showing how the features are correlated to each other or not. Each point in the scatter plot represents a piece of fruit, colored according to the class it belongs to. And positioned using the parafeatures assigned to that

```
from matplotlib import cm
cmap = cm.get_cmap('gnuplot')
scatter = pd.scatter_matrix(X_train, c= y_train, marker = 'o', s=40, hist_kwds={'bins':15}, figsize=(12,12), cmap=cmap)
```

scatter plot. Along the diagonal is a histogram showing the distribution of feature values for that feature. So in this pair plot, the dimensions shown here in order are, height, width, mass. And color score of the fruit examples in our training set. So the upper left corner of the histogram here shows the distribution of the height feature for all samples in the training set. And the scatter plot to its immediate right plots the width of each sample on the x-axis and the height of the sample on the y-axis. Just by looking at this pair plot, we can already see that some pairs of features, like the height and color score in the top right corner here, are good for separating out different classes of fruit. And this suggests that a classifier that was trained using those features could likely learn to classify the various fruit types reasonably well. Here's the code that we'll use to create this plot, and let's run this now on our training set. Note that a pair plot like this does not show interactions between all features that might exist, just between pairs of them. So the plot itself may not show all the interesting relationships that do exist between the features. But it does give you a rough idea of some of the interactions that might exist.




```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(X_train['width'], X_train['height'], X_train['color_score'], c = y_train, marker = 'o', s=100)
ax.set_xlabel('width')
ax.set_ylabel('height')
ax.set_zlabel('color_score')
plt.show()
```

Here again, each point represents a single piece of fruit and its color according to its fruit label value. So in this 3D plot, you could actually rotate the plot along the various axes by holding down the mouse button and then dragging. And you can clearly see that the different fruit types are in pretty well-defined clusters that are also well-separated in feature space.

The k-Nearest Neighbor (k-NN) Classifier Algorithm

Given a training set X_train with labels y_train, and given a new instance x_test to be classified:

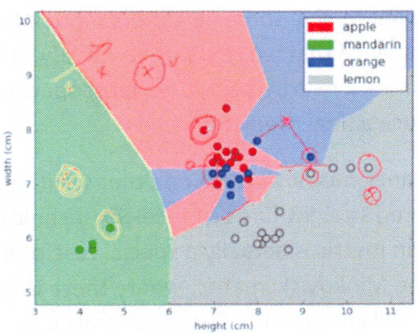
1. Find the most similar instances (let's call them X_NN) to x_test that are in X_train.
2. Get the labels y_NN for the instances in X_NN
3. Predict the label for x_test by combining the labels y_NN e.g. simple majority vote

[KNN] k-Nearest Neighbor Classifier Algorithm

The K-Nearest Neighbors algorithm can be used for classification and regression. Though, here we'll focus for the time being on using it for classification. k-NN classifiers are an example of what's called instance based or memory based supervised learning. What this means is that instance based learning methods work by memorizing the labeled examples that they see in the training set. And then they use those memorized examples to classify new objects later. The k in k-NN refers to the number of nearest neighbors the classifier will retrieve and use in order to make its prediction. In particular, the k-NN algorithm has three steps that can be specified. First of all, when given a new previously unseen instance of something to classify, a k-NN classifier will look into its set of memorized training examples to find the k examples that have closest features.

Second, the classifier will look up the class labels for those k-Nearest Neighbor examples. And then once it's done that, it will combine the labels of those examples to make a prediction for the label of the new object. Typically, for example, by using simple majority vote. [Code explanations.]

A visual explanation of k-NN classifiers



Fruit dataset
Decision boundaries
with k = 1

what I've done here is taken our training set and plotted each piece of fruit using two of its features. The width and the height. These two features together form what is called the feature space of the classifier. And I've shown each data point's label using the colors shown in the legend. There are four different colors here that correspond to the four types of fruit that are in our data set. These broad areas of color show how any point, given a width and height, could be classified according to a one nearest neighbor rule. So, I've set k = 1 here. So, for example, a data point over here that had a height of six centimeters and a width of nine centimeters would be classified as an apple. Because it falls within this red area here that the k-NN classifier has decided map two objects that are apples based on the dimensions. Or for example if I saw that the classifier saw a piece of fruit that had a height of four centimeters and a width of seven centimeters, it would be classified as a mandarin orange. Because it falls within this decision area that's assigned to mandarin oranges.

Let's take a look at the apple example over here. The object that has a height of six centimeters and a width of nine centimeters, if you give the classifier that point, what it will do is search within all the different training examples that it's seen. So it would look at all these points and see which one is closest, so we're looking at the nearest single neighbor. So we're looking at k=1. We're trying to find the one point that's closest to our, what's called the query point. This is the point we want to classify. So, in this case, we can see probably that this point over here is the nearest neighbor of the query point in feature space. Similarly, if we look at the mandarin example over here, the closest training set object to the query point is one of these.

