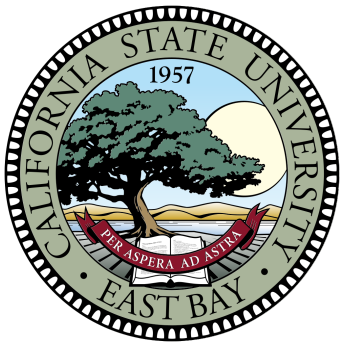


CS 6820 – Machine Learning

Lecture 7 (Includes NumPy ndarrays)

Instructor: Eric S. Gayles, PhD.

Jan 29, 2018

Survey of Machine Learning Algorithms

Supervised Learning

Classification/Regression

- Linear Regression
- Polynomial Regression
- Logistic Regression
- Decision Trees
- Random Forest
- Boosting
- Support Vector Machines
- KNN (K-Nearest Neighbors)
- Neural Networks
- Naïve Bayes

Unsupervised Learning

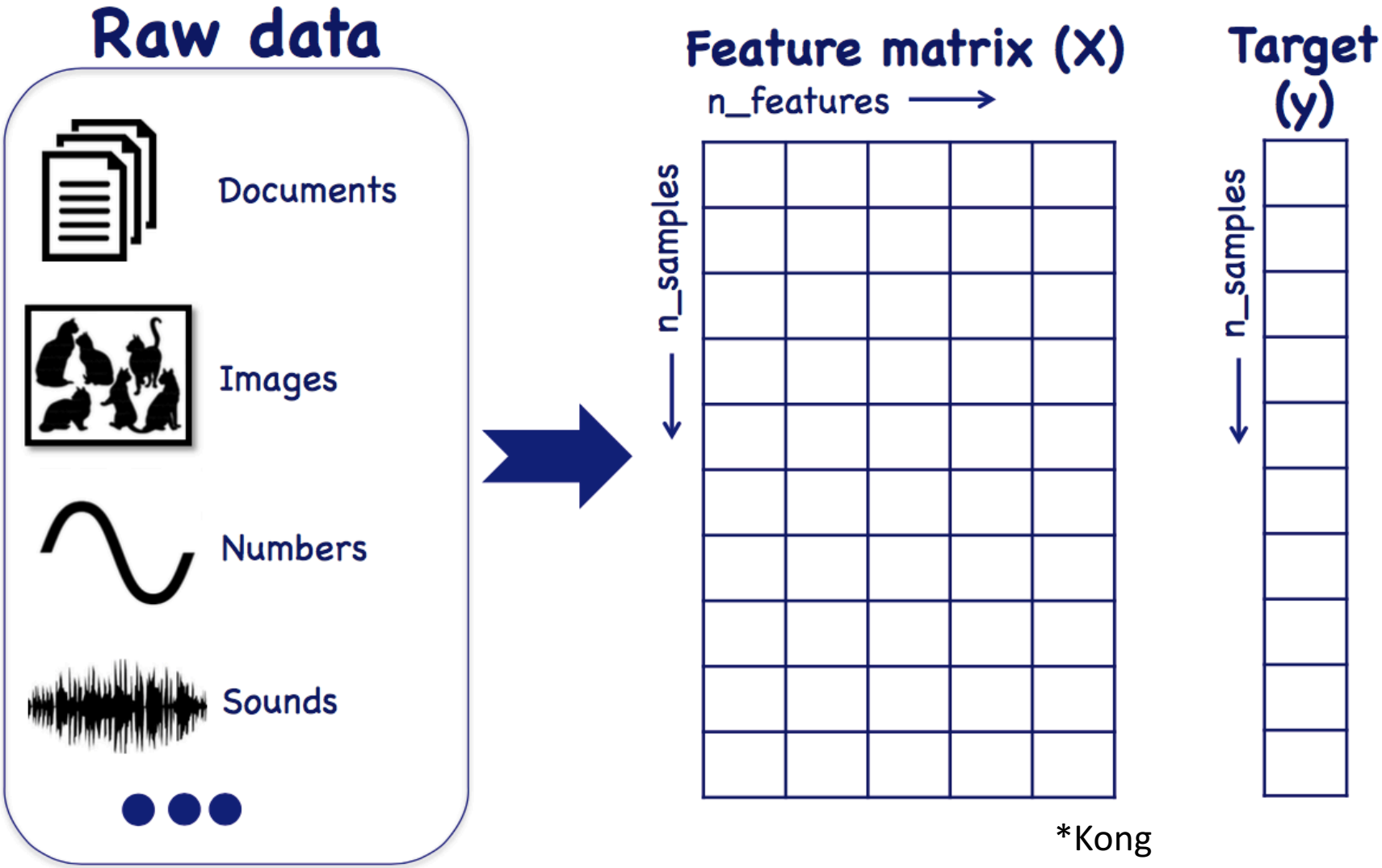
Clustering & Variable reduction

- K-means clustering
- PCA (Principal Component Analysis)

Supporting Techniques

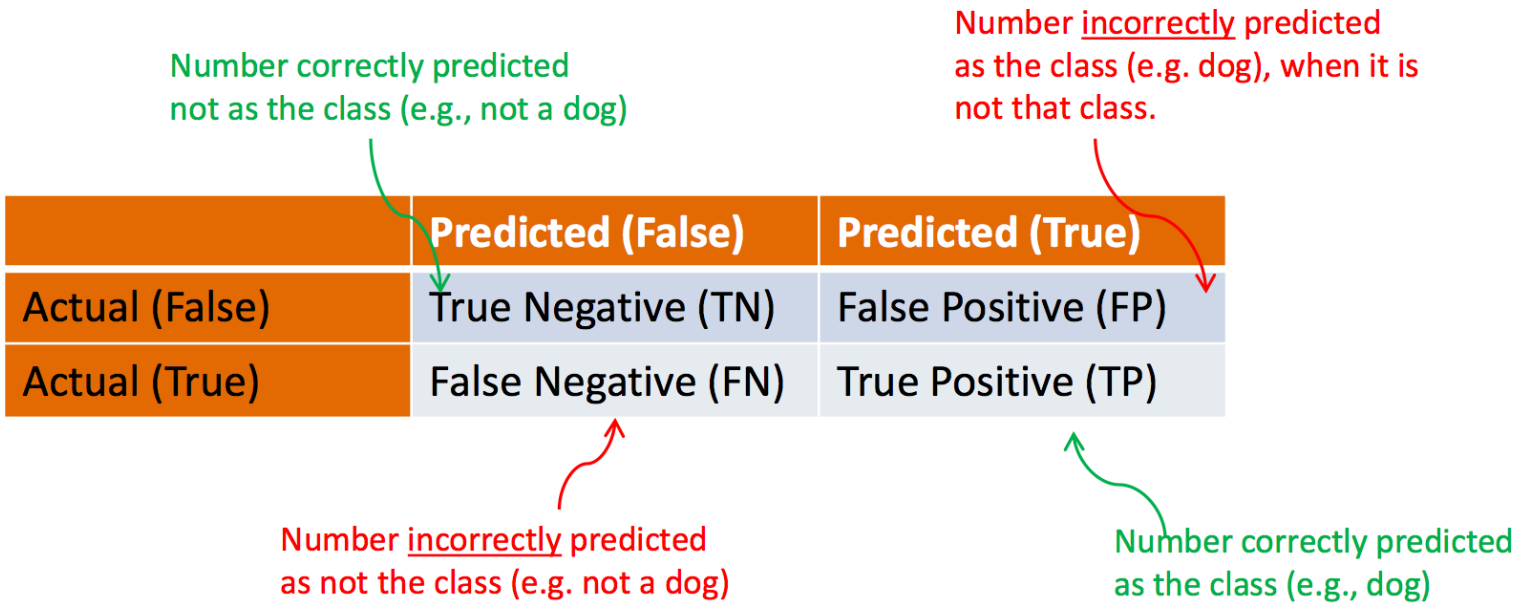
- Cross Validation
- Gradient Descent
- Grid Search

