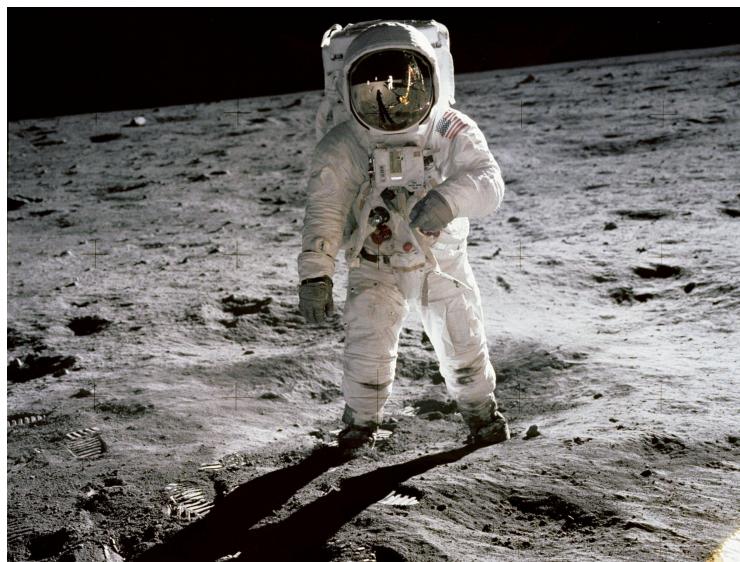


# Labs for Foundations of Applied Mathematics

Volume 3  
Modeling with Uncertainty and Data

Jeffrey Humpherys & Tyler J. Jarvis, managing editors





# List of Contributors

B. Barker	T. Christensen
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Evans	M. Cook
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Evans	M. Cutler
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Grout	R. Dorff
<i>Drake University</i>	<i>Brigham Young University</i>
J. Humpherys	B. Ehler
<i>Brigham Young University</i>	<i>Brigham Young University</i>
T. Jarvis	O. Escobar Rodriguez
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Whitehead	M. Fabiano
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Adams	K. Finlinson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
K. Baldwin	J. Fisher
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Bejarano	R. Flores
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Bennett	R. Fowers
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Berry	A. Frandsen
<i>Brigham Young University</i>	<i>Brigham Young University</i>
Z. Boyd	R. Fuhriman
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Brown	T. Gledhill
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Carr	S. Giddens
<i>Brigham Young University</i>	<i>Brigham Young University</i>
C. Carter	C. Gigena
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. Carter	M. Graham
<i>Brigham Young University</i>	<i>Brigham Young University</i>

F. Glines	J. Leete
<i>Brigham Young University</i>	<i>Brigham Young University</i>
C. Glover	Q. Leishman
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Goodwin	J. Lytle
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Grout	E. Manner
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Grundvig	M. Matsushita
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. Halverson	R. McMurray
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Hannesson	S. McQuarrie
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. Harding	E. Mercer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
K. Harmer	D. Miller
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Henderson	J. Morrise
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Hendricks	M. Morrise
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Henriksen	A. Morrow
<i>Brigham Young University</i>	<i>Brigham Young University</i>
I. Henriksen	J. Murphey
<i>Brigham Young University</i>	<i>Brigham Young University</i>
B. Hepner	R. Murray
<i>Brigham Young University</i>	<i>Brigham Young University</i>
C. Hettinger	J. Nelson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. Horst	C. Noorda
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Howell	A. Oldroyd
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Ibarra-Campos	J. Oliphant
<i>Brigham Young University</i>	<i>Brigham Young University</i>
K. Jacobson	A. Oveson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Jenkins	E. Parkinson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Larsen	M. Probst
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Larsen	M. Proudfoot
<i>Brigham Young University</i>	<i>Brigham Young University</i>

- D. Reber  
*Brigham Young University*
- H. Ringer  
*Brigham Young University*
- C. Robertson  
*Brigham Young University*
- M. Russell  
*Brigham Young University*
- K. Sandall  
*Brigham Young University*
- R. Sandberg  
*Brigham Young University*
- C. Sawyer  
*Brigham Young University*
- N. Schill  
*Brigham Young University*
- N. Sill  
*Brigham Young University*
- D. Smith  
*Brigham Young University*
- J. Smith  
*Brigham Young University*
- P. Smith  
*Brigham Young University*
- M. Stauffer  
*Brigham Young University*
- E. Steadman  
*Brigham Young University*
- J. Stewart  
*Brigham Young University*
- S. Suggs  
*Brigham Young University*
- A. Tate  
*Brigham Young University*
- T. Thompson  
*Brigham Young University*
- B. Trendler  
*Brigham Young University*
- M. Victors  
*Brigham Young University*
- E. Walker  
*Brigham Young University*
- J. Webb  
*Brigham Young University*
- R. Webb  
*Brigham Young University*
- J. West  
*Brigham Young University*
- R. Wonnacott  
*Brigham Young University*
- A. Zaitzeff  
*Brigham Young University*



# Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics Volume 3: Modeling with Uncertainty and Data* by Humpherys and Jarvis. The labs present various aspects of important machine learning algorithms. The reader should be familiar with Python [VD10] and its NumPy [Oli06, ADH<sup>+</sup>01, Oli07] and Matplotlib [Hun07] packages before attempting these labs. See the Python Essentials manual for introductions to these topics.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

<https://github.com/Foundations-of-Applied-Mathematics/Labs>  
as the original source of this work.

To view a copy of the Creative Commons Attribution 3.0 License, visit

<http://creativecommons.org/licenses/by/3.0/us/>  
or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.





# Contents

Preface	v
<b>I      Labs</b>	<b>1</b>
<b>1      Advanced Numpy</b>	<b>3</b>
<b>2      Pandas 1: Introduction</b>	<b>39</b>
<b>3      Pandas 2: Plotting</b>	<b>63</b>
<b>4      Pandas 3: Grouping</b>	<b>77</b>
<b>5      Information Theory</b>	<b>89</b>
<b>6      GeoPandas</b>	<b>97</b>
<b>7      LSI and SkLearn</b>	<b>109</b>
<b>8      K-Means Clustering</b>	<b>123</b>
<b>9      Random Forests</b>	<b>133</b>
<b>10     Data Cleaning</b>	<b>143</b>
<b>11     Intro to Parallel Computing</b>	<b>153</b>
<b>12     Linear Regression</b>	<b>167</b>
<b>13     Logistic Regression</b>	<b>175</b>
<b>14     Naive Bayes</b>	<b>183</b>
<b>15     Apache Spark</b>	<b>191</b>
<b>16     Parallel Programming with MPI</b>	<b>209</b>
<b>17     Web Scraping</b>	<b>219</b>

<b>18</b>	<b>Metropolis Algorithm</b>	<b>233</b>
<b>19</b>	<b>Gibbs Sampling and LDA</b>	<b>243</b>
<b>20</b>	<b>Gaussian Mixture Models</b>	<b>257</b>
<b>21</b>	<b>Discrete Hidden Markov Models</b>	<b>267</b>
<b>22</b>	<b>Speech Recognition using CDHMMs</b>	<b>275</b>
<b>23</b>	<b>Kalman Filter</b>	<b>281</b>
<b>24</b>	<b>ARMA Models</b>	<b>291</b>
<b>25</b>	<b>Non-negative Matrix Factorization Recommender</b>	<b>309</b>
<b>26</b>	<b>Intro to Deep Learning and PyTorch</b>	<b>315</b>
<b>27</b>	<b>Recurrent Neural Networks</b>	<b>335</b>
<b>II</b>	<b>Appendices</b>	<b>347</b>
<b>A</b>	<b>NumPy Visual Guide</b>	<b>349</b>
<b>B</b>	<b>Matplotlib Customization</b>	<b>353</b>
	<b>Bibliography</b>	<b>369</b>

# Part I

# Labs



# 1

# Advanced Numpy

**Lab Objective:** NumPy is a vast library with many useful functions that can be easily forgotten if not used and reviewed. This lab will help you remember some of its functionality that you may have forgotten and give you a few new functions to master.

## NOTE

Some of this lab is review, but there is new material near the end and additional materials beyond that.

## Data Access

### Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access elements of the array with slicing syntax.
>>> x[3]                                # The element at index 3.
3
>>> x[:3]                               # Everything up to index 3 (exclusive).
array([0, 1, 2])
>>> x[3:]                               # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                             # The elements from index 3 to 8.
array([3, 4, 5, 6, 7])
```

```
>>> A = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> A[1, 2]                                # The element at row 1, column 2.
7
>>> A[:, 2:]                               # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

### NOTE

Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. Changes to a copy of an array does not affect the original array, but copying an array uses more time and memory than getting a view.

## Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)      # The integers from 0 to 50 by tens.
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])    # Get the 3rd, 1st, and 4th elements.
>>> x[index]                      # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]                       # Get the 0th and 3rd entries.
array([0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like `<` and `==` to create masks.

```
>>> y = np.arange(10, 20, 2)      # Every other integers from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])
```

```
# Extract the values of 'y' larger than 15.
>>> mask = y > 15                                # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False, True, True], dtype=bool)
>>> y[mask]                                         # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

**Problem 1.** Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the resulting array.

## Array Manipulation

### Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)                               # The integers from 0 to 12 (exclusive).
>>> print(A)
[0 1 2 3 4 5 6 7 8 9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3, 4))                            # The new shape is specified as a tuple.
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2, -1))
array([[0, 1, 2, 3, 4, 5],
       [6, 7, 8, 9, 10, 11]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> A = np.arange(12).reshape((3, 4))
>>> A
array([[0, 1, 2, 3],
```

```
[4,  5,  6,  7],
[8,  9, 10, 11])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                                # Equivalent to A.reshape(A.size)
array([0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                                         # Equivalent to np.transpose(A).
array([[0,  4,  8],
       [1,  5,  9],
       [2,  6, 10],
       [3,  7, 11]])
```

### NOTE

By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into “flat” 1-D arrays. For example, by default an array will have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented to purposefully work well with 1-D “row arrays”.

```
>>> A = np.arange(10).reshape((2, 5))
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Slicing out a column of A still produces a "flat" 1-D array.
>>> x = A[:, 1]                               # All of the rows, column 1.
>>> x
array([1, 6])                                 # Not array([[1],
                                               #           [6]])
>>> x.shape
(2,)
>>> x.ndim
1
```

However, it is occasionally necessary to change a 1-D array into a “column array”. Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((-1, 1))                      # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
```

```
[1],  
[2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

## Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
<code>concatenate()</code>	Join a sequence of arrays along an existing axis
<code>vstack()</code>	Stack arrays in sequence vertically (row wise).
<code>hstack()</code>	Stack arrays in sequence horizontally (column wise).
<code>column_stack()</code>	Stack 1-D or 2-D arrays horizontally (column wise) into a 2-D array.

```
>>> A = np.arange(6).reshape((2, 3))  
>>> B = np.zeros((4, 3))  
  
# vstack() stacks arrays vertically (row-wise).  
>>> np.vstack((A, B, A))  
array([[0.,  1.,  2.], # A  
       [3.,  4.,  5.],  
       [0.,  0.,  0.], # B  
       [0.,  0.,  0.],  
       [0.,  0.,  0.],  
       [0.,  0.,  0.],  
       [0.,  1.,  2.], # A  
       [3.,  4.,  5.]])  
  
>>> A = A.T  
>>> B = B.T  
  
# hstack() stacks arrays horizontally (column-wise).  
>>> np.hstack((A, B, A))  
array([[0.,  3.,  0.,  0.,  0.,  0.,  0.,  3.],  
       [1.,  4.,  0.,  0.,  0.,  0.,  1.,  4.],  
       [2.,  5.,  0.,  0.,  0.,  0.,  2.,  5.]])  
      # A      # B      # A  
  
# column_stack() stacks arrays horizontally, including 1-D arrays.  
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))  
array([[0.,  3.,  0.,  1.,  2.],  
       [1.,  4.,  0.,  1.,  2.],  
       [2.,  5.,  0.,  1.,  2.]])
```

See <http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html> for more array manipulation routines and documentation.