So for any query point in the future space, let's say we're over here in the lemon zone. The nearest neighbor to this query point would be this training set point right here. And because the training set points are labeled, we have the labels for those, in the one nearest neighbor case, the classifier will simply assign to the query point, the class of the nearest neighbor object. Because the nearest neighbor here is an apple, the query point here will get classified as an apple. If we look at the mandarin orange point over here, the nearest data point is the mandarin orange. So it will get classified in that region. And if we do that for every possible query point, we get this pattern of class regions. The zones where we transition from one class to another, this line right here is called the decision boundary. Because query points that are on one side of the line get mapped to one class. And points that are on the other side of the line get mapped to a different class. So, because this is a k-nearest neighbor classifier, and we are looking at the case where k = 1, we can see that the class boundaries here, the decision boundaries.

We have a point over here that's an orange, another point that's a lemon here. And we can see the decision boundary because it's based on Euclidean distance and we're weighting points equally that the decision boundary goes exactly at a distance equidistant to these two things. That's exactly halfway between these two points. Because if it were any closer to this point, it would be mapped to the orange class. And if it were any closer to the lemon instance over here, it would be mapped as lemon class. So you can see this pattern for all the decision boundaries. This line is equidistant to these two nearest points. So this is basically how the K-Nearest Neighbor classifies new instances. So you can imagine with query points where k is larger than 1, we might use a slightly more complicated decision rule. Suppose we have a query point that's in orange. And we're doing two nearest neighbors classifications. So in that case, the two nearest points are these two points here. And in cases where k is bigger than 1, we use a simple majority vote. We take the class that's most predominant in the labels of the neighbor examples. So in this case, the labels happen to agree they're both oranges. And so this point in a two nearest neighbors situation would be classified as an orange.

A nearest neighbor algorithm needs four things specified

1. A distance metric
2. How many 'nearest' neighbors to look at?
3. Optional weighting function on the neighbor points
4. Method for aggregating the classes of neighbor points

When you get points that are closer to a point over here, the two nearest neighbors, in that case, might be these points here. In which case you could choose randomly between them to break the tie. Typically, the value of k is odd, so that we can simply, k equals three it might look like this. And, you can see that the three nearest neighbors here are a red point an apple, a blue point an orange, and another red point an apple. So, in the k = 3 case, the vote would go towards labeling this as an apple.

And that's basically all there is to the basic mechanism for k-NN classifiers. More generally, to use the nearest neighbor algorithm, we specify four things. First, we need to define what distance means in our future space, so we know how to properly select the nearby neighbors. In the example that I just showed you with the fruit data set, we used the simple straight line, or Euclidean distance to measure the distance between points. Second, we must tell the algorithm how many of these nearest neighbors to use in making a prediction. This must be a number that is at least one. Third, we may want to give some neighbors more influence on the outcome. For example, we may decide that neighbors that are closer to the new instance that we're trying to classify, should have more influence, or more votes on the final label. Finally, once we have the labels of the k nearby points, we must specify how to combine them to produce a final prediction.

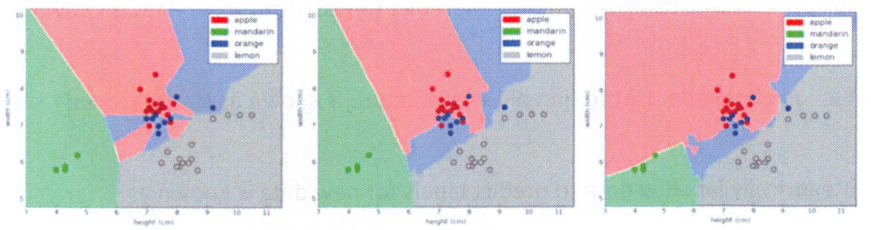
A nearest neighbor algorithm needs four things specified

1. A distance metric
   Typically Euclidean (Minkowski with p = 2)
2. How many 'nearest' neighbors to look at?
   e.g. five
3. Optional weighting function on the neighbor points
   Ignored
4. How to aggregate the classes of neighbor points
   Simple majority vote
   (Class with the most representatives among nearest neighbors)