Data Representation



Model Accuracy

- **Four Quadrant Measurement on "Performance" of a model.**



- **Accuracy = (TP + TN) / N**
- **Misclassification = (FP + FN) / N**
- **Precision = TP / (TP + FP)**

Example Measurements

* Andrew Ferlitsch

Feature Scaling

- Feature scaling is a common method in multivariate linear regression.
- If we have many features, there may be a problem: the features are not of the same scale.
- Say for example we have two features,
  - x_1 is in the range in [−1,1]
  - Whereas x_2 is in [−100,100]
  - In terms of the cost function, our parameter θ_1 has to be much bigger and θ_2 has to be much smaller to make cost function zero.
- This may cause convergence to be very slowly.
- Instead, scale the features on a similar scale before implementing gradient descent.
- Example - mean normalization:

$$x_i^* = \frac{x_i - \mu_i}{s_i}$$

Where x^*_i is the i-th scaled feature, μ_i is the average value of x_i in the training set, and s_i can take the range value or the standard variance.

Feature Scaling in Scikit-Learn

In [3]:
```
from sklearn import preprocessing
import numpy as np
X = np.array([
    [0., 0., 5., 13., 9., 1.],
    [0., 0., 13., 15., 10., 15.],
    [0., 3., 15., 2., 0., 11.]
    ])
print (preprocessing.scale(X))
```

```
[[ 0.         -0.70710678 -1.38873015  0.52489066  0.59299945 -1.35873244]
 [ 0.         -0.70710678  0.46291005  0.87481777  0.81537425  1.01904933]
 [ 0.          1.41421356  0.9258201  -1.39970842 -1.4083737   0.33968311]]
```

In [ ]:

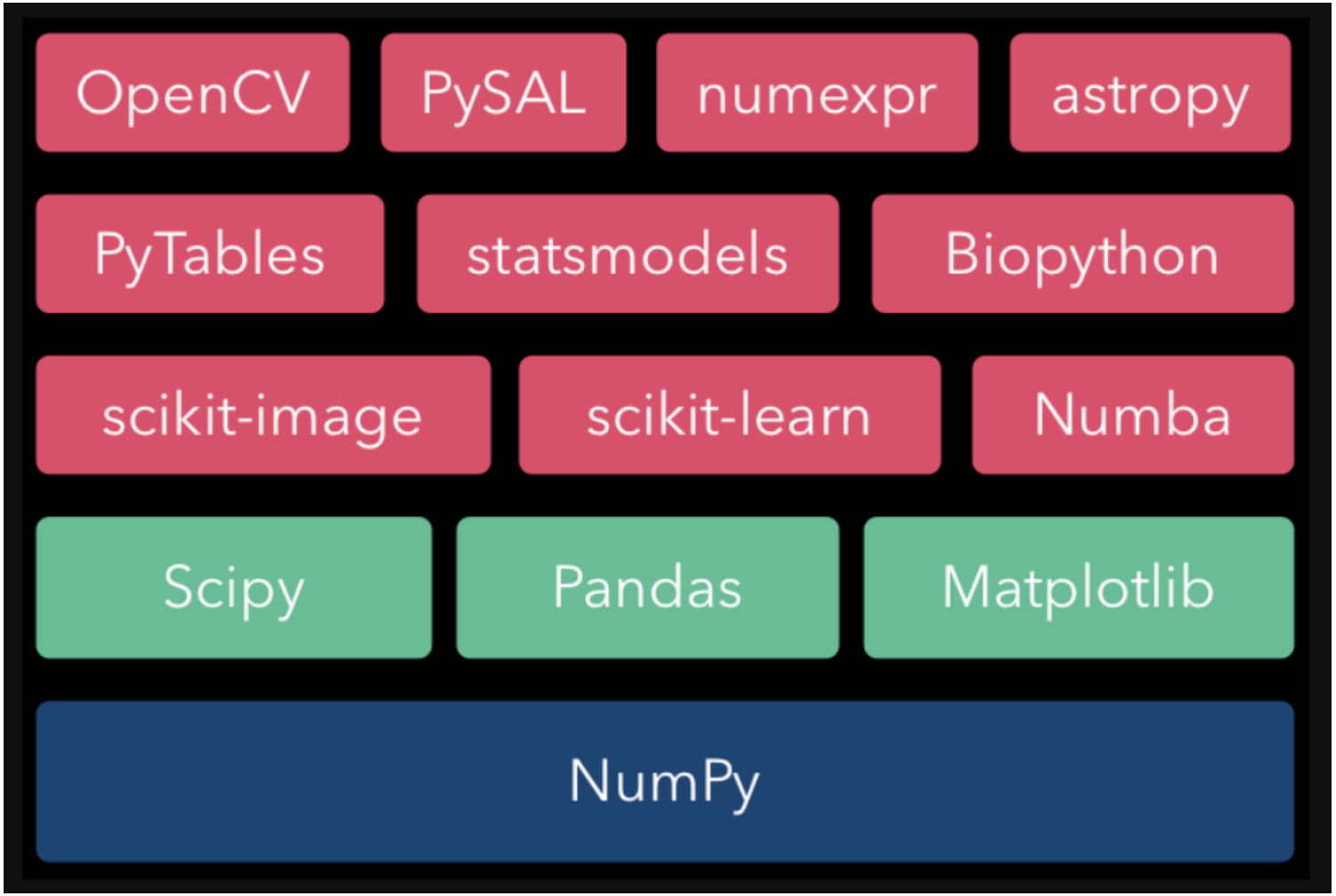
NumPy Arrays

- NumPy arrays are specialized object made more efficient than Python lists when it comes to numerical operations.

- NumPy code requires less explicit loops than equivalent Python code. This is based on vectorization.

- There are certain optimizations using low-level C routines to make many operations more efficient.