## Working with Dimensions

In many scientific disciplines, arrays of more than 2 dimensions are readily utilized. Numpy's function are designed to work on these larger arrays but sometimes it's necessary to convert arrays to a different dimension. To do this we'll use `np.squeeze()` and `np.vstack`

`np.squeeze()` eliminates any superfluous dimensions in an array. These will be any dimension of value 1 when the shape attribute is called. It does not matter in which position the dimension is located. As a result of this, using `np.squeeze()` on several arrays of different shape can result in arrays of the same shape.

```
# Define 3 arrays of different dimensions
>>> test0 = np.arange(9).reshape(1, 3, 3)
>>> test1 = np.arange(9).reshape(3, 1, 3)
>>> test2 = np.arange(9).reshape(3, 3, 1)

# but all arrays become the same after being squeezed
>>> np.squeeze(test0)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]))

>>> np.squeeze(test1)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]))

>>> np.squeeze(test2)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]))

# even arrays with many extra dimensions reduce to the same matrix
>>> ridiculous = np.arange(9).reshape(1, 1, 1, 1, 3, 3, 1, 1, 1, 1)
>>> np.squeeze(ridiculous)
array([[0.,  1.,  2.],
       [3.,  4.,  5.],
       [6.,  7.,  8.]))

# however arrays that have no 1 value in their shape will remain the same
>>> stoic = np.arange(20).reshape(2, 2, 5)
>>> print(stoic)
array([[[0,  1,  2,  3,  4],
        [5,  6,  7,  8,  9]],
       [[10, 11, 12, 13, 14],
```

```
[15, 16, 17, 18, 19]]))

>>> print(np.squeeze(stoic))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9],
       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]]])
```

`np.dstack()`, on the other hand, performs like the other two stacking function except that it stacks on the third dimensions rather than the first or second. For example, using `np.dstack()` on two matrices of shape  $(3, 3)$  would make a matrix of shape  $(3, 3, 2)$ . If a matrix already has three dimensions or more, `np.dstack()` will only affect the third one (i.e., shapes  $(3, 3, 2, 2)$  with  $(3, 3, 2, 2)$  will create  $(3, 3, 4, 2)$ )

**Problem 2.** Write a function that accepts a list of arrays, squeezes them, pads them with 0's on the "right" and "bottom" so that they're all the same dimensions, and then stacks them along the 3rd dimension. Thus, the arrays in the list can be, individually, any size and dimension. However, you may assume the arrays will all be 2-dimensional once the extra dimensions are squeezed out.

Hint: Use the various stacking commands to pad the inputted arrays appropriately with 0's on the "right" and "bottom" so that they can easily be stacked into the three dimensional array in the end. Again, you may assume all arrays in the list, once squeezed, will be two dimensional arrays.

## Array Broadcasting

As we have seen up to this point, when two arrays have the same shape, performing a binary operation (such as addition or multiplication) between these arrays automatically performs that operation elementwise.

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([4, 5, 6])
>>> x + y
array([5, 7, 9])
```

While this is useful, there are many times in which we would like to perform a binary operation on two arrays with different shapes.

Suppose, for example, that we would like to multiply a scalar value `b` (in other words, a zero-dimensional array) by every element in a 1-dimensional array `a`. We could accomplish this naively using a loop:

```
>>> a = np.array([1, 2, 3])
>>> b = 2.0
>>> z = np.zeros(3)
>>> for i in range(3):
...     z[i] = a[i] * b
```

```
...
>>> z
array([2., 4., 6.])
```

While this approach is intuitive, we can use *array broadcasting* to accomplish this task more efficiently:

```
>>> a = np.array([1, 2, 3])
>>> b = 2.0
>>> z = a * b
>>> z
array([2., 4., 6.])
```

We can think of what happened in this operation as a two step process:

- 1) First, this operation duplicated the value of  $b=2$  into the array  $[2, 2, 2]$ , which matches the dimensions of  $a$ .
- 2) Second, the multiplication  $a * [2, 2, 2]$  was performed elementwise, giving the same result as the looping procedure we used above.

This process is visualized in the following figure:<sup>1</sup>

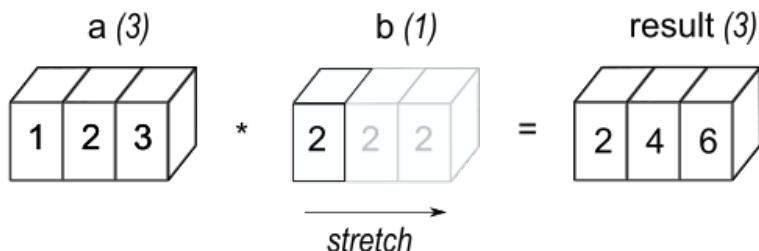


Figure 1.1: Adding a scalar value to a 1-dimensional array using array broadcasting

The beauty of array broadcasting is that this duplication doesn't actually happen in memory, but the results are the same as if it had.

The exact same idea can be extended to arrays of higher dimension by using the following set of rules:

- 1) If the two arrays have different numbers of dimensions, add size-1 dimensions *to the left* of the current dimensions of the smaller array until the number of dimensions matches.
- 2) Stretch any singleton (size-one) dimensions in either of the two arrays to match the size of the corresponding dimension in the other array.
- 3) Add the resulting arrays together elementwise.

---

<sup>1</sup>This and all other diagrams in this section are taken from Numpy's documentation for array broadcasting, which can be found at <https://numpy.org/doc/stable/user/basics.broadcasting.html>

If, at any point, there are corresponding non-singleton dimensions in each array that do not match, broadcasting will raise a `ValueError`: `operands could not be broadcast together` exception.

To illustrate this concept more concretely, consider the following examples.

- **Example 1:** Adding a vector to the rows of a matrix

Suppose, for example, that we have a  $m \times n$  array **a**, and we would like to add the  $n$ -dimensional vector **b** to each row of **a** individually.

We can list the shapes of these arrays as follows:

```
a (2d array): m x n
b (1d array):      n
```

Step (1) of broadcasting will pad the shape of **b** with a size-1 dimension on the left:

```
a (2d array): m x n
b (1d array): 1 x n
```

Notice that the leftmost dimension is a mismatch, with a 1 in **b**'s slot and an  $m$  in **a**'s slot. Thus, step (2) of the broadcasting operation will copy the inner portion of **b**  $m$  times to match the dimension of **a**:

```
a (2d array): m x n
b (1d array): 1 x n -> m x n
```

We can think of this operation as stacking the  $n$ -dimensional row vector **b** vertically with itself  $m$  times.

Finally, step (3) of the broadcasting operation is that these two resulting arrays are added elementwise. The result is the following:

```
>>> a = np.array([[ 0.0,  0.0,  0.0],
...                 [10.0, 10.0, 10.0],
...                 [20.0, 20.0, 20.0],
...                 [30.0, 30.0, 30.0]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```

Thus, each row of **a** is added to **b**, as desired.

A visual representation of this process is given in Figure 1.2:

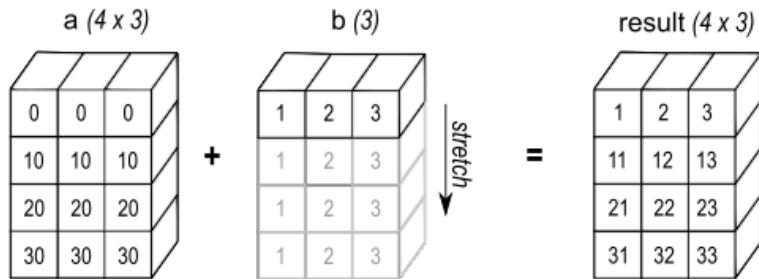


Figure 1.2: Adding entries of vector to the columns of a 2-dimensional array using array broadcasting

- **Example 2:** Adding a vector to the columns of a matrix

As another example, suppose we have the same  $m \times n$  array **a**, and we would like to add the  $m$ -dimensional vector **b** to each of its columns.

Notice that doing this in the same way as the previous example would throw an error:

```
>>> a
array([[ 0.,  0.,  0.],
       [10., 10., 10.],
       [20., 20., 20.],
       [30., 30., 30.]])
>>> b = np.array([1., 2., 3., 4.])
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4, 3) ←
(4,)
```

To understand why this error is being thrown, notice the shapes of these two arrays after step (1) of broadcasting:

a	(2d array): m x n
b	(1d array): 1 x m

The rightmost dimension is a mismatch, but neither dimension is 1. This is the reason the error is thrown. A visual representation of this error is given in Figure 1.3.

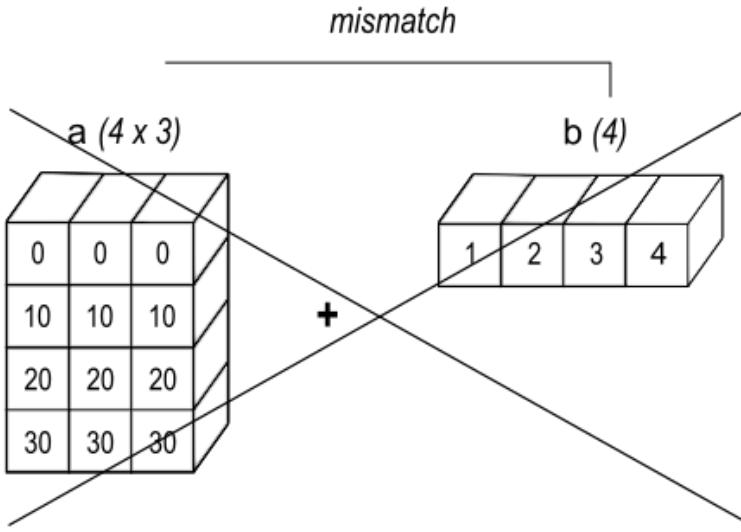


Figure 1.3: The trailing dimensions are a mismatch, so these arrays cannot be broadcast together

In reality, we want to add **b** as a *column vector*; that is, we want to make sure that the  $m$ 's match up in the *leftmost* column. However, as we saw above, we can't rely on array broadcasting to do this automatically for us since it only pads with ones to the *left* (thus, implicitly treating **b** as a row vector).

To accomplish what we want, we must explicitly reshape **b** to be a column vector using a `np.newaxis` argument (or its equivalent). This will pad the shape of **b** with a size-one dimension to the *right* of its normal dimension, which will allow the dimensions to line up appropriately:

<code>a</code>	<code>(2d array): m x n</code>
<code>b[:, np.newaxis]</code>	<code>(1d array): m x 1</code>

The resulting operation is exactly what we want:

```
>>> a
array([[ 0.,  0.,  0.],
       [10., 10., 10.],
       [20., 20., 20.],
       [30., 30., 30.]])
>>> b = np.array([1., 2., 3., 4.])
>>> a + b[:, np.newaxis]
array([[ 1.,  1.,  1.],
       [12., 12., 12.],
       [23., 23., 23.],
       [34., 34., 34.]])
```

- **Example 3:** Outer Sum of Two Vectors

The outer sum of two vectors  $\mathbf{a} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m$  is defined by the matrix  $A \in M_{n \times m}(\mathbb{R})$  such that

$$[A]_{ij} = a_i + b_j, \quad 1 \leq i \leq n, 1 \leq j \leq m$$

We can define this operation by thinking of wanting to make  $m$  copies of  $\mathbf{a}$  (horizontally stacked as column vectors), and add it to  $n$  copies of  $\mathbf{b}$  (horizontally stacked as row vectors).

As it stands, if  $\mathbf{a}$  and  $\mathbf{b}$  are each defined to be one-dimensional arrays with  $n$  and  $m$  elements, respectively, lining up their shapes looks like the following:

<code>a</code>	(1d array):	<code>n</code>
<code>b</code>	(1d array):	<code>m</code>

We want the result to be an  $n \times m$  matrix. To do this, we want to add the elements of  $\mathbf{a}$  along the leftmost dimension (columns), and the elements of  $\mathbf{b}$  along the rightmost dimension (rows). So we use a `np.newaxis` on  $\mathbf{a}$  to make it a column vector:

<code>a[:, np.newaxis]</code>	(2d array):	<code>n x 1</code>
<code>b</code>	(1d array):	<code>m</code>

At this point, broadcasting rules would pad the dimensions of  $\mathbf{b}$  to match the number of dimensions in this `a[:, np.newaxis]`:

<code>a[:, np.newaxis]</code>	(2d array):	<code>n x 1</code>
<code>b</code>	(2d array):	<code>1 x m</code>

Next, the column vector `a[:, np.newaxis]` would repeat itself in  $m$  horizontally-stacked copies, and the row vector  $\mathbf{b}$  would repeat itself in  $n$  vertically-stacked copies, to get matching sizes along each dimension:

<code>a[:, np.newaxis]</code>	(2d array):	<code>n x 1 -&gt; n x m</code>
<code>b</code>	(2d array):	<code>1 x m -&gt; n x m</code>

Finally, the results would be summed elementwise, producing the  $n \times m$  outer-sum matrix  $A$  we desired:

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```

A visual representation of what is occurring here can be seen in Figure 1.4.