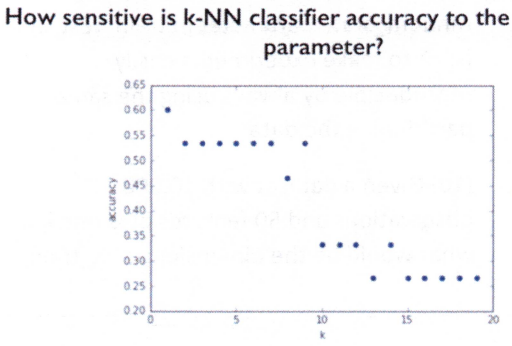
The most common distance metric and the one that scikit-learn uses by default is the Euclidean no straight-line distance. So, technically if you are interested, the Euclidean metric is actually a special case of a more general metric called the Minkowski metric, where there is a parameter p that's set to two that will give you the Euclidean metric. And you'll see this in the notebook, when you look at the actual settings, there done in terms of the Minkowski metric. We might choose to use the five nearest neighbors, for our choice of k. And, we might specify that there's no special treatment for neighbors that are closer, so we have a uniform waiting. Also, by default scikit-learn will apply for the fourth criterion a simple majority vote, and it will predict the class with the most representatives among the nearest neighbors. We'll look at a number of alternative classifiers in next week's class on supervised learning methods. For now, let's take a look at the notebook to see how we apply a k a nearest neighbor classifier in Python to our example fruit data set. As a reminder you can use Shit+tab to show the documentation pop up for the method as you're entering it to help remind you of the different parameter option and syntax. So, recall that our task was to correctly classify new, incoming objects. So, our newly trained classifier will take a data instance as input, and give us a predicted label as output. The first step in Python that we need to take is to load the necessary modules. As well as load the data set into a panda's data frame. After we've done that, we can dump out the first few rows using the head method to check the column names, and get some examples of the first few instances.

Now, what I'm doing next here in the notebook is defining a dictionary that takes a numeric fruit label as the input key. And returns a value that's a string with the name of the fruit, and this dictionary just makes it easier to convert the output of a classifier prediction to something a person can more easily interpret, the name of a fruit in this case. So, for this example we'll define a variable capital X that holds the features of our data set without the label. And here I 'm going to use the mass, width and height of the fruit as features. So, this collection of features is called the feature space. We define a second variable, lower case y, to hold the corresponding labels for the instances in x. Now, we can parse x and y to the train test split function in Scikit-learn. Normally these splitting into training and test sets is done randomly, but for this lecture I want to make sure we all get the same results. So, I set the random state parameter to a specific f=value, in this case I happen to choose zero. [Look at the Code]

The results of the train test split function are put into the four variables you see on the left. And these are marked as x_train, x_test, y_train, and y_test. We're going to be using this x and y variable naming convention throughout the course to refer to data and labels. Once we have our train-test split, we then need to create an instance of the classifier object. And in this case a k-NN classifier. And the set the important parameter in this case the number of neighbors to a specific value to be used by the classifier. We then train the classifier by passing in the training set date in X_train, and the labels in y_train to the classifiers fit method. Now, the k-NN classifier in this case is an example of a more general class called an estimator in scikit-learn. So, all estimators have a fit method that takes the training data, and then changes the state of the classifier, or estimator object to essentially enable prediction once the training is finished. In other words, it updates the state of the k and n variable here, which means, that in the case of k-nners neighbors it will memorize the training set examples in some kind of internal storage for future use. And that's really all there is to training the k-NN classifier, and the first thing we can do with this newly trained classifier is to see how accurate it's likely to be on some new, previously unseen instances. To do this, we can apply they classifier to all the instanced in the test set that we've put aside. Since these test instances were not explicitly included in the classifiers training. One simple way to assess if the classifier is likely to be good at predicting the label of future, previously unseen data instances, is to compute the classifier's accuracy on the test set data items. Remember, that the k-NN classifier did not see any of the fruits in the test set during the training phase. To do this we use the score method for the classifier object. This will take the test set points as input and compute the accuracy. The accuracy is defined as the fraction of test set items, whose true label was correctly predicted by the classifier. We can also use our new classifier to classify individual instances of a fruit. In fact this was our goal in the first place. Was to be able to take individual instances of objects and assign them a label. So, here, for example. I'll enter the mass, width and height for a hypothetical piece of fruit that is fairly small. And if we ask the classifier to predict the label using the predict method. We can see the output is that it predicts it's a mandarin orange. I can then pass a different example, which is maybe a larger, slightly elongated fruit that has a height that's greater than the width and a larger mass. In this case, using the predict method on this instance results in a label that says the classifier thinks this object is a lemon. Now let's use a utility function called plot fruit knn that's included in the shared utilities module that comes with this course. This will produce the colored plots I showed you earlier that have the decision boundaries. You can then try out different values of k for yourself to see what the effect is on the decision boundaries. The uniformed parameter that I pass in here, as the last parameter is the waiting method to be used. So here I'm passing in the string uniform, which means to treat all neighbors equally when combining their labels. If you like, you can try changing this to the word distance, if you want to try a distance wave method.