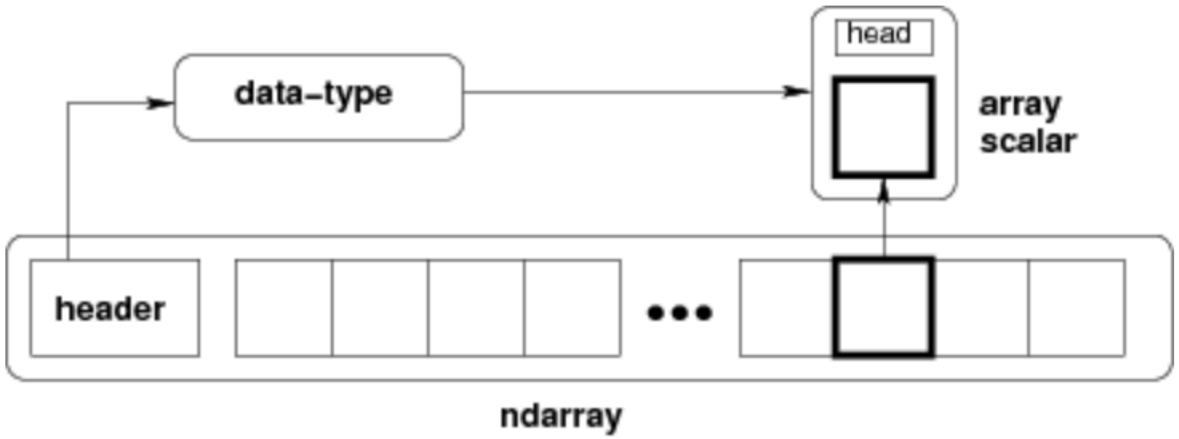
NumPy


OpenCV
PySAL
numexpr
astropy
PyTables
statsmodels
Biopython
scikit-image
scikit-learn
Numba
Scipy
Pandas
Matplotlib
NumPy

NumPy Arrays

- NumPy has a multidimensional array object called ***ndarray (N-Dimensional Array)***.

- It consists of two parts, which are as follows:
  - The actual data
  - Some metadata describing the data



NumPy Arrays

- NumPy array are, in general, homogeneous.
- NumPy arrays can execute vectorized operations, processing a complete array, in contrast to Python lists, where you usually have to loop through the list and execute the operation on each element.
- NumPy arrays are indexed just like in Python, starting from 0.
- Data types are represented by special objects.

```
In [26]: a = np.arange(5)

In [27]: a.dtype

Out[27]: dtype('int64')
```

NumPy Arrays

- This is an array representing a vector.
- The vector has five components with values ranging from 0 to 4.
- The shape property of the array is a tuple; in this instance, a tuple of 1 element, which holds the length in each dimension.

```
In [28]: a
Out[28]: array([0, 1, 2, 3, 4])

In [29]: a.shape
Out[29]: (5,)
```

NumPy Arrays

- Here we make a 2 x 2 array with the arange() subroutine.
- The array() function creates an array from an object that you pass to it.

```
In [31]: m = np.array([np.arange(2), np.arange(2)])

In [32]: m

Out[32]: array([[0, 1],
                [0, 1]])

In [33]: m.shape

Out[33]: (2, 2)
```

NumPy Arrays

- Element Selection

```
In [36]: a

Out[36]: array([[1, 2],
                [3, 4]])

In [37]: a[0,0]

Out[37]: 1

In [38]: a[1,1]

Out[38]: 4
```

NumPy Arrays

- Python supports “Negative Indexing”, meaning that you can write **`x[-1]`** for example.
- Negative index numbers mean that you count from the right instead of the left.
- For example, **`myArray[-1]`** refers to the last element, **`myArray[-2]`** is the second to last, and so on.
- In general, indexes of -x mean the xth item from the end of the list
- So **`myArray[-1]`** means the last item in the array

```
x1[-1]
```

```
9
```

```
x1[-2]
```

```
7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

NumPy Arrays

```
In [56]: a = np.arange(9)

In [57]: # Extract elements
         a[3:7]

Out[57]: array([3, 4, 5, 6])

In [58]: # choose elements from indexes the 0 to 7 with an increment of 2:
         a[:7:2]

Out[58]: array([0, 2, 4, 6])

In [59]: # we can use negative indices and reverse the array:
         a[::-1]

Out[59]: array([8, 7, 6, 5, 4, 3, 2, 1, 0])
```

NumPy Arrays

The `range()` function has two sets of parameters, as follows:

`range(stop)`

- `stop`: Number of integers (whole numbers) to generate, starting from zero. eg. `range(3) == [0, 1, 2]`.

`range([start], stop[, step])`

- `start`: Starting number of the sequence.
- `stop`: Generate numbers up to, but not including this number.
- `step`: Difference between each number in the sequence.

Note that:

- All parameters must be integers.
- All parameters can be positive or negative.
- `range()` (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. eg. The syntax to access the first element of a list is `mylist[0]`. Therefore the last integer generated by `range()` is up to, but not including, `stop`. For example `range(0, 5)` generates integers from 0 up to, but not including, 5.

NumPy Arrays

```
def pythonsum(n):
    a = range(n)
    b = range(n)
    c = []
    for i in range(len(a)):
        a[i] = i ** 2
        b[i] = i ** 3
        c.append(a[i] + b[i])
    return c
```

Does not work in Python 3

```
def numpysum(n):
    a = numpy.arange(n) ** 2
    b = numpy.arange(n) ** 3
    c = a + b
    return c
```

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

NumPy Arrays

```
def pythonsum(n):
    a = list(range(n))
    b = list(range(n))
    c = []
    for i in range(len(a)):
        a[i] = i ** 2
        b[i] = i ** 3
        c.append(a[i] + b[i])
    return c
```

```
def numpysum(n):
    a = np.arange(n) ** 2
    b = np.arange(n) ** 3
    c = a + b
    return c
```

```
import numpy as np

size = int(10)
c = pythonsum(size)
print ("The last 2 elements of the sum", c[-2:])
c = numpysum(size)
print ("The last 2 elements of the sum", c[-2:])
```

```
The last 2 elements of the sum [576, 810]
The last 2 elements of the sum [576 810]
```

NumPy Arrays

- We get the same results whether we are using NumPy or not.
- However, the result that is printed differs in representation.
- Notice that the result from the numpysum() function does not have any commas.
- Why? Because we have been returned a Python list, not a NumPy array.

NumPy Array Manipulation

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

References

- Many of the examples in this lecture relative to NumPy come directly or indirectly from the book below.

- Other examples come from my private Jupyter session

NumPy Numerical Data Types

- Python has an integer type, a float type, and complex type
- These however are not sufficient for scientific calculations.
- For Scientific Computing more data types are needed with varying precisions and, consequently, different storage sizes of the type.
- For this reason, NumPy has many more data types.
- The bulk of the NumPy mathematical types ends with a number.
- This number designates the count of bits related to the type.
- The following table (adapted from the NumPy user guide) presents an overview of NumPy numerical types:

NumPy Numerical Data Types

Type	Description
`bool`	Boolean (`True` or `False`) stored as a bit
`inti`	Platform integer (normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2 ** 31 to 2 ** 31 -1)
`int64`	Integer (-2 ** 63 to 2 ** 63 -1)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 2 ** 32 - 1)
`uint64`	Unsigned integer (0 to 2 ** 64 - 1)
`float16`	Half precision float: sign bit, 5 bits exponent, and 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, and 23 bits mantissa

NumPy Numerical Data Types

Type	Description
`float64` or `float`	Double precision float: sign bit, 11 bits exponent, and 52 bits mantissa
`complex64`	Complex number, represented by two 32-bit floats (real and imaginary components)
`complex128` or `complex`	Complex number, represented by two 64-bit floats (real and imaginary components)

NumPy Numerical Data Types

For each data type, there exists a matching conversion function (look at the `numericaltypes.py` script of this book's code bundle):

```
In: float64(42)
Out: 42.0
In: int8(42.0)
Out: 42
In: bool(42)
Out: True
In: bool(0)
Out: False
In: bool(42.0)
Out: True
In: float(True)
Out: 1.0
In: float(False)
Out: 0.0
```

Many functions have a data type argument, which is frequently optional:

```
In: arange(7, dtype=uint16)
Out: array([0, 1, 2, 3, 4, 5, 6], dtype=uint16)
```

NumPy Numerical Data Types

- **Data type objects** are instances of the numpy.dtype class.
- Once again, arrays have a data type. To be exact, each element in a NumPy array has the same data type.
- The data type object can tell you the size of the data in bytes.
- The size in bytes is given by the **itemsize** property of the **dtype** class

```
In [39]: a.dtype.itemsize

Out[39]: 8
```

NumPy Numerical Data Types

- **Character codes** are included for backward compatibility with **Numeric**. Numeric is the predecessor of NumPy.
- Its use is not recommended, but you will see the code and therefore should be aware
- The following table lists several different data types and character codes related to them:

Type	Character code
integer	`i`
Unsigned integer	`u`
Single precision float	`f`
Double precision float	`d`
bool	`b`
complex	`D`
string	`S`
unicode	`U`
Void	`V`

NumPy Numerical Data Types

Take a look at the following code to produce an array of single precision floats (refer to `charcodes.py` in this book's code bundle):

```
In: arange(7, dtype='f')
Out: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.], dtype=float32)
```

Likewise, this creates an array of complex numbers:

```
In: arange(7, dtype='D')
Out: array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  5.+0.j,
6.+0.j])
```

NumPy Numerical Data Types

```
In [40]: # We have a variety of means to create data types.

In [43]: # We can use the general Python float, as shown in the following lines of code:

In [42]: np.dtype(float)
Out[42]: dtype('float64')

In [44]: # We can specify a single precision float with a character code:

In [46]: np.dtype('f')
Out[46]: dtype('float32')

In [47]: # We can use a double precision float with a character code:

In [48]: np.dtype('d')
Out[48]: dtype('float64')

In [49]: # We can pass the dtype constructor a two-character code.
         # The first character stands for the type;
         # the second character is a number specifying the number of bytes
         #   in the type (the numbers 2, 4, and 8 correspond to floats of
         #   16, 32, and 64 bits, respectively):

In [51]: np.dtype('f8')
Out[51]: dtype('float64')
```

NumPy Numerical Data Types

- The dtype class has a number of useful properties.

- For example, we can get information about the character code of a data type.

```
In [52]: t = np.dtype('Float64')

/Users/ericgayles/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: DeprecationWarning: Numeric-style type codes are deprecated and will result in an error in the future.
  """Entry point for launching an IPython kernel.

In [53]: t.char
Out[53]: 'd'

In [54]: t.type
Out[54]: numpy.float64
```

NumPy Numerical Data Types

- The *data* pointer indicates the memory address of the first byte in the array.
- The *data type* or dtype pointer describes the kind of elements that are contained within the array.
- The *shape* indicates the shape of the array.
- The *strides* are the number of bytes that should be skipped in memory to go to the next element.
  - If your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.

NumPy Arrays

```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

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

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

randint(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).

NumPy Arrays

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
x1[0] = 3.14159  # this will be truncated!
x1
```

```
array([3, 0, 3, 3, 7, 9])
```

NumPy Arrays

```
x2
```

```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

```
x2[0, 0]
```

```
3
```

```
x2[2, 0]
```

```
1
```

```
x2[2, -1]
```

```
7
```

Values can also be modified using any of the above index notation:

```
x2[0, 0] = 12
x2
```

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

NumPy Arrays

- Examples of standard matrix operations.

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print x + y
print np.add(x, y)

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print x - y
print np.subtract(x, y)

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print x * y
print np.multiply(x, y)

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print x / y
print np.divide(x, y)

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print np.sqrt(x)
```

NumPy Arrays

- Examples of Matrix and Vector product operations.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print v.dot(w)
print np.dot(v, w)

# Matrix / vector product; both produce the rank 1 array
print x.dot(v)
print np.dot(x, v)

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print x.dot(y)
print np.dot(x, y)
```

NumPy Arrays

- Advanced Matrix Operations.

```
import numpy as np

x = np.array([[1,2],[3,4]])

print np.sum(x)  # Compute sum of all elements; prints "10

print np.sum(x, axis=0)  # Compute sum of each column; pri
print np.sum(x, axis=1)  # Compute sum of each row; prints
```

```
import numpy as np

x = np.array([[1,2], [3,4]])
print x    # Prints "[[1 2]
           #          [3 4]]"
print x.T  # Prints "[[1 3]
           #          [2 4]]"

# Note that taking the transpose of a rank 1 array does no
v = np.array([1,2,3])

print v    # Prints "[1 2 3]"
print v.T  # Prints "[1 2 3]"
```

NumPy Arrays

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

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

If any of these are unspecified, they default to the values `start=0` , `stop= size of dimension` , `step=1` . We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

NumPy Arrays

- One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:)

```
print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

```
print(x2[0, :])  # first row of x2
```

```
[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
print(x2[0])  # equivalent to x2[0, :]
```

```
[12  5  2  4]
```

NumPy Arrays

```
x = np.arange(10)
x
```

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

```
x[:5]  # first five elements
```

```
array([0, 1, 2, 3, 4])
```

```
x[5:]  # elements after index 5
```

```
array([5, 6, 7, 8, 9])
```

```
x[4:7]  # middle sub-array
```

```
array([4, 5, 6])
```

```
x[::2]  # every other element
```

```
array([0, 2, 4, 6, 8])
```

```
x[1::2]  # every other element, starting at index 1
```

```
array([1, 3, 5, 7, 9])
```

NumPy Arrays

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
x[::-1]  # all elements, reversed
```

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

```
x[5::-2]  # reversed every other from index 5
```

```
array([5, 3, 1])
```

NumPy Arrays

```
x2
```

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

```
x2[:2, :3]  # two rows, three columns
```

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

```
x2[:3, ::2]  # all rows, every other column
```

```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

NumPy Arrays

- Subarray dimensions can even be reversed together:

```
x2[::-1, ::-1]
```

```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

NumPy Arrays

- One important–and extremely useful–thing to know about array slices is that they return views rather than copies of the array data.

- This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies.

```
print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Let's extract a 2×2 subarray from this:

```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

Now if we modify this subarray, we'll see that the original array is changed!

Observe:

```
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

NumPy Arrays

- Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray.

- This can be most easily done with the copy() method:

NumPy Arrays

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

NumPy Arrays

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

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
x = np.array([1, 2, 3])

# row vector via reshape
x.reshape((1, 3))
```

```
array([[1, 2, 3]])
```

```
# row vector via newaxis
x[np.newaxis, :]
```

```
array([[1, 2, 3]])
```

```
# column vector via reshape
x.reshape((3, 1))
```

```
array([[1],
       [2],
       [3]])
```

```
# column vector via newaxis
x[:, np.newaxis]
```

```
array([[1],
       [2],
       [3]])
```

NumPy Arrays

- Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines np.concatenate, np.vstack, and np.hstack.