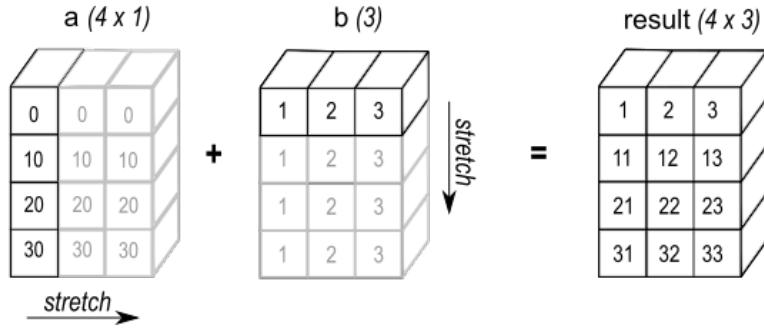


Figure 1.4: Outer sum of two one-dimensional arrays using array broadcasting

Another way we can think about array broadcasting is that we start at the outermost dimension and work our way in, either multiplying intermediate arrays elementwise or broadcasting a single element to match the size of a multi-dimensional element. To see this, let's consider the following two arrays:

$$x = [[1], [2]]$$

$$y = [[3, 4]]$$

To multiply these two arrays together using array broadcasting, we start at the outermost dimension of the arrays and then work our way inward. In this case, the outermost dimension (axis=0) of our arrays is colored in blue, and the inner dimension (axis=1) is colored in orange.

Starting with  $x$ , there are two objects (the two orange arrays) inside of the blue brackets. In  $y$ , there is only one object in the blue brackets (the one orange array). Thus, we will distribute the one orange array in  $y$  to the two orange arrays in  $x$ . For now, we will outline these two arrays with blank arrays as place holders.

$$[], []$$

Performing the first multiplication of orange arrays, there is one element in the orange array from  $x$  (the number 1) and two elements in the orange array from  $y$  (the numbers 3 and 4). We will thus distribute the single element from  $x$  to the two elements of  $y$  by multiplying them elementwise. The result of this is shown below:

$$[[3, 4], []]$$

Performing the second multiplication of orange arrays is similar. The one element of the array from  $x$  (the number 2) is distributed to the two elements from the array in  $y$  (the numbers 3 and 4) and multiplied by each of them individually. The result of this multiplication is shown below:

$$A = \begin{bmatrix} 3, 4 \\ 6, 8 \end{bmatrix}$$

These are some simple examples of array broadcasting. Much more complicated cases come up all the time in scientific computing. However, by lining up dimensions appropriately, many loops can be turned into elementwise broadcasting operations. This, in turn, saves a significant amount of time compared to using a for loop to accomplish the same task.

## Numerical Computing with NumPy

### Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an array.

Function	Description
<code>abs()</code> or <code>absolute()</code>	Calculate the absolute value element-wise.
<code>exp()</code> / <code>log()</code>	Exponential ( $e^x$ ) / natural log element-wise.
<code>maximum()</code> / <code>minimum()</code>	Element-wise maximum / minimum of two arrays.
<code>sqrt()</code>	The positive square-root, element-wise.
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , etc.	Element-wise trigonometric operations.

```
>>> x = np.arange(-2, 3)
>>> print(x, np.abs(x))           # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                  # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.          ,  0.84147098,  0.90929743])
```

**Problem 3.** Write a function that accepts a universal function and an  $n \times n$  NumPy array, and returns how many times as fast it is to operate on the entire array element-wise, rather than by using a nested for loop to operate on each element individually. Run each way of operating on the matrix 10 times, and return the ratio of the averages of the two methods. Vow that you will avoid unnecessary nested for loops, especially when operating on large arrays.

See <http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs> for a more comprehensive list of universal functions.

## ACHTUNG!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.085536923187668

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.          ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

## Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

Method	Returns
<code>all()</code>	<code>True</code> if all elements evaluate to <code>True</code> .
<code>any()</code>	<code>True</code> if any elements evaluate to <code>True</code> .
<code>argmax()</code>	Index of the maximum value.
<code>argmin()</code>	Index of the minimum value.
<code>argsort()</code>	Indices that would sort the array.
<code>clip()</code>	restrict values in an array to fit within a given range
<code>max()</code>	The maximum element of the array.
<code>mean()</code>	The average value of the array.
<code>min()</code>	The minimum element of the array.
<code>roll()</code>	shuffles the elements of the array according to specified amount.
<code>sort()</code>	Return nothing; sort the array in-place.
<code>std()</code>	The standard deviation of the array.
<code>sum()</code>	The sum of the elements of the array.
<code>var()</code>	The variance of the array.

Each of these `np.ndarray` methods has an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed can operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an  $n$ -D array, the return value is an  $(n - 1)$ -D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar.

```
>>> A = np.arange(9).reshape((3, 3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
>>> A.min(axis=0)           # np.array([min(A[:,i]) for i in range(3)])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)          # np.array([sum(A[i,:]) for i in range(3)])
array([3, 12, 21])
```

Similar to the above discussion about array broadcasting, we can color the arrays to gain more visual intuition for how operations can be performed across axes.

Consider again the array

$$A = \begin{bmatrix} 3, 4 \\ 6, 8 \end{bmatrix}$$

Suppose that we wanted to evaluate the call `B = np.sum(A, axis = 1)`. Axis dimensions are numbered from outmost to inmost so in our case axis 0 denotes the blue brackets and axis 1 denotes the orange brackets. Summing over axis 1 means that we sum everything in the orange brackets, but leave the result where it lies in the larger array. So in this case the result would be:

$$B = [7, 14]$$

If we instead wanted to compute the sum `B = np.sum(A, axis = 0)`, we would sum the things inside of the blue brackets. In this case, there are two things inside of the blue brackets (the two orange arrays). These are summed elementwise, resulting in the following array:

$$B = [9, 12]$$

Refer to the NumPy Visual Guide in the appendix for more visual examples.

Also see <http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html> for a more comprehensive list of array methods.

**Problem 4.** A matrix is called *row-stochastic*<sup>a</sup> if its rows each sum to 1. Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function than accepts a matrix (as a 2-D NumPy array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

<sup>a</sup>Similarly, a matrix is called *column-stochastic* if its columns each sum to 1.

## Vectorizing functions

Whenever possible making your functions ‘numpy aware’ can greatly reduce complexity, increase readability and simplicity of code and make functions more versatile. Designing functions to be able to work with and utilize numpy arrays and numpy functions is one of the best ways to optimize code. However, sometimes the functions we need to use are very difficult to vectorize. In this case it can be useful to employ `np.vectorize()`

`np.vectorize()` accepts as an argument a function whose input and output is a scalar. It returns a new function that is ‘numpy aware’, meaning that it will accept a numpy array of values and output an array where each entry had the operation defined by the original function performed on it.

```
# Define a function to double a number
>>> def Double(x):
...     return 2*x

>>> test = np.array([1, 2, 3, 4, 5])
# Using a for loop we can get the doubled array
>>> for i,val in enumerate(test):
...     test[i] *= 2
>>> test
array([2, 4, 6, 8, 10])

# Vectorizing our Double function
>>> DoubleVectorized = np.vectorize(Double)

>>> test = np.array([1, 2, 3, 4, 5])
# with the function vectorized the implementation is simple
>>> DoubleVectorized(test)
array([2, 4, 6, 8, 10])
```

### NOTE

While the above example can easily be done with array broadcasting, `np.vectorize()` can be implemented with very complex scalar functions for which no array broadcasting method exists. However, it should be noted that this function is used only for convenience and readability since it does not improve temporal complexity like normal array broadcasting would. Even though it doesn't improve the complexity, it is often simpler than trying to formulate the for loop.

### ACHTUNG!

Note that `np.vectorize()` will infer the type of its output based on the first element of the input. This behavior can cause confusing results if multiple return types are used within one function. Make sure to either use the same return type in all branches of the function to be vectorized or specify a return type with the `otypes` keyword argument. The following example illustrates the importance of carefully handling datatypes:

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0

>>> f = np.vectorize(half_heavyside)

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0, 0, 0, 0, 0])
```

At first glance, it looks like `np.vectorize()` is simply broken. However, what is actually happening is `np.vectorize()` is inferring the datatype from our first input. Since the first input, -1, returned 0 (an integer), `np.vectorize()` assumes that all subsequent outputs will be of type `int` as well. When `half_heavyside` does return a 0.5, the `float` is cast to an `int`, thus returning a 0.

Two example fixes for this problem are shown below. The first fix explicitly ensures both return statements are of type `float`. The second fix uses the `otypes` keyword argument of `np.vectorize()` to specify the correct return type. Only one type per return value of the wrapped function may be specified with `otypes`.

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0.

>>> f = np.vectorize(half_heavyside)

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0., 0., 0., 0.5, 0.5])
```

```
def half_heavyside(x):
    if (x > 0):
        return 0.5
    else:
        return 0

>>> f = np.vectorize(half_heavyside, otypes = [float])

>>> f([-1., -0.5, 0., 0.5, 1.])
array([0. , 0. , 0. , 0.5, 0.5])
```

**Problem 5.** Given to you is the code that finds the prime factorization of a number and returns the largest prime in the factorization. Vectorize the function using `np.vectorize()` and program a function that either uses the vectorized function or the naive for loop depending on the argument ‘naive’ being passed in as True or False.

Make sure your function returns a numpy array of the same size for both cases.

Hint: Make sure the naive approach returns the array with a dtype of ‘int32’

## Einsum

While numpy has many functions to help multiply arrays, multiplying the elements of arrays in unorthodox ways usually requires the conglomeration of quite a few of these functions. `np.einsum()` is designed to eliminate this problem by making a general framework for multiplication and addition in arrays using their shapes and allowing the coder to tell the function which elements exactly are to be multiplied or summed and how those operations are to be returned.

### Einsteinian summation notation

The function name `einsum()` comes from the term *Einsteinian summation*, which is a standard notational technique in physics and engineering for matrix and tensor operations.

To understand how this works, consider the simple operation of the dot product  $\mathbf{x} \cdot \mathbf{y}$ , where  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ . As we know, this is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i$$

In the equation above, the  $i$  is what we refer to as a *dummy index*; that is,  $i$  is only used inside the sum, and never appears outside the sum.

As another example, consider the operation of matrix multiplication. Given  $A \in M_{n \times m}(\mathbb{R})$  and  $B \in M_{m \times \ell}(\mathbb{R})$ , we know that the  $ij$ -element of the resulting matrix product  $AB \in M_{n \times \ell}(\mathbb{R})$  is given by

$$[AB]_{ij} = \sum_{k=1}^m a_{ik} b_{kj}$$

Again, since  $k$  only appears inside the summation, it is a dummy index. However,  $i$  and  $j$  appear on both the left and right hand sides of this equation, so they are called *free indexes* (and they are not summed over).

Notice that, in both of these examples, we have written vector-valued and matrix-valued operations, respectively, by denoting what an element of the output array should look like in terms of indexes. In the dot product example, the output is a scalar, so we have denoted what the scalar should be in terms of the elements of the input vectors to that operation. In the matrix multiplication example, the output is a matrix, so we have denoted what each  $ij$  element of that matrix should be in terms of the elements of the input matrices to that operation.

In essence, Einsteinian summation notation does exactly this: it denotes what a given element of the output array should be in terms of the elements of the input array, just as we have done above. However, for brevity, all summation symbols are dropped in this notation, and all dummy indexes are implicitly summed over.

In this notation, the dot product would be written

$$\mathbf{x} \cdot \mathbf{y} = x_i y_i$$

Similarly, the matrix product would be written

$$[AB]_{ij} = a_{ik} b_{kj}$$

For this reason, Einsteinian summation notation is often called *index notation*.