You can also pass your own function. We can also see how our new classifier behaves for different values of k. So in this series of plots, we can see the different decision boundaries that are produced as k is varied from one to five to ten. We can see that when K has a small value like 1, the classifier is good at learning the classes for individual points in the training set. But with a decision boundary that's fragmented with considerable variation. This is because when K = 1, the prediction is sensitive to noise, outliers, mislabeled data, and other sources of variation in individual data points. For larger values of K, the areas assigned to different classes are smoother and not as fragmented and more robust to noise in the individual points. But possibly with some mistakes, more mistakes in individual points.



Fruit dataset with 75%/25% train-test split

This is an example of what's known as the bias variance tradeoff. And we'll look at that phenomenon and its implications in more depth in next week's class. Given the changes in the classifier's decision boundaries we observed when we changed k, the natural question might be how the value of k, how the choice of k, affects the accuracy of the classifier. We can plot the accuracy as a function of k very easily using this short snippet of code. We see that, indeed, larger values of k do lead to worse accuracy for this particular dataset and fixed single train test split. Keep in mind though, these results are only for this particular training test split. To get a more reliable estimate of likely future accuracy for a particular value of

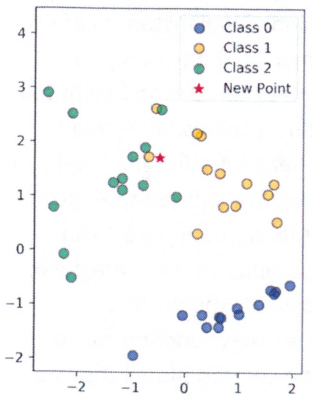
Which of these could be an acceptable sequence of operations using scikit-learn to apply the k-nearest neighbors classification method?

- read_table, train_test_split, fit, KNeighborsClassifier, score
- read_table, train_test_split, KNeighborsClassifier, fit, score

k, we would want to look at results over multiple possible train test splits. We'll go into this issue of model selection next week as well. In general, the best choice of the value of k, that is the one that leads to the highest accuracy, can vary greatly depending on the data set. In general with k-nearest neighbors, using a larger k suppresses the effects of noisy individual labels. But results in classification boundaries that are less detailed.

We've looked at a data set, plotted some features. We then took the features and learned how to compute a train test split. Used that to train a classifier and used the resulting classifier to make some predictions for new objects.

- [x] A low value of "k" (close to 1) is more likely to overfit the training data and lead to worse accuracy on the test data, compared to higher values of "k".
- [x] Setting "k" to the number of points in the training set will result in a classifier that *always* predicts the majority class.
- [x] The k-nearest neighbors classification algorithm has to memorize all of the training examples to make a prediction.



Training a model using labeled data and using this model to predict the labels for new data is known as Supervised Learning.

Modeling the features of an unlabeled dataset to find hidden structure is known as Unsupervised Learning.

Training a model using categorically labelled data to predict labels for new data is known as Classification.

Training a model using labelled data where the labels are continuous quantities to predict labels for new data is known as Regression.

Using the data for classes 0, 1, and 2 plotted below, what class would a KNeighborsClassifier classify the new point as for k = 1 and k = 3? K=1: Class1, k=3: Class2

Which of the following is true for the nearest neighbor classifier (Select all that apply):

- [x] Memorizes the entire training set
- [ ] Partitions observations into k clusters where each observation belongs to the cluster with the nearest mean
- [ ] Given a data instance to classify, computes the probability of each possible class using a statistical model of the input features
- [ ] A higher value of k leads to a more complex decision boundary

(8) The key purpose of splitting the dataset into training and test sets is: → to estimate how well the learned model will generalize to new data.

7. Why is it important to examine your dataset as a first step in applying machine learning? (Select all that apply):

- [x] See what type of cleaning or preprocessing still needs to be done
- [x] You might notice missing data
- [x] Gain insight on what machine learning model might be appropriate, if any
- [x] Get a sense for how difficult the problem might be
- [ ] It is not important

(9) The purpose of setting the random_state parameter in train_test_split is: → to make experiments easily reproducible by always using the same partitioning the data

(10) Given a dataset with 10,000 observations and 50 features plus one label, what would be the dimensions of X_train, y_train, X_test, and y_test? Assume a train/test split of 75%/25%.

- X_train: (7500, 50)
- y_train: (7500,)
- X_test: (2500, 50)
- y_test: (2500,)