- np.concatenate takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

```
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

NumPy Arrays

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

NumPy Arrays

- For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack), np.hstack (horizontal stack), np.dstack functions:

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

NumPy Arrays

- The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, np.vsplit, and np.dsplit.

- For each of these, we can pass a list of indices giving the split points:

NumPy Arrays

The function column_stack stacks 1D arrays as columns into a 2D array. It is equivalent to vstack only for 1D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))   # With 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([2.,8.])
>>> a[:,newaxis]  # This allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  2.],
       [ 2.,  8.]])
>>> np.vstack((a[:,newaxis],b[:,newaxis])) # The behavior of vstack is different
array([[ 4.],
       [ 2.],
       [ 2.],
       [ 8.]])
```

NumPy Arrays

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

NumPy Arrays

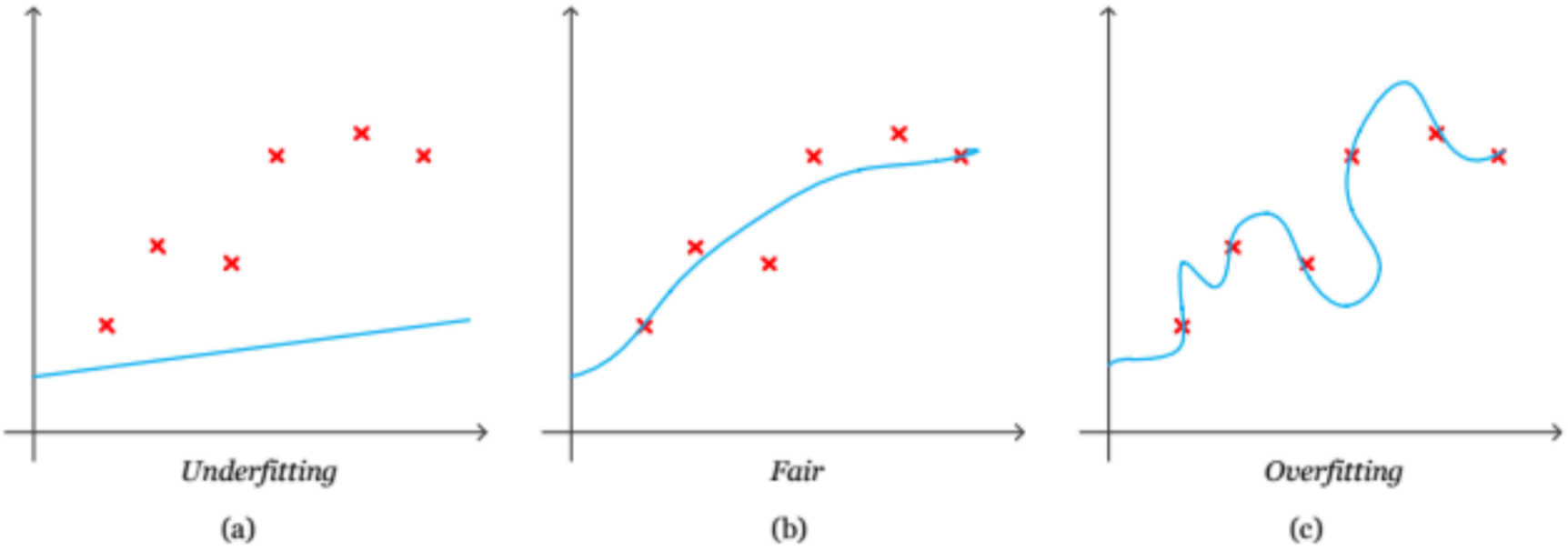
```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Model Quality



Regularization

- Regularization is a collection of techniques that can be used to prevent over-fitting.
- Regularization adds information to a problem, often in the form of a penalty against complexity, to a problem.
- Occam's razor states that a hypothesis with the fewest assumptions is the best.
- Accordingly, regularization attempts to find the simplest model that explains the data.

Regularization

- We have two options to reduce overfitting
  - Build a simple model.
  - Drop all polynomial terms.
    - But this might lead to underfitting

- Or keep the complex terms but give them less weightage.
  - This will take care of overfitting
  - Instead of minimising SSE alone, minimise both SSE and weights
  - The regularization term Imposes some penalty on weights. It impacts the overall weights and reduces the overfitting

Regularization

- **Ridge regression**, also known as **Tikhonov regularization**, penalizes model parameters that become too large.

- Ridge regression modifies the residual sum of the squares cost function by adding the L2 norm of the coefficients, as follows:

$$RSS_{\text{ridge}} = \sum_{i=1}^{n} \left(y_i - x_i^T \beta \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

RSS - Residual sum of squares

Regularization

- λ is a hyperparameter that controls the strength of the penalty.
- **Hyperparameters** are parameters of the model that are not learned automatically and must be set manually.
- As λ increases, the penalty increases, and the value of the cost function increases.
- When λ is equal to zero, ridge regression is equal to linear regression.

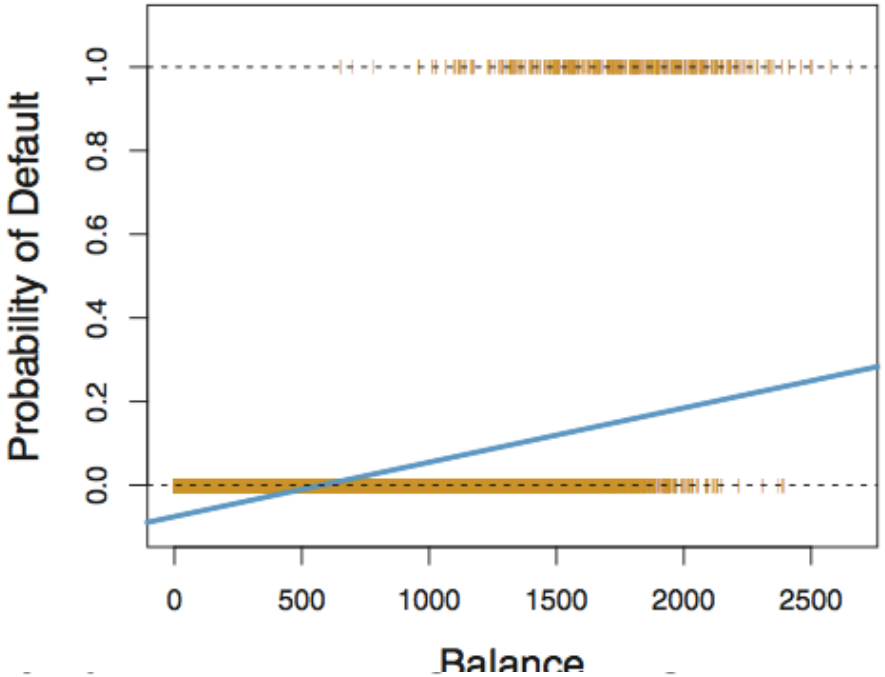
Regularization

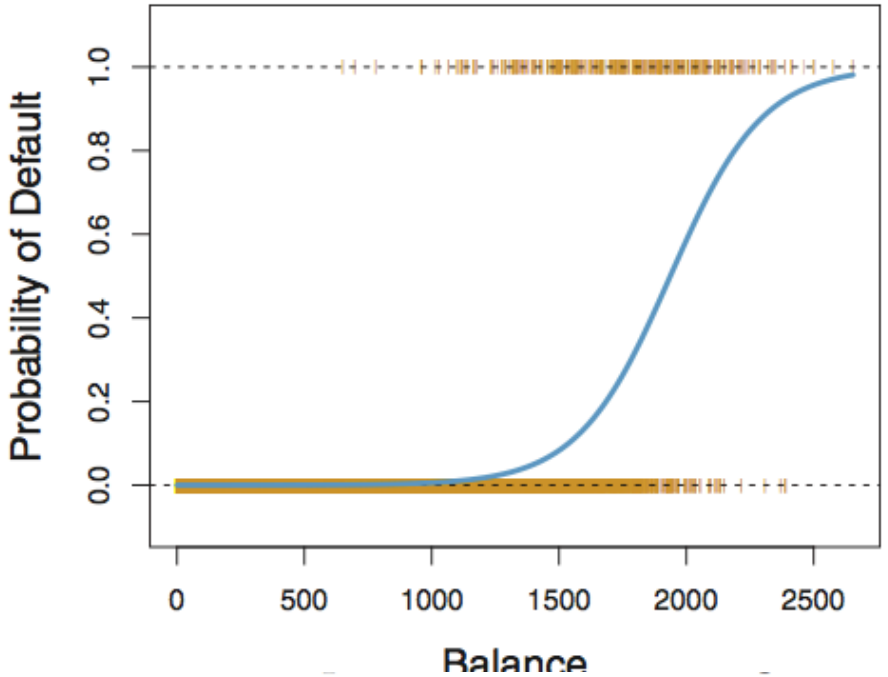
- scikit-learn also provides an implementation of the **Least Absolute Shrinkage and Selection Operator** (**LASSO**).

- LASSO penalizes the coefficients by adding their L1 norm to the cost function

Logistic Regression

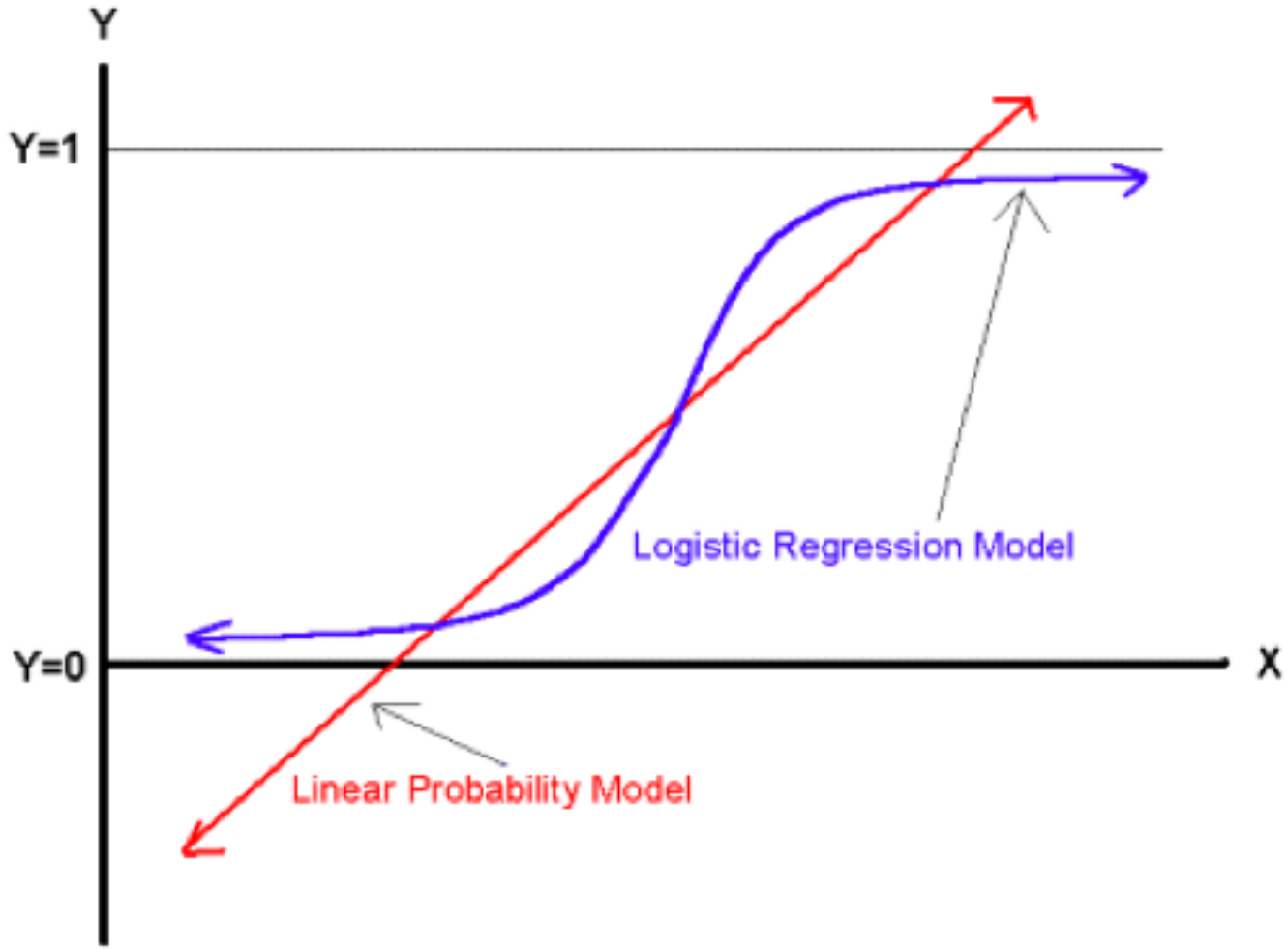
- Logistic Regression models the probability that a sample Y belongs to a particular category.





The orange ticks indicate the 0/1 values coded for `default` *(*`No` *or* `Yes`*). Right: Predicted probabilities of* `default` *using logistic regression. All probabilities lie between 0 and 1.*

Logistic Function



Logistic Function

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}}$$

Logistic Regression