Notice that clever, unorthodox ways of taking “products” of matrices and vectors can be defined in this way. For example, given two matrices  $A \in M_{n \times m}(\mathbb{R})$  and  $B \in M_{m \times n}(\mathbb{R})$ , we can define a clever product  $A \odot B$  that (1) takes the normal matrix product and (2) takes the trace of that resulting matrix as

$$A \odot B = \sum_{i=1}^n [AB]_{ii} = \sum_{i=1}^n \sum_{k=1}^m a_{ik} b_{ki}$$

By noting that  $i$  and  $k$  are now dummy variables, and the output is a scalar, we can write this unusual product in index notation as

$$A \odot B = a_{ik} b_{ki}$$

### NOTE

In traditional index notation, the only time an index is summed over is when it appears twice in an expression, such as the index  $i$  in the dot product  $\mathbf{u} \cdot \mathbf{v} = u_i v_i$ . Thus, the only things considered “dummy indexes” are indexes which appear twice in an expression.

However, there are many more expressions that we can represent by summing over indexes which are not repeated. For example, we can take a row sum of a  $m \times n$  matrix  $A$ , resulting in the  $m$ -dimensional vector  $\mathbf{v}$ , using the following rule:

$$[v]_i = \sum_{j=1}^n a_{ij}$$

Note that we are summing over  $j$  here, but it is not a repeated index. So this expression would not be representable in traditional index notation without a much more convoluted expression. Despite this limitation, such an expression *is* representable using `np.einsum()`, as we will describe in the next section. So, while the notation of `np.einsum()` is similar to that of traditional index notation, it is more flexible.

So, through the rest of this manual, we will write mathematical development of these expressions with summation notation, rather than index notation, to make clear exactly what variables are being summed over (that is, to distinguish between free variables and dummy variables).

## Numpy's Einsum

The function `np.einsum()` is a function which is designed to be used with operations defined in terms of Einsteinian summation notation, defined in the previous section. It can be used on arrays of higher dimensions, but we'll keep the scope of this section to working with input arrays of 1 or 2 dimensions.

The numbers in the following syntax only represent positions, each position will be explained below:

```
np.einsum("12,34 -> 56", 7, 8)
```

The meaning of the positions are as follows:

- 1 and 2) Symbols representing the index of an element in the first input array. In the case of 1 dimensional vectors, there will only be one variable and the second will be omitted.
- 3 and 4) Similarly, symbols representing the index of an element in the second input array.
- 5 and 6) Symbols representing the index of an item in the output array. There can zero (for scalars), one (for vectors), two (for matrices), or even three (for higher-order tensors) variables here.
- 7) The first input array. Make sure it is outside the quotation mark ending the variables section and preceded by a comma.
- 8) The second input array.

Positions 7 and 8 are obviously arrays, but position 1-6 are filled with textual symbols, usually  $i$ ,  $j$ , and  $k$ , which represent the indexes of elements in the input and output arrays.

**The key here** is that we can translate anything in summation notation to an einsum expression. To do this, the subscripts of anything *inside* the summation are the letters on the left-hand side of the arrow, with commas separating different quantities. The subscripts of anything on the *result* of a summation notation expression are the letters on the right-hand side of the arrow. This idea will be demonstrated more fully in the following section.

## Einsum Rules

The way `einsum()` interprets its inputted variables are as follows:

- 1) If the variables contained in positions 1 or 2 share a variable in 3 or 4, the values along the axes specified by the repeated variables positions will be multiplied together
- 2) On the other side of the arrow (positions 5 and 6), specify the dimensions of the output. Omitting a variable from these positions causes the products to be summed.  
(That is, any variables that appear on both sides of the arrow are free variables, and any variables only on the left-hand side are dummy variables that are summed over).

As an example, we will define normal matrix multiplication using einsum. Recall that the summation notation for this (rewritten with a different dummy index) is

$$[AB]_{ik} = \sum_{j=1}^m a_{ij}b_{jk}$$

This means that the following `np.einsum()` command will perform normal matrix multiplication:

```
>>> import numpy as np

>>> A = np.eye(3)
>>> A[0,:] += A[2,:]
>>> A
array([[1, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])

>>> B = np.arange(9).reshape((3, 3))
>>> B
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# We use einsum to define normal matrix multiplication
# note the 'j' is repeated in axis 1 and axis 0 respectively
>>> np.einsum("ij,jk -> ik", A, B)
array([[6, 8, 10],
       [3, 4, 5],
       [6, 7, 8]])
```

Since the `j` was repeated in the previous example, the elements on those 2 axes are paired and multiplied together elementwise. The first pair of indices has `j` in the second position, which corresponds to the column position (`axis=1`; moving from column to column down a single row). The second pair of indices has `j` in the first position, which corresponds to the row position (`axis=0`; moving row to row down a single column). Thus, Einsum takes the elementwise products of the elements along each respective row of `A` and column of `B`. This corresponds to the  $a_{ij}b_{jk}$  in the summation notation, above.

Additionally, the output indexes were specified to be `ik` instead of `ijk`. Therefore, since  $j$  was present on the left-hand-side of the Einsum expression, and not the right-hand side, the elementwise products  $a_{ij}b_{jk}$  were summed over all of the  $j$  indexes (exactly as defined in the summation notation above).

It's important to note here that Einsum adds and multiplies in a very similar manner to the `axis` argument in `np.sum()` and other numpy functions.

## Common operations using Einsum

The below table describes some classical operations on vectors and matrices, and how they may be defined in terms of (1) summation notation and (2) the corresponding `np.einsum()` command.

Operation	Summation Notation	Einsum Command
Transpose of a matrix $A$	$[A^T]_{ij} = a_{ji}$	<code>np.einsum("ji -&gt; ij", A)</code>
Row sum vector $r$ of a matrix $A$	$[r]_i = \sum_{j=1}^m a_{ij}$	<code>np.einsum("ij -&gt; i", A)</code>
Column sum vector $c$ of a matrix $A$	$[c]_j = \sum_{i=1}^n a_{ij}$	<code>np.einsum("ij -&gt; j", A)</code>
Sum $s$ of all elements of a matrix $A$	$s = \sum_{i=1}^n \sum_{j=1}^m a_{ij}$	<code>np.einsum("ij-&gt;", A)</code>
Trace of a matrix $A$	$\text{tr}(A) = \sum_{i=1}^n a_{ii}$	<code>np.einsum("ii-&gt;", A)</code>
Dot product of $u$ and $v$	$u \cdot v = \sum_{i=1}^n u_i v_i$	<code>np.einsum("i, i -&gt;", x, y)</code>
Outer product of $u$ and $v$	$[u \otimes v]_{ij} = u_i v_j$	<code>np.einsum("i, j -&gt; ij", u, v)</code>
Matrix product of $A$ and $B$	$[AB]_{ik} = \sum_{j=1}^n a_{ij} b_{jk}$	<code>np.einsum("ij, jk -&gt; ik", A, B)</code>
Elementwise product $B$ of a vector $u$ by each <i>row</i> of a matrix $A$	$[B]_{ij} = a_{ij} u_j$	<code>np.einsum("ij, j -&gt; ij", A, u)</code>
Elementwise product $B$ of a vector $v$ by each <i>column</i> of a matrix $A$	$[B]_{ij} = a_{ij} v_i$	<code>np.einsum("ij, i -&gt; ij", A, v)</code>

### NOTE

Notice again that excluding a letter after the `->` symbol will cause summation of the specified scalar products along the axis whose letter was excluded. On the other hand, including a letter after the `->` symbol will *disallow* summation along that index, and will instead create an output dimension of the same size as the input dimension corresponding to that letter.

For example, see the row sum entry in the above table. Since the row index `i` is included after the `->` symbol, the resulting vector will have the same dimension as the number of rows. However, since the column index `j` is *not* included after the `->` symbol, the scalar values  $a_{ij}$  must be summed over all `j` values corresponding to each `i` value before they are output. This corresponds to summing over all of the columns in the `i`th row before storing the output in the `i`th entry of the output vector.

## A longer example

Now, to demonstrate Einsum's true power we'll show a more complicated example.

Imagine you needed to take an outer product of each column of a matrix  $A \in M_{n,m}(\mathbb{R})$  with the corresponding column of another matrix  $B \in M_{\ell,m}(\mathbb{R})$ , storing the result in an array called `outer_products`, where `outer_products[k]` is defined to be the  $n \times \ell$  outer product array of the  $k$ th column of  $A$  with the  $k$ th column of  $B$  for every  $1 \leq k \leq m$ .

You could accomplish this in a for loop, but it would be slow and not very effective. Conversely, this can be done in one line with `np.einsum`, with the correct mathematical setup.

Let  $\mathbf{u}_k$  being the  $k$ th column of  $A$ , and  $\mathbf{v}_k$  be the corresponding  $k$ th column of  $B$ . We look to find the outer product

$$[\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = [u_k]_i [v_k]_j, \quad 1 \leq i \leq n, 1 \leq j \leq \ell$$

for  $1 \leq k \leq m$ .

We know that the  $i$ th element of  $\mathbf{u}_k$  (the  $k$ th column of  $A$ ) is just  $a_{ik}$ . Similarly, the  $j$ th element of  $\mathbf{v}_k$  (the  $k$ th column of  $B$ ) is just  $b_{jk}$ . Substituting these values into the above expression

$$[\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = a_{ik} b_{jk}, \quad 1 \leq i \leq n, 1 \leq j \leq \ell$$

Finally, letting  $C$  denote the three-dimensional array `outer_products`, we have found that

$$[C]_{kij} = [\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = a_{ik} b_{jk}$$

This is already in index notation, which allows us to write the corresponding Einsum expression in one line:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
# Notice these are just the indexes from the final index notation expression
>>> outer_products = np.einsum("ik, jk -> kij", A, B)
>>> outer_products
array([[[0,  0,  0,  0],
       [0,  9, 18, 27],
       [0, 18, 36, 54]],

      [[1,  4,  7, 10],
       [4, 16, 28, 40],
       [7, 28, 49, 70]],

      [[ 4, 10, 16, 22],
       [10, 25, 40, 55],
       [16, 40, 64, 88]]])
```

The output dimension having three variables creates a 3-dimensional matrix where each element in the first axis is a 3 by 4 outer product matrix.

If we then wanted to sum the rows of each of those outer-product matrices, resulting in a  $k \times n$  array of row sums, notice that we would want an output array  $C$  such that

$$[C]_{ki} = \sum_{j=1}^n [\mathbf{u}_k \otimes \mathbf{v}_k]_{ij} = \sum_{j=1}^n a_{ik} b_{jk}$$

Notice now that  $j$  has become a dummy variable, and thus, we must eliminate it from the right-hand side of the arrow in the Einsum expression:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
>>> outer_products_sum = np.einsum("ik, jk -> ki", A, B)
>>> outer_products_sum
array([[ 0,  54, 108],
       [22,  88, 154],
       [52, 130, 208]])
```

Finally, perhaps we want to multiply a different vector  $\mathbf{v}$  elementwise to the rows of this resulting `outer_products_sum` array. That is (again letting  $[C]_{ki}$  denote the output elements of this output array), we want the  $i$ th element of  $v$  to be multiplied by our earlier expression for  $[C]_{ki}$ :

$$[C]_{ki} = \left( \sum_{j=1}^n a_{ik} b_{jk} \right) v_i = \sum_{j=1}^n a_{ik} b_{jk} v_i$$

The resulting code displays the results:

```
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.arange(12).reshape((4, 3))
>>> v = np.array([0, 1, -1])
>>> outer_products_with_broadcast = np.einsum("ik, jk, i -> ki", A, B, v)
>>> outer_products_with_broadcast
array([[ 0,  54, -108],
       [ 0,  88, -154],
       [ 0, 130, -208]])
```

Thus, three difficult operations have been reduced to a few letters by the power of `np.einsum()`.

## Einsum Optimize

`np.einsum()`, in most cases, performs faster than built in numpy functions. However, the way Einsum organizes its operations creates redundancy when trying to perform multiple operations at once, such as multiplying two matrices, broadcasting a vector to its rows and then summing the resulting matrix columns. One should be cautious when using Einsum to perform multiple operations since you may actually be making your complexity worse rather than improving it.

There are two ways around this problem. First, performing each operation individually will preserve the integrity of Einsum's performance, although brevity of code will suffer. Second, multiple operations can be performed efficiently using kwarg `optimize=True`.

Setting `optimize=True` creates an extra step of operational analysis before any calculations are made to ensure efficient order of operations. In addition, this mode uses a more spatially complex method of computation in exchange for ensured temporal gains. This is why `optimize` defaults to `False`: to allow the programmer to know whether or not there is sufficient memory or need for the optimize functionality.

**Problem 6.** Write a function that accepts 3 vectors and a matrix of appropriate sizes and returns a matrix that is the result of an outer product of the first 2 vectors, the 3rd vector array broadcast multiplied onto the columns of that matrix and then the multiplication via normal matrix multiplication of that result to the inputted matrix. Your function should take a keyword argument `split` that defaults to `False`. If `split=True`, then the einsum operations should be performed one at a time. Otherwise, the operations should be performed all at once while using `optimize=True`.

Write an additional function that performs the same operations only using numpy operations.

Hint: Your result should return the equivalent of `np.outer(x,y)*z.reshape(-1,1)@A` where `x`, `y`, and `z` are vectors and `A` is a matrix.

**Problem 7.** Time your einsum function from Problem 6 versus its numpy function equivalent for vectors of size 3 through 500 and arrays of size (3,3) through (500,500). You should time your einsum function both with and without `split=True`. Plot the results on a neatly formatted and labeled graph. Compare your results to Figure 1.5.

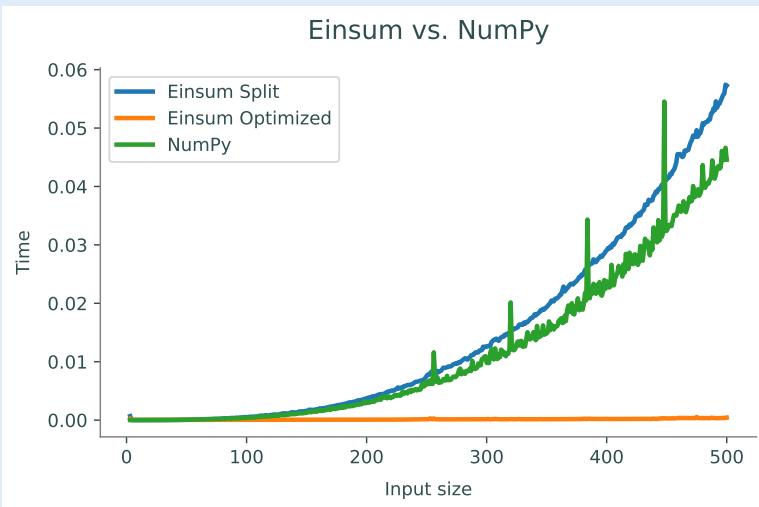


Figure 1.5

## Additional Material

### Random Sampling

The submodule `np.random` holds many functions for creating arrays of random values chosen from probability distributions such as the uniform, normal, and multinomial distributions. It also contains some utility functions for getting non-distributional random samples, such as random integers or random samples from a given array.

Function	Description
<code>choice()</code>	Take random samples from a 1-D array.
<code>random()</code>	Uniformly distributed floats over [0, 1).
<code>randint()</code>	Random integers over a half-open interval.
<code>random_integers()</code>	Random integers over a closed interval.
<code>randn()</code>	Sample from the standard normal distribution.
<code>permutation()</code>	Randomly permute a sequence / generate a random sequence.
Function	Distribution
<code>beta()</code>	Beta distribution over [0, 1].
<code>binomial()</code>	Binomial distribution.
<code>exponential()</code>	Exponential distribution.
<code>gamma()</code>	Gamma distribution.
<code>geometric()</code>	Geometric distribution.
<code>multinomial()</code>	Multivariate generalization of the binomial distribution.
<code>multivariate_normal()</code>	Multivariate generalization of the normal distribution.
<code>normal()</code>	Normal / Gaussian distribution.
<code>poisson()</code>	Poisson distribution.
<code>uniform()</code>	Uniform distribution.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 5 uniformly distributed values in the interval [0, 1].
>>> np.random.random(5)
array([0.21845499, 0.73352537, 0.28064456, 0.66878454, 0.44138609])

# A 2x5 matrix (2-D array) of integers in the interval [10, 20].
>>> np.random.randint(10, 20, (2, 5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

### Saving and Loading Arrays

It is often useful to save an array as a file for later use. NumPy provides several easy methods for saving and loading array data.

Function	Description
<code>save()</code>	Save a single array to a <code>.npy</code> file.
<code>savez()</code>	Save multiple arrays to a <code>.npz</code> file.
<code>savetxt()</code>	Save a single array to a <code>.txt</code> file.
<code>load()</code>	Load and return an array or arrays from a <code>.npy</code> or <code>.npz</code> file.
<code>loadtxt()</code>	Load and return an array from a text file.

```
# Save a 100x100 matrix of uniformly distributed random values.
>>> x = np.random.random((100, 100))
>>> np.save("uniform.npy", x)          # Or np.savetxt("uniform.txt", x).

# Read the array from the file and check that it matches the original.
>>> y = np.load("uniform.npy")        # Or np.loadtxt("uniform.txt").
>>> np.allclose(x, y)                # Check that x and y are close entry-wise.
True
```

To save several arrays to a single file, specify a keyword argument for each array in `np.savez()`. Then `np.load()` will return a dictionary-like object with the keyword parameter names from the save command as the keys.

```
# Save two 100x100 matrices of normally distributed random values.
>>> x = np.random.randn(100, 100)
>>> y = np.random.randn(100, 100)
>>> np.savez("normal.npz", first=x, second=y)

# Read the arrays from the file and check that they match the original.
>>> arrays = np.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```

## Polynomials

The `np.poly1d` object represents a polynomial in NumPy. The constructor is called with the coefficients of the desired polynomial.

```
>>> poly = np.poly1d([3, 5, 1, 2, 0, 1])
>>> print(poly)
      5      4      3      2
3 x + 5 x + 1 x + 2 x + 1
```

The object `poly` represents the polynomial  $3x^5 + 5x^4 + x^3 + 2x^2 + 1$ . NumPy provides many functions to operate on `poly1d` objects (see <http://docs.scipy.org/doc/numpy/reference/routines.polynomials.polynomial.html>).

Recall that

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

The following function evaluates the  $N$ th partial sum of this series at the value  $a$ .

```
>>> from scipy.special import factorial
>>> def exp(a, N=25):
...     """Construct an array in reverse order from n to 0."""
...     n = np.arange(N, -1, -1)
...     # Use broadcasting to compute coefficients
...     coeffs = 1. / factorial(n)
...     poly = np.poly1d(coeffs)          # Make a polynomial object.
...     return poly(a)
...
```

The last two lines can be condensed by using the following command:

```
np.polyval(p, a)
```

## Iterating Through Arrays

Iterating through an array (using a `for` loop) negates most of the advantages of using NumPy. Avoid iterating through arrays as much as possible by using array broadcasting and universal functions. When absolutely necessary, use `np.nditer()` to create an efficient iterator for the array. See <http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html> for details.

## Expressing Einsum Expressions in Terms of Other Functions

Conceptually, we can think of `np.einsum()` as projecting tensors into higher dimensional space, elementwise multiplying the matrices together, and then summing along the specified dimensions. We can thus write many `np.einsum()` expressions as a combination of transposes, elementwise multiplications, sums, and expanding dimensions. Note that writing `np.einsum()` expressions in this way is much less efficient and convenient than just using a call to `np.einsum()`. This is rather used as an alternate way of understanding what `np.einsum()` is doing. We will work through an example of how this is done to illustrate the concepts.

Consider the following `np.einsum()` expression.

```
np.einsum("ijk, jil -> kil", A, B)
```

Each of the different letters in the Einsum expression will label a different axis. Since there are four letters, we project into four dimensions. It doesn't necessarily matter how this axis-to-letter pairing is done, as long as we are consistent, but it's often convenient to do it alphabetically. We will let `i` denote axis 0, `j` denote axis 1, `k` denote axis 2, and `l` denote axis 3.

We can increase the dimensions of the matrices with a call to `np.expand_dims()`. In this case we would call,

```
A = np.expand_dims(A, 4)
B = np.expand_dims(B, 4)
```

We now need to align all the axes of all matrices we are working with. In our example, with our choice of axis-to-letter pairing, the axes of matrix  $A$  are already in the correct alignment. We can order the dimensions of  $B$  with the following call to `np.transpose()`, where the axes passed to `np.transpose` correspond to the axis-to-letter pairing that we denoted earlier:

```
B = np.transpose(B, (1, 0, 3, 2))
```

After correctly ordering the dimensions, we elementwise multiply the matrices together.

```
C = A * B
```

With our final matrix, we sum along all dimensions that are missing from the output specifier. In our case, since  $j$  is missing, we sum along axis 1.

```
C = np.sum(C, axis = 1)
```

Finally, we transpose the result so that the axis labels are consistent with the output specifier. Note that  $k$  has now become axis 1 and  $1$  has become axis 2.

```
C = np.transpose(C, (1, 0, 2))
```

## An Advanced Example of Broadcasting and Einsum

The techniques of array broadcasting and `np.einsum()`, while quite daunting at first, constitute an arsenal of extremely powerful techniques that allow you to design algorithms that run quickly on large quantities of data.

Consider the following real-world example, built off of the machine-learning idea of Gaussian Mixture Models (GMMs), which demonstrates the power of using these techniques in tandem.

Assume you have  $n$  data points  $\{z_0, \dots, z_{n-1}\}$ , where each data point is  $d$ -dimensional (that is,  $z_i \in \mathbb{R}^d$  for  $i = 0, \dots, n - 1$ ).

Underlying the phenomenon we are studying, assume there are  $K$  multivariate normal distributions  $\{\mathcal{N}(\mu_0, \Sigma_0), \dots, \mathcal{N}(\mu_{K-1}, \Sigma_{K-1})\}$ , each of which contribute with some corresponding weight  $\{w_0, \dots, w_{K-1}\}$  to the full probability distribution:

$$P(z|\theta) = \sum_{k=0}^{K-1} w_k \mathcal{N}(z|\mu_k, \Sigma_k)$$

(where  $\mathcal{N}(z|\mu_k, \Sigma_k)$  denotes the standard p.d.f. for the normal distribution with mean  $\mu_k$  and covariance  $\Sigma_k$ ).

Our goal, given the data points  $z_i$ , is to figure out which parameters  $w_k, \mu_k$ , and  $\Sigma_k$  (for  $k = 0, \dots, K - 1$ ) best fit the data.

Skipping some details, the GMM algorithm iteratively updates guesses for these  $3K$  parameters at each timestep  $t$ . Specifically, given (1) an updated mean  $\mu_k^{t+1}$  and (2) some probabilities  $q_i^t(k)$  dependent on (a) the data point  $z_i$ , (b) the distribution index  $k \in [0, \dots, K-1]$ , and (c) the timestep  $t$ , the update step for the covariance matrices is given by:

$$\Sigma_k^{t+1} = \frac{\sum_{i=1}^n q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^n q_i^t(k)}$$

The goal of this section will be to describe how to implement this in Python code.

Assume that you have the following arrays available to you, each having to do with one of the three symbols in the right-hand side of the above expression:

- `q_values`: ( $K, n$ ) array with the elements being  $q_values[k, i] = q_i^t(k)$ .
- `Z`: ( $n, d$ ) array with each row being  $Z[i, :] = z_i$
- `new_means`: ( $K, d$ ) array with each row being  $new_means[k, :] = \mu_k^{t+1}$

We want to store the resulting covariance matrices in an array `new_covars`, which will satisfy  $new\_covars[k, :, :] = \Sigma_k^{t+1}$ . Noting that each covariance matrix will be  $d \times d$ , we can see that `new_covars` will have shape  $(K, d, d)$ .

One naive way to implement this would be to use a nested for loop:

```
>>> K, d = new_means.shape
>>> _, n = q_values.shape
>>> new_covars = np.zeros((K, d, d))
>>> for k in range(K):
...     for i in range(n):
...         centered = Z[i, :] - new_means[k, :]
...         outer_prod = np.outer(centered, centered)
...         new_covars[k, :, :] += q_values[k, i] * outer_prod
...     new_covars[k, :, :] /= np.sum(q_values[k, :])
```

However, this is extremely inefficient, and can take hours to run as the number of data points becomes large. As a result, this operation calls for some more advanced techniques.

Our goal will be to do this entire operation in a series of array broadcasting steps, using only a single Einsum command. This will make the code much more efficient.

To do so, we will start by taking bite-size chunks of the expression, figuring out how to write them in code using broadcasting and Einsum. Then, through a series of steps, we will gradually come up with a formula that will perform the calculation for this expression without a need for any looping whatsoever.

- **Step 1:** (Computing the differences between the means  $\mu_i$  and the data points  $z_k$ )

Notice that the numerator contains the expression

$$(z_i - \mu_k^{t+1})$$

more than once.

As a first step, we look to create an array called `obs_centered` with shape  $(K, n, d)$ , such that  $obs\_centered[k, i, :] = z_i - \mu_k^{t+1}$ .

That is, we want a 3-dimensional array, where each inner element is a 2-dimensional matrix. Each 2-dimensional matrix will be produced by subtracting one row of the `new_means` matrix from all  $n$  rows of  $Z$ .

Recall: if we have two 1-dimensional arrays  $a$  and  $b$ , and we want to take the “outer difference” between them (that is, we want to create a 2-dimensional array containing the differences of every element of  $a$  with every element of  $b$ ), we can do something akin to what is shown in Figure 1.4 and its preceding example:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5])
>>> a - b[:, np.newaxis]
array([[[-3, -2, -1],
       [-4, -3, -2]])
```

Notice that the shapes of the arrays in this broadcast are  $(3,)$  and  $(2,)$ , respectively, and the output shape is  $(2, 3)$ .