- Once the coefficients have been estimated, it is a simple matter to compute the probability of default for any given credit card balance.

- For example, using the following coefficient estimates we predict that the default probability for an individual with a balance of $1, 000 is

$$\hat{p}(X) = \frac{e^{\hat{\beta}_0+\hat{\beta}_1 X}}{1+e^{\hat{\beta}_0+\hat{\beta}_1 X}} = \frac{e^{-10.6513+0.0055\times 1,000}}{1+e^{-10.6513+0.0055\times 1,000}} = 0.00576$$

Odds & Odds Ratios

- The definitions of an **odds**:

$$odds = \frac{p}{1-p}$$

- The odds has a range of 0 to ∞
- Values greater than 1 associated with an event are more likely to occur than to not occur.
- Values less than 1 associated with an event that is less likely to occur than not occur.

Logistic Regression

- For example, 1 in 5 people with an odds of 1/4 will default, since p(X) = 0.2 implies an odds of 0.2 = 1/4.

- Likewise on average nine out of every ten people with 1−0.2 an odds of 9 will default

$$\frac{p(X)}{1-p(X)} = e^{\beta_0+\beta_1 X}$$

Logistic Regression

- By taking the logarithm of both sides of we arrive at the equation below
- The left-hand side is called the log-odds or logit.
- The logistic regression model has a logit that is linear in X.
- In Linear Regression, β1 gives the average change in Y associated with a one-unit increase in X.
- In contrast, in a logistic regression model, increasing X by one unit changes the log odds by β1, or equivalently it multiplies the odds by $e^{\beta 1.}$

$$ln\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X$$

Logistic Regression

- The coefficients β0 and β1 in are unknown, and must be estimated based on the available training data.
- Previously we used the least squares approach to estimate the unknown linear regression coefficients.
- We could use (non-linear) least squares to fit the model …
- However the the more general method of maximum likelihood is preferred, since it has better statistical properties.

Logistic Regression

- Basic intuition behind using maximum likelihood
- We seek estimates for β0 and β1 such that the predicted probability of default for each individual corresponds as closely as possible to the individual's observed default status.
- In other words, we try to find β_0 and β_1 such that plugging these estimates into the model for p(X), given yields a number close to one for all individuals who defaulted, and a number close to zero for all individuals who did not.
- This intuition can be formalized using a mathematical equation called a likelihood function:

Logistic Regression

- The estimates β_0 and β_1 are chosen to maximize this likelihood function.

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

Logistic Regression

- Consider the problem of predicting a binary response using multiple predictors. We can generalize as follows:

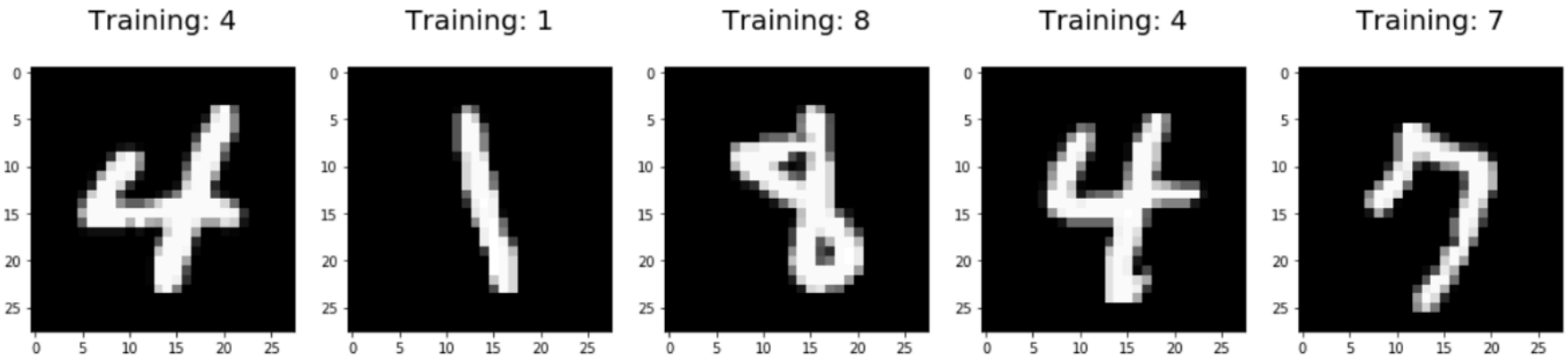
$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}$$

$$ln\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

MNIST Data Set

- The MNIST dataset consists of handwritten digit images and it is divided in 60,000 examples for the training set and 10,000 examples for testing.
- All digit images have been size-normalized and centered in a fixed size image of 28 x 28 pixels.
- In the original dataset each pixel of the image is represented by a value between 0 and 255, where 0 is black, 255 is white and anything in between is a different shade of grey.

Tutorial



Visualizing the Images and Labels in the MNIST Dataset

Scikit-Learn – Four Steps

1. Importing Scikit-learn
2. Importing Scikit-learn's Dataset
3. Organizing Data into Sets
4. Evaluating the Model's Accuracy

Load the Data