Digging deeper into what’s going on here, the broadcasting operation of subtraction (and padding of dimensions) executed the following rearranging of shapes before performing the elementwise subtraction:

<code>a</code>	<code>:</code>	$(3,)$	$\rightarrow$	$(1, 3)$	$\rightarrow$	$(2, 3)$
<code>b</code>	<code>:</code>	$(2,)$	$\rightarrow$	$(2, 1)$	$\rightarrow$	$(2, 3)$

Now, what we seek to do in our current problem is to take a very similar “outer difference” of the *rows* of  $Z$  and `new_means`. This is almost identical to the above case, but instead of subtracting 0-dimensional scalars, we are looking to take the difference between 1-dimensional vectors, each with  $d$  elements.

Therefore, we look to outer-subtract the  $n$  rows of  $Z$  with the  $K$  rows of `new_means` in all possible combinations, in the exact way we looked to outer-subtract the scalars of the input arrays  $a$  and  $b$  in our example above.

The ultimate output we desire is a  $(K, n, d)$  array (similar to how we ended up with a  $(2, 3)$  array in the scalar example).

The code in this case is, therefore, nearly identical to the scalar case. All we need to figure out is how to pad the dimensions of these array in the correct way to get this outer difference of vectors. This is a bit trickier since having a  $d$  in the rightmost slot means we can’t just use `np.newaxis` as easily as before.

Despite this setback, however, looking to the two leftmost dimensions, and leaving the rightmost dimension (the vectors we wish to subtract) alone, comparison to the scalar example shows that we want to achieve the following shapes:

<code>Z</code>	<code>:</code>	$(n, d)$	$\rightarrow$	$(1, n, d)$
<code>new_means</code>	<code>:</code>	$(K, d)$	$\rightarrow$	$(K, 1, d)$

So, what commands do we need to use to achieve these paddings?

The command to add a singleton dimension in the middle of the original shape of `new_means` is `np.expand_dims(new_means, 1)`.

Once that is done, the padding to the left of  $Z$  is automatic by the rules of array broadcasting. So the desired command is:

```
>>> obs_centered = Z - np.expand_dims(new_means, 1)
```

- **Step 2:** (Computing the outer products of these differences with themselves)

Notice that there is a term in the numerator of our desired expression that looks like:

$$(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

This is exactly an outer product of one of the vectors found in Step (1) with itself.

Our goal in this step is to produce an intermediate array `outer_products` of shape with shape  $(K, n, d, d)$  such that  $\text{outer\_products}[k, i, :, :] = (z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$ .

Given that  $\text{obs_centered}[k, i, :] = (z_i - \mu_k^{t+1})$  by our design, we therefore see that `outer_products[k, i, :, :]` is just the outer product of `obs_centered[k, i, :]` with itself.

In other words,

$$\text{outer_products}[k, i, :, :] = \text{obs_centered}[k, i, :] \otimes \text{obs_centered}[k, i, :].$$

It follows that the  $pq$ -element of `outer_products[k, i, :, :]` is equal to the  $pq$ -element of `obs_centered[k, i, :] \otimes obs_centered[k, i, :]`.

Writing this in summation/index notation, letting  $u$  denote `obs_centered` and  $v$  denote `outer_products`, we have shown that

$$v_{kipq} = [v_{ki::}]_{pq} = [u_{ki:} \otimes u_{ki:}]_{pq} = u_{kip} u_{kiq}$$

Therefore, we can write the full einsum expression (with all indexes *inside* the expression) as

```
>>> outer_products = np.einsum("kip, kiq -> kipq", obs_centered, ←
                                obs_centered)
```

- **Step 3:** (Summing the outer products over the dimension representing the number of data points)

Notice that the numerator has a summation (excluding some information) that looks like

$$\sum_{i=1}^n (z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

In this step, we look to create an array `outer_product_sums` with shape  $(K, d, d)$  such that  $\text{outer\_product\_sums}[k, :, :] = \sum_{i=1}^n (z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$ .

Notice that  $(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top = \text{outer_products}[k, i, :, :]$  by our construction.

So, all we are doing is summing along the `i`-axis of `outer_products`.

Therefore, letting  $w$  be a variable representing `outer_product_sums`, with  $u$  representing `obs_centered` and  $v$  representing `outer_products`, we have that

$$w_{kpq} = \sum_{i=1}^n v_{kipq} = \sum_{i=1}^n u_{kip} u_{kiq}$$

Thus, we can write this directly with the following Einsum (which looks nearly identical to Step 2, but excludes the index `i` from the right-hand-side of the arrow to denote a summation):

```
>>> outer_product_sums = np.einsum("kip, kiq -> kpq", obs_centered, ←
    obs_centered)
```

- **Step 4:** (Finishing the numerator)

Finally, we look to tackle the entire numerator in a single Einsum expression.

The numerator looks exactly like the following weighted sum of inner products:

$$\sum_{i=1}^n q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top$$

In this step, we look to create an array `weighted_sum_numerator` with shape `(K, d, d)` such that

$$\text{weighted\_sum\_numerator}[k, :, :] = \sum_{i=1}^n q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top.$$

Recall that  $q_i^t(k) = \text{q\_values}[k, i]$  and  $(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top = \text{outer_products}[k, i, :, :]$  by our construction.

Letting  $q$  represent `q_values` and  $z$  represent `weighted_sum_numerator`, with  $u$  representing `obs_centered` and  $v$  representing `outer_products` yet again, we have that

$$z_{kpq} = \sum_{i=1}^n q_{ki} v_{kipq} = \sum_{i=1}^n q_{ki} u_{kip} u_{kiq}$$

Therefore, we end up with the following Einsum expression for the numerator:

```
>>> weighted_sum_numerator = np.einsum("ki, kip, kiq -> kpq", q_values, ←
    obs_centered, obs_centered)
```

- **Step 5:** (Full expression)

Finally, we look to compute the full expression

$$\Sigma_k^{t+1} = \frac{\sum_{i=1}^n q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^n q_i^t(k)}$$

We have found the numerator to be `weighted_sum_numerator` from Step (4). The denominator is exactly just the sum  $\sum_{i=1}^n q_i^t(k)$ . Notice that this sum depends on  $k$ , as does the result  $\Sigma_k^{t+1}$ . Thus, we must make sure that the sum is taken over all data points, leaving a  $K$ -dimensional vector of sums that we must then broadcast to the corresponding dimension of the numerator.

Recall that since the shape of `q_values` is `(K, n)`, we will take the sum over the second axis (`axis=1`) to get:

```
>>> q_sum = q_values.sum(axis=1)
```

Now, we have that the shape of `q_sum` is `(K,)`. We need to broadcast this to the corresponding dimension of `weighted_sum_numerator`.

Recall that the shape of `weighted_sum_numerator` is `(K, d, d)`.

Thus, we want to change the shape of `q_sum` to match this to make broadcasting work properly:

<code>weighted_sum_numerator</code>	:	(K, d, d)
<code>q_sum</code>	:	(K,) -> (K, 1, 1)

Thus, we need to add 2 singleton dimensions to the *right* of the shape of `q_sum`. This can be accomplished by the `reshape` command just fine.

Hence, we get the following sequence of Python commands for the final expression, with no loops:

```
>>> obs_centered = Z - np.expand_dims(new_means, 1)
>>> weighted_sum_numerator = np.einsum("ki, kip, kiq -> kpq", q_values, ←
    obs_centered, obs_centered)
>>> q_sum = q_values.sum(axis=1)
>>> new_covars = weighted_sum_numerator / q_sum.reshape(-1, 1, 1)
```



# 2

# Pandas 1: Introduction

**Lab Objective:** Though NumPy and SciPy are powerful tools for numerical computing, they lack some of the high-level functionality necessary for many data science applications. Python's pandas library, built on NumPy, is designed specifically for data management and analysis. In this lab we introduce pandas data structures and syntax, and we explore the capabilities of pandas for quickly analyzing and presenting data.

## Pandas Basics

Pandas is a python library used primarily to analyze data. It combines functionality of NumPy, MatPlotLib, and SQL to create an easy to understand library that allows for the manipulation of data in various ways. In this lab we focus on the use of Pandas to analyze and manipulate data in ways similar to NumPy and SQL.

### Pandas Data Structures

#### Series

The first pandas data structure is a **Series**. A **Series** is a one-dimensional array that can hold any datatype, similar to an **ndarray**. However, a **Series** has an explicitly defined **index** that gives a label to each entry. An **index** generally is used to label the data.

Typically a **Series** contains information about one feature of the data. For example, the data in a **Series** might show a class's grades on a test and the **Index** would indicate each student in the class. To initialize a **Series**, the first parameter is the data and the second is the index.

```
>>> import pandas as pd
>>> import numpy as np
# Initialize Series of student grades
>>> math = pd.Series(np.random.randint(0, 100, 4), ["Mark", "Barbara",
...      "Eleanor", "David"])
>>> english = pd.Series(np.random.randint(0, 100, 5), ["Mark", "Barbara",
...      "David", "Greg", "Lauren"])
```

## DataFrame

The second key pandas data structure is a `DataFrame`. A `DataFrame` is a collection of multiple `Series`. It can be thought of as a 2-dimensional array, where each row is a separate datapoint and each column is a feature of the data. The rows are labeled with an `index` (as in a `Series`) and the columns are labeled in the attribute `columns`.

There are many different ways to initialize a `DataFrame`. One way to initialize a `DataFrame` is by passing in a dictionary as the data of the `DataFrame`. The keys of the dictionary will become the labels in `columns` and the values are the `Series` associated with the label.

```
# Create a DataFrame of student grades
>>> grades = pd.DataFrame({"Math": math, "English": english})
>>> grades
      Math   English
Barbara    52.0     73.0
David      10.0     39.0
Eleanor    35.0      NaN
Greg        NaN     26.0
Lauren     NaN     99.0
Mark       81.0     68.0
```

Notice that `pd.DataFrame` automatically lines up data from both `Series` that have the same index. If the data only appears in one of the `Series`, the corresponding entry for the other `Series` is `NaN`.