```
from sklearn.datasets import load_digits
digits = load_digits()
```

```
# Print to show there are 1797 images (8 by 8 images for a
dimensionality of 64)
print(“Image Data Shape” , digits.data.shape)

# Print to show there are 1797 labels (integers from 0–9)
print("Label Data Shape", digits.target.shape)
```

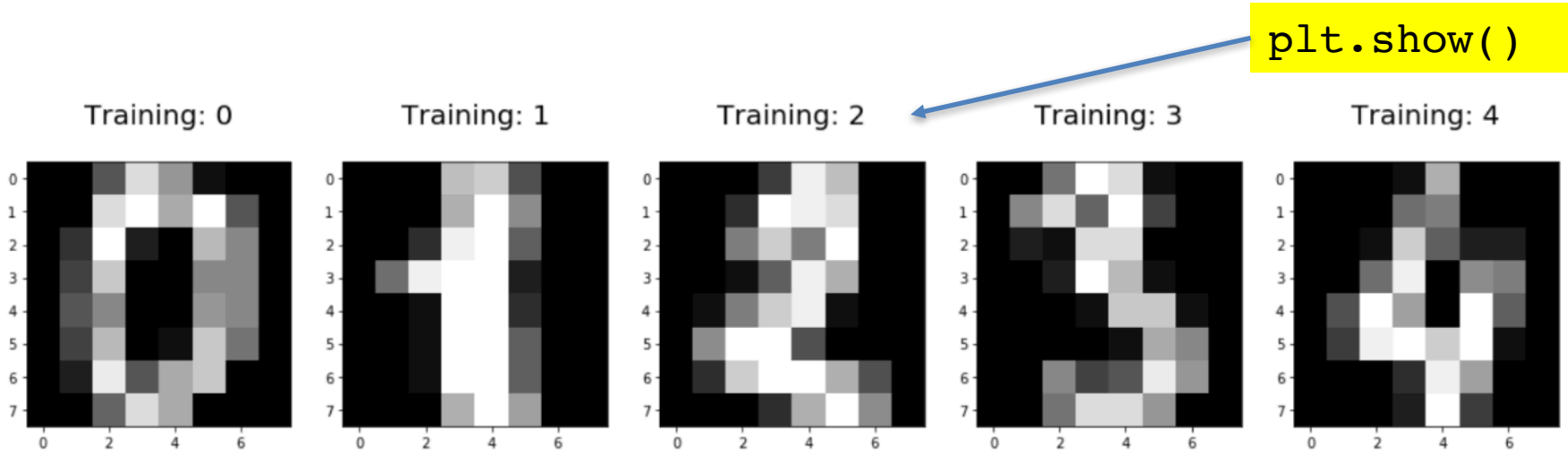
Zip Function in Python

- Make an iterator that aggregates elements from each of the iterables.
- Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables.
- The iterator stops when the shortest input iterable is exhausted.

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
```

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(digits.data[0:5],
digits.target[0:5])):
 plt.subplot(1, 5, index + 1)
 plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
 plt.title('Training: %i\n' % label, fontsize = 20)
```



Visualizing the Images and Labels in our Dataset

Split Train Data from Test Data

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(digits.data,
digits.target, test_size=0.25, random_state=0)
```

```
Out[9]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                verbose=0, warm_start=False)
```

Instantiate the Model

```
from sklearn.linear_model import LogisticRegression
```

```
# all parameters not specified are set to their defaults
logisticRegr = LogisticRegression()
```

Home Installation Documentation Examples

Google Custom Search Search

Previous sklearn.linea.. Next sklearn.linea.. Up API Reference

scikit-learn v0.19.1
Other versions

Please cite us if you use the software.

sklearn.linear_model.LogisticRegression
Examples using sklearn.linear_model.Logistic

sklearn.linear_model.LogisticRegression

class sklearn.linear_model. **LogisticRegression** (*penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1*) [source]

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross- entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag' and 'lbfgs' solvers. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty.

Read more in the User Guide.

Parameters:

penalty : str, 'l1' or 'l2', default: 'l2'

Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties.