We can also initialize a `DataFrame` with a NumPy array. With this method, the data is passed in as a 2-dimensional NumPy array, while the column labels and the index are passed in as parameters. The first column label goes with the first column of the array, the second with the second, and so forth. The index works similarly.

```
# Initialize DataFrame with NumPy array. This is identical to the grades ←
# DataFrame above.
>>> data = np.array([[52.0, 73.0], [10.0, 39.0], [35.0, np.nan],
...      [np.nan, 26.0], [np.nan, 99.0], [81.0, 68.0]])
>>> grades = pd.DataFrame(data, columns = ["Math", "English"], index =
...      ["Barbara", "David", "Eleanor", "Greg", "Lauren", "Mark"])

# View the columns
>>> grades.columns
Index(['Math', 'English'], dtype='object')

# View the Index
>>> grades.index
Index(['Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'], dtype='object')
```

A DataFrame can also be viewed as a NumPy array using the attribute `values`.

```
# View the DataFrame as a NumPy array
>>> grades.values
array([[ 52.,  73.],
       [ 10.,  39.],
       [ 35.,  nan],
       [ nan,  26.],
       [ nan,  99.],
       [ 81.,  68.]])
```

## Data I/O

The pandas library has functions that make importing and exporting data simple. The functions allow for a variety of file formats to be imported and exported, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
<code>to_csv()</code>	Write the index and entries to a CSV file
<code>read_csv()</code>	Read a csv and convert into a DataFrame
<code>to_json()</code>	Convert the object to a JSON string
<code>to_pickle()</code>	Serialize the object and store it in an external file
<code>to_sql()</code>	Write the object data to an open SQL database
<code>read_html()</code>	Read a table in an html page and convert to a DataFrame

Table 2.1: Methods for exporting data in a pandas `Series` or `DataFrame`.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. Some frequently-used keyword arguments include the following:

- **delimiter**: The character that separates data fields. It is often a comma or a whitespace character.
- **header**: The row number (0 indexed) in the CSV file that contains the column names.
- **index\_col**: The column (0 indexed) in the CSV file that is the index for the DataFrame.
- **skiprows**: If an integer  $n$ , skip the first  $n$  rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- **names**: If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list.

Another particularly useful function is `read_html()`, which is useful when scraping data. It takes in a url or html file and an optional argument `match`, a string or regex, and returns a list of the tables that match the `match` in a DataFrame.

## Data Manipulation

### Accessing Data

In general, the best way to access data in a `Series` or `DataFrame` is through the indexers `loc` and `iloc`. While array slicing can be used, it is more efficient to use these indexers. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than slicing because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc` or `iloc` explicitly bypasses these extra checks. The `loc` index selects rows and columns based on their labels, while `iloc` selects them based on their integer position. With these indexers, the first and second arguments refer to the rows and columns, respectively, just as array slicing.

```
# Use loc to select the Math scores of David and Greg
>>> grades.loc[[ "David", "Greg"], "Math"]
David    10.0
Greg      NaN
Name: Math, dtype: float64

# Use iloc to select the Math scores of David and Greg
>>> grades.iloc[[1,3], 0]
David    10.0
Greg      NaN
```

To access an entire column of a `DataFrame`, the most efficient method is to use only square brackets and the name of the column, without the indexer. This syntax can also be used to create a new column or reset the values of an entire column.

```
# Create a new History column with array of random values
>>> grades[ "History"] = np.random.randint(0, 100, 6)
>>> grades[ "History"]
Barbara     4
David      92
Eleanor    25
Greg       79
Lauren     82
Mark       27
Name: History, dtype: int64

# Reset the column such that everyone has a 100
>>> grades[ "History"] = 100.0
>>> grades
   Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0     NaN     100.0
Greg     NaN      26.0    100.0
Lauren   NaN      99.0    100.0
Mark     81.0     68.0    100.0
```

Datasets can often be very large and thus difficult to visualize. Pandas has various methods to make this easier. The methods `head` and `tail` will show the first or last  $n$  data points, respectively, where  $n$  defaults to 5. The method `sample` will draw  $n$  random entries of the dataset, where  $n$  defaults to 1.

```
# Use head to see the first n rows
>>> grades.head(n=2)
      Math   English   History
Barbara    52.0      73.0     100.0
David      10.0      39.0     100.0

# Use sample to sample a random entry
>>> grades.sample()
      Math   English   History
Lauren     NaN      99.0     100.0
```

It may also be useful to re-order the columns or rows or sort according to a given column.

```
# Re-order columns
>>> grades.reindex(columns=["English", "Math", "History"])
      English   Math   History
Barbara    73.0    52.0     100.0
David      39.0    10.0     100.0
Eleanor     NaN    35.0     100.0
Greg        26.0    NaN      100.0
Lauren     99.0    NaN      100.0
Mark        68.0    81.0     100.0

# Sort descending according to Math grades
>>> grades.sort_values("Math", ascending=False)
      Math   English   History
Mark      81.0     68.0     100.0
Barbara   52.0     73.0     100.0
Eleanor   35.0     NaN      100.0
David      10.0     39.0     100.0
Greg        NaN     26.0     100.0
Lauren     NaN     99.0     100.0
```

Other methods used for manipulating `DataFrame` and `Series` panda structures can be found in Table 2.2.

Method	Description
<code>append()</code>	Concatenate two or more <code>Series</code> .
<code>drop()</code>	Remove the entries with the specified label or labels
<code>drop_duplicates()</code>	Remove duplicate values
<code>dropna()</code>	Drop null entries
<code>fillna()</code>	Replace null entries with a specified value or strategy
<code>reindex()</code>	Replace the index
<code>sample()</code>	Draw a random entry
<code>shift()</code>	Shift the index
<code>unique()</code>	Return unique values

Table 2.2: Methods for managing or modifying data in a pandas `Series` or `DataFrame`.

**Problem 1.** The file `budget.csv` contains the budget of a college student over the course of 4 years. Write a function that performs the following operations in this order:

1. Read in `budget.csv` as a `DataFrame` with the index as column 0. Hint: Use `index_col=0` to set the first column as the index when reading in the csv.
2. Reindex the columns such that amount spent on groceries is the first column and all other columns maintain the same ordering.
3. Sort the `DataFrame` in descending order by how much money was spent on `Groceries`.
4. Reset all values in the "`Rent`" column to `800.0`.
5. Reset all values in the first 5 data points to `0.0`.

Return the values of the updated `DataFrame` as a NumPy array.

## Basic Data Manipulation

Because the primary pandas data structures are based off of `ndarray`, most NumPy functions work with pandas structures. For example, basic vector operations work as would be expected:

```
# Sum history and english grades of all students
>>> grades["English"] + grades["History"]
Barbara    173.0
David      139.0
Eleanor     NaN
Greg       126.0
Lauren     199.0
Mark       168.0
dtype: float64

# Double all Math grades
>>> grades["Math"]*2
Barbara    104.0
David      20.0
```

```
Eleanor      70.0
Greg        NaN
Lauren      NaN
Mark       162.0
Name: Math, dtype: float64
```

In addition to arithmetic, `Series` has a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 2.3.

Method	Returns
<code>abs()</code>	Object with absolute values taken (of numerical data)
<code>idxmax()</code>	The index label of the maximum value
<code>idxmin()</code>	The index label of the minimum value
<code>count()</code>	The number of non-null entries
<code>cumprod()</code>	The cumulative product over an axis
<code>cumsum()</code>	The cumulative sum over an axis
<code>max()</code>	The maximum of the entries
<code>mean()</code>	The average of the entries
<code>median()</code>	The median of the entries
<code>min()</code>	The minimum of the entries
<code>mode()</code>	The most common element(s)
<code>prod()</code>	The product of the elements
<code>sum()</code>	The sum of the elements
<code>var()</code>	The variance of the elements

Table 2.3: Numerical methods of the `Series` and `DataFrame` pandas classes.

## Basic Statistical Functions

The pandas library allows us to easily calculate basic summary statistics of our data, which can be useful when we want a quick description of the data. The `describe()` function outputs several such summary statistics for each column in a `DataFrame`:

```
# Use describe to better understand the data
>>> grades.describe()
    Math   English   History
count    4.000000    5.000000     6.0
mean    44.500000   61.000000   100.0
std     29.827281   28.92231     0.0
min     10.000000   26.000000   100.0
25%    28.750000   39.000000   100.0
50%    43.500000   68.000000   100.0
75%    59.250000   73.000000   100.0
max    81.000000   99.000000   100.0
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available.

```
# Find the average grade for each student
```

```
>>> grades.mean(axis=1)
Barbara    75.000000
David      49.666667
Eleanor   67.500000
Greg       63.000000
Lauren    99.500000
Mark      83.000000
dtype: float64

# Give correlation matrix between subjects
>>> grades.corr()
          Math  English  History
Math      1.00000  0.84996     NaN
English   0.84996  1.00000     NaN
History    NaN      NaN      NaN
```

The method `rank()` can be used to rank the values in a data set, either within each entry or with each column. This function defaults ranking in ascending order: the least will be ranked 1 and the greatest will be ranked the highest number.

```
# Rank each student's performance in their classes in descending order
# (best to worst)
# The method keyword specifies what rank to use when ties occur.
>>> grades.rank(axis=1, method="max", ascending=False)
          Math  English  History
Barbara   3.0      2.0      1.0
David     3.0      2.0      1.0
Eleanor   2.0      NaN      1.0
Greg      NaN      2.0      1.0
Lauren    NaN      2.0      1.0
Mark      2.0      3.0      1.0
```

These methods can be very effective in interpreting data. For example, the `rank()` example above shows use that Barbara does best in History, then English, and then Math.

## Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing or anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. If we are not interested in the missing values, we can simply drop them from the data altogether, or we can fill them with some other value, such as the mean. `NaN` might also mean something specific, such as some default value, which should inform what to do with `NaN` values.

```
# Grades with all NaN values dropped
>>> grades.dropna()
          Math  English  History
Barbara   52.0     73.0    100.0
David     10.0     39.0    100.0
```

```

Mark      81.0    68.0    100.0

# fill missing data with 50.0
>>> grades.fillna(50.0)
      Math   English   History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0     50.0    100.0
Greg     50.0     26.0    100.0
Lauren   50.0     99.0    100.0
Mark     81.0    68.0    100.0

```

When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using. For example, `sum()` and `mean()` ignore NaN values in the computation.

### ACHTUNG!

Always consider missing data carefully when analyzing a dataset. It may not always be helpful to drop the data or fill it in with a random number. Consider filling the data with the mean of surrounding data or the mean of the feature in question. Overall, the choice for how to fill missing data should make sense with the dataset.

**Problem 2.** Write a function which uses `budget.csv` to answer the questions "Which category affects living expenses the most? Which affects other expenses the most?" Perform the following manipulations:

1. Fill all NaN values with 0.0.
2. Create two new columns, "`Living Expenses`" and "`Other`". Set the value of "`Living Expenses`" to be the sum of the columns "`Rent`", "`Groceries`", "`Gas`" and "`Utilities`". Set the value of "`Other`" to be the sum of the columns "`Dining Out`", "`Out With Friends`" and "`Netflix`".
3. Identify which column, other than "`Living Expenses`", correlates most with "`Living Expenses`" and which column, other than "`Other`", correlates most with "`Other`". This can indicate which columns in the budget affect the overarching categories the most.

Return the names of each of those columns as a tuple. The first should be of the column corresponding to "`Living Expenses`" and the second to "`Other`".

## Complex Operations in Pandas

Often times, the data that we have is not exactly the data we want to analyze. In cases like this we use more complex data manipulation tools to access only the data that we need.

For the examples below, we will use the following data:

```
>>> name = ["Mylan", "Regan", "Justin", "Jess", "Jason", "Remi", "Matt",
...     "Alexander", "JeanMarie"]
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ["Sp", "Se", "Fr", "Se", "Sp", 'J', 'J', 'J', "Se"]
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({"ID": ID, "Name": name, "Sex": sex, "Age": age,
...     "Class": rank})
>>> otherInfo = pd.DataFrame({"ID": ID, "GPA": GPA, "Financial_Aid": aid})
>>> mathInfo = pd.DataFrame({"ID": mathID, "Grade": mathGd, "Math_Major":
...     major})
```

Before querying our data, it is helpful to know some of its basic properties, such as number of columns, number of rows, and the datatypes of the columns. This can be done by simply calling the `info()` method on the desired DataFrame:

```
>>> mathInfo.info()
<class "pandas.core.frame.DataFrame">
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype  
---  --          --          --      
 0   ID          5 non-null    int64  
 1   Grade       5 non-null    float64 
 2   Math_Major  5 non-null    object  
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