New in version 0.19: l1 penalty with SAGA solver (allowing 'multinomial' + L1)

dual : bool, default: False

Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

tol : float, default: 1e-4

Train the Model

```
logisticRegr.fit(x_train, y_train)
```

Predict Labels

```
# Returns a NumPy Array
# Predict for One Observation (image)
logisticRegr.predict(x_test[0].reshape(1,-1))
```

```
logisticRegr.predict(x_test[0:10])
```

```
predictions = logisticRegr.predict(x_test)
```

Evaluate Performance

```
# Use score method to get accuracy of model
score = logisticRegr.score(x_test, y_test)
print(score)
```

Confusion Matrix

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
```

```
cm = metrics.confusion_matrix(y_test, predictions)
print(cm)
```

```
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

What's That Now ...?

- A common method for describing the performance of a classification model consisting of true positives, true negatives, false positives, and false negatives.

- It is called a confusion matrix because it shows how confused the model is between the classes.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model correctly classified 50 apples and 50 oranges.

* Credit for these slides goes to Samuel Bohman of Stockholm University

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model correctly classified 75 cases as not belonging to class apple.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model correctly classified 105 cases as not belonging to class orange.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model correctly classified 115 cases as not belonging to class pear.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model incorrectly classified 15 cases as apples.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model incorrectly classified 10 cases as oranges.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model incorrectly classified 70 cases as pears.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model incorrectly classified 55 cases as not belonging to class apple.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

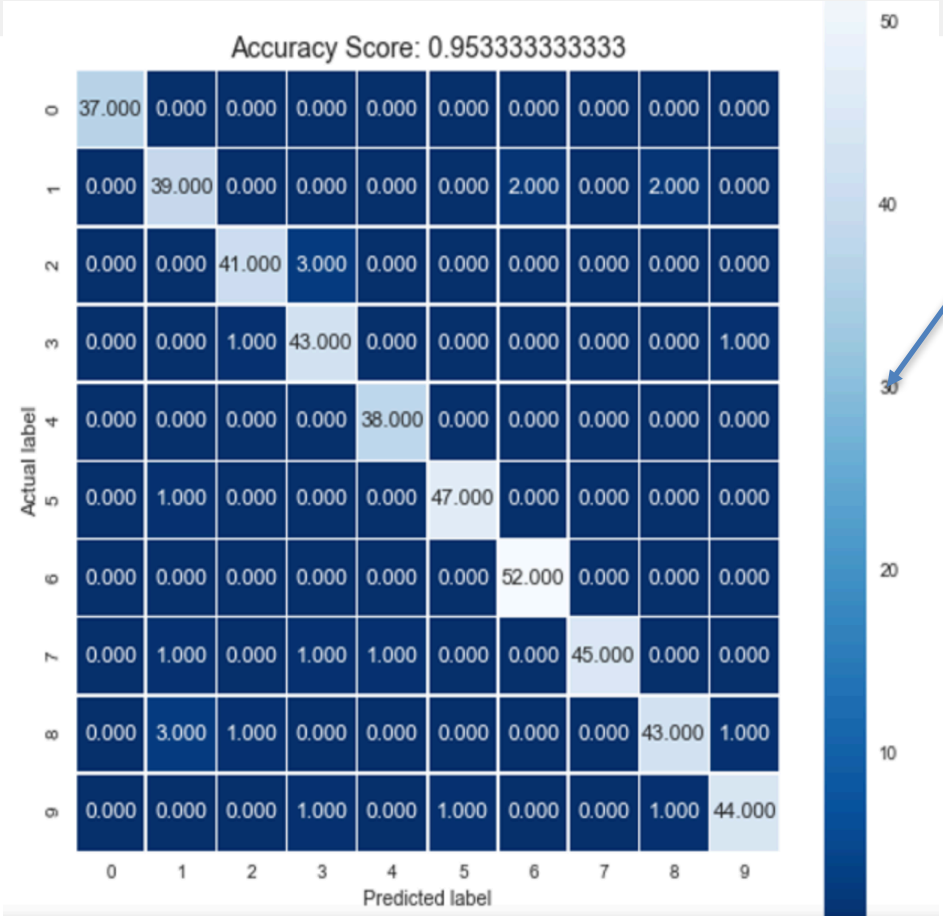
The model incorrectly classified 30 cases as not belonging to class orange.

Confusion Matrix

		Predicted class		
		Apple	Orange	Pear
Actual class	Apple	50	5	50
	Orange	10	50	20
	Pear	5	5	0

The model incorrectly classified 10 cases as not belonging to class pears.

```
plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=True, fmt=".3f", linewidths=.5, square = True,
cmap = 'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(score)
plt.title(all_sample_title, size = 15);
```



plt.show()

Better Example

- The digit dataset contained in sklearn is too small to be representative of a real world machine learning task.
- We are going to use the MNIST dataset to represent a real world example.
- Notice the parameter tuning.

Load the Data

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
```

```
# These are the images
# There are 70,000 images (28 by 28 images for a dimensionality of
784)
print(mnist.data.shape)

# These are the labels
print(mnist.target.shape)
```

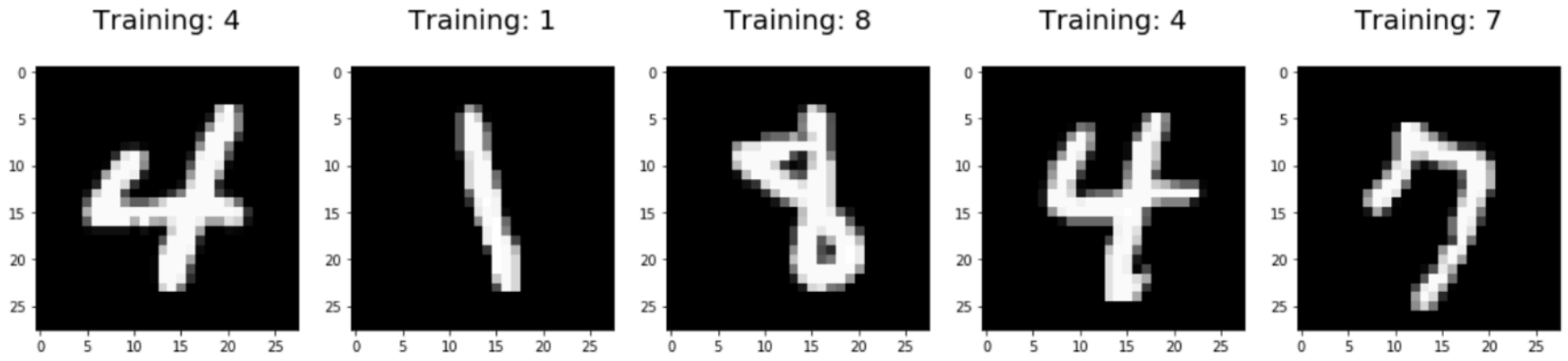
Split the Data

```
from sklearn.model_selection import train_test_split

train_img, test_img, train_lbl, test_lbl = train_test_split(
 mnist.data, mnist.target, test_size=1/7.0, random_state=0)
```

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(train_img[0:5],
train_lbl[0:5])):
 plt.subplot(1, 5, index + 1)
 plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
 plt.title('Training: %i\n' % label, fontsize = 20)
```



Visualizing the Images and Labels in our Dataset

The Model

```
from sklearn.linear_model import LogisticRegression
```

```
# all parameters not specified are set to their defaults
# default solver is incredibly slow thats why we change it
logisticRegr = LogisticRegression(solver = 'lbfgs')
```

Look up Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

Train the Model

```
logisticRegr.fit(train_img, train_lbl)
```