## Masks

Sometimes, we only want to access data from a single column. For example if we want to only access the ID of the students in the `studentInfo DataFrame`, then we would use the following syntax.

```
# Get the ID column from studentInfo
>>> studentInfo.ID
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
```

```
8    8
Name: ID, dtype: int64
```

If we want to access multiple columns at once we can use a list of column names.

```
# Get the ID and Age columns.
>>> studentInfo[["ID", "Age"]]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   20
```

Now we can access the specific columns that we want. However, some of these columns may still contain data points that we don't want to consider. In this case we can build a mask. Each mask that we build will return a pandas `Series` object with a `bool` value at each index indicating if the condition is satisfied.

```
# Create a mask for all student receiving financial aid.
>>> mask = otherInfo["Financial_Aid"] == 'y'
# Access other info where the mask is true and display the ID and GPA columns.
>>> otherInfo[mask][["ID", "GPA"]]
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4
```

We can also create compound masks with multiple statements. We do this using the same syntax you would use for a compound mask in a normal NumPy array. Useful operators are `&`, the AND operator; `|`, the OR operator; and `~`, the NOT operator.

```
# Get all student names where Class = 'J' OR Class = "Sp".
>>> mask = (studentInfo.Class == 'J') | (studentInfo.Class == "Sp")
>>> studentInfo[mask].Name
0      Mylan
4     Jason
5      Remi
6      Matt
7  Alexander
Name: Name, dtype: object
# This can also be accomplished with the following command:
# studentInfo[studentInfo["Class"].isin(['J', "Sp"])]["Name"]
```

**Problem 3.** Read in the file `crime_data.csv` as a pandas object. The file contains data on types of crimes in the U.S. from 1960 to 2016. Set the index as the column "`Year`". Answer the following questions using the pandas methods learned in this lab. The answer of each question should be saved as indicated. Return the answers to all three questions as a tuple (i.e. `(answer_1, answer_2, answer_3)`).

1. Identify the three crimes that have a mean yearly number of occurrences over 1,500,000. Of these three crimes, which two are very correlated? Which of these two crimes has a greater maximum value? Save the title of this column as a variable to return as the answer. Hint: Consider using `idxmax()` to extract the title.
2. Examine the data from 2000 and later. Sort this data (in ascending order) according to number of murders. Find the years where aggravated assault is greater than 850,000. Save the indices (the years) of the masked and reordered `DataFrame` as a NumPy array to return as the answer.
3. What year had the highest crime rate? In this year, which crime was committed the most? What percentage of the total crime that year was it? Return this percentage as the answer in decimal form (e.g. 50% would be returned as 0.5). Note that the crime rate is `#crimes/population`.

## Working with Dates and Times

The `datetime` module in the standard library provides a few tools for representing and operating on dates and times. The `datetime.datetime` object represents a *time stamp*: a specific time of day on a certain day. Its constructor accepts a four-digit year, a month (starting at 1 for January), a day, and, optionally, an hour, minute, second, and microsecond. Each of these arguments must be an integer, with the hour ranging from 0 to 23.

```
>>> from datetime import datetime

# Represent November 18th, 1991, at 2:01 PM.
>>> bday = datetime(1991, 11, 18, 14, 1)
>>> print(bday)
1991-11-18 14:01:00

# Find the number of days between 11/18/1991 and 11/9/2017.
>>> dt = datetime(2017, 11, 9) - bday
>>> dt.days
9487
```

The `datetime.datetime` object has a parser method, `strptime()`, that converts a string into a new `datetime.datetime` object. The parser is flexible so the user must specify the format that the dates are in. For example, if the dates are in the format "`Month/Day//Year::Hour`", specify `format="=%m/%d//%Y::%H"` to parse the string appropriately. See Table 2.4 for formatting options.

Pattern	Description
%Y	4-digit year
%y	2-digit year
%m	1- or 2-digit month
%d	1- or 2-digit day
%H	Hour (24-hour)
%I	Hour (12-hour)
%M	2-digit minute
%S	2-digit second

Table 2.4: Formats recognized by `datetime.strptime()`

```
>>> print(datetime.strptime("1991-11-18 / 14:01", "%Y-%m-%d / %H:%M"),
...       datetime.strptime("1/22/1996", "%m/%d/%Y"),
...       datetime.strptime("19-8, 1998", "%d-%m, %Y"), sep='\n')
1991-11-18 14:01:00          # The date formats are now standardized.
1996-01-22 00:00:00          # If no hour/minute/seconds data is given,
1998-08-19 00:00:00          # the default is midnight.
```

## Converting Dates to an Index

The `TimeStamp` class is the pandas equivalent to a `datetime.datetime` object. A pandas index composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with a `DatetimeIndex` is called a *time series*. The function `pd.to_datetime()` converts a collection of dates in a parsable format to a `DatetimeIndex`. The format of the dates is inferred if possible, but it can be specified explicitly with the same syntax as `datetime.strptime()`.

```
>>> import pandas as pd

# Convert some dates (as strings) into a DatetimeIndex.
>>> dates = ["2010-1-1", "2010-2-1", "2012-1-1", "2012-1-2"]
>>> pd.to_datetime(dates)
DatetimeIndex(['2010-01-01', '2010-02-01', '2012-01-01', '2012-01-02'],
               dtype='datetime64[ns]', freq=None)

# Create a time series, specifying the format for the DatetimeIndex.
>>> dates = ["1/1, 2010", "1/2, 2010", "1/1, 2012", "1/2, 2012"]
>>> date_index = pd.to_datetime(dates, format="%m/%d, %Y")
>>> pd.Series([x**2 for x in range(4)], index=date_index)
2010-01-01    0
2010-01-02    1
2012-01-01    4
2012-01-02    9
dtype: int64
```

**Problem 4.** The file `DJIA.csv` contains daily closing values of the Dow Jones Industrial Average from 2006–2016. Read the data into a `Series` or `DataFrame` with a `DatetimeIndex` as the index. Drop any rows without numerical values, cast the "`VALUE`" column to floats, then return the updated `DataFrame`.

Hint: You can change the column type the same way you'd change a numpy array type.

## Generating Time-based Indices

Some time series datasets come without explicit labels but have instructions for deriving timestamps. For example, a list of bank account balances might have records from the beginning of every month, or heart rate readings could be recorded by an app every 10 minutes. Use `pd.date_range()` to generate a `DatetimeIndex` where the timestamps are equally spaced. The function is analogous to `np.arange()` and has the following parameters:

Parameter	Description
<code>start</code>	Starting date
<code>end</code>	End date
<code>periods</code>	Number of dates to include
<code>freq</code>	Amount of time between consecutive dates
<code>normalize</code>	Normalizes the start and end times to midnight

Table 2.5: Parameters for `pd.date_range()`.

Exactly three of the parameters `start`, `end`, `periods`, and `freq` must be specified to generate a range of dates. The `freq` parameter accepts a variety of string representations, referred to as *offset aliases*. See Table 2.6 for a sampling of some of the options. For a complete list of the options, see [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html#timeseries-offset-aliases](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases).

Parameter	Description
<code>'D'</code>	calendar daily (default)
<code>'B'</code>	business daily (every business day)
<code>'H'</code>	hourly
<code>'T'</code>	minutely
<code>'S'</code>	secondly
<code>"MS"</code>	first day of the month (Month Start)
<code>"BMS"</code>	first business day of the month (Business Month Start)
<code>"W-MON"</code>	every Monday (Week-Monday)
<code>"WOM-3FRI"</code>	every 3rd Friday of the month (Week of the Month - 3rd Friday)

Table 2.6: Options for the `freq` parameter to `pd.date_range()`.

```
# Create a DatetimeIndex for 5 consecutive days starting on September 28, 2016.
>>> pd.date_range(start="9/28/2016 16:00", periods=5)
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-29 16:00:00',
```

```

'2016-09-30 16:00:00', '2016-10-01 16:00:00',
'2016-10-02 16:00:00'],
dtype='datetime64[ns]', freq='D')

# Create a DatetimeIndex with the first weekday of every other month in 2016.
>>> pd.date_range(start="1/1/2016", end="1/1/2017", freq="2BMS")
DatetimeIndex(['2016-01-01', '2016-03-01', '2016-05-02', '2016-07-01',
               '2016-09-01', '2016-11-01'],
              dtype='datetime64[ns]', freq='2BMS')

# Create a DatetimeIndex for 10 minute intervals between 4:00 PM and 4:30 PM on ←
# September 9, 2016.
>>> pd.date_range(start="9/28/2016 16:00",
                  end="9/28/2016 16:30", freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
               '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')

# Create a DatetimeIndex for 2 hour 30 minute intervals between 4:30 PM and ←
# 2:30 AM on September 29, 2016.
>>> pd.date_range(start="9/28/2016 16:30", periods=5, freq="2h30min")
DatetimeIndex(['2016-09-28 16:30:00', '2016-09-28 19:00:00',
               '2016-09-28 21:30:00', '2016-09-29 00:00:00',
               '2016-09-29 02:30:00'],
              dtype='datetime64[ns]', freq='150T')

```

**Problem 5.** The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. Paychecks are given every other Friday, starting on March 14, 2008, and the employee started working on March 13, 2008.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Set this as the new index of the `DataFrame` and return the `DataFrame`.

## Elementary Time Series Analysis

### Shifting

`DataFrame` and `Series` objects have a `shift()` method that allows you to move data up or down relative to the index. When dealing with time series data, we can also shift the `DatetimeIndex` relative to a time offset.

```

>>> df = pd.DataFrame(dict(VALUE=np.random.rand(5)),
                      index=pd.date_range("2016-10-7", periods=5, freq='D'))
>>> df
          VALUE
2016-10-07  0.127895

```

```

2016-10-08  0.811226
2016-10-09  0.656711
2016-10-10  0.351431
2016-10-11  0.608767

>>> df.shift(1)
      VALUE
2016-10-07      NaN
2016-10-08  0.127895
2016-10-09  0.811226
2016-10-10  0.656711
2016-10-11  0.351431

>>> df.shift(-2)
      VALUE
2016-10-07  0.656711
2016-10-08  0.351431
2016-10-09  0.608767
2016-10-10      NaN
2016-10-11      NaN

>>> df.shift(14, freq='D')
      VALUE
2016-10-21  0.127895
2016-10-22  0.811226
2016-10-23  0.656711
2016-10-24  0.351431
2016-10-25  0.608767

```

Shifting data makes it easy to gather statistics about changes from one timestamp or period to the next.

```

# Find the changes from one period/timestamp to the next
>>> df - df.shift(1)           # Equivalent to df.diff().
      VALUE
2016-10-07      NaN
2016-10-08  0.683331
2016-10-09 -0.154516
2016-10-10 -0.305279
2016-10-11  0.257336

```

**Problem 6.** Compute the following information about the DJIA dataset from Problem 4 that has a DateTimeIndex.

- The single day with the largest gain.
- The single day with the largest loss.

Return the DateTimeIndex of the day with the largest gain and the day with the largest loss as a tuple.

(Hint: Call your function from Problem 4 to get the DataFrame already cleaned and with DatetimeIndex).

More information on how to use `datetime` with Pandas is in the additional material section. This includes working with `Periods` and more analysis with time series.

## Additional Material

### SQL Operations in pandas

DataFrames are tabular data structures bearing an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases; however, pandas can accomplish many of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, eliminating the need to switch between programming languages for different tasks. Within pandas, we can handle both the querying *and* data analysis.

For the examples below, we will use the following data:

```
>>> name = ["Mylan", "Regan", "Justin", "Jess", "Jason", "Remi", "Matt",
...           "Alexander", "JeanMarie"]
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ["Sp", "Se", "Fr", "Se", "Sp", 'J', 'J', 'J', "Se"]
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({"ID": ID, "Name": name, "Sex": sex, "Age": age,
...           "Class": rank})
>>> otherInfo = pd.DataFrame({"ID": ID, "GPA": GPA, "Financial_Aid": aid})
>>> mathInfo = pd.DataFrame({"ID": mathID, "Grade": mathGd, "Math_Major": 
...           major})
```

SQL SELECT statements can be done by column indexing. WHERE statements can be included by adding masks (just like in a NumPy array). The method `isin()` can also provide a useful WHERE statement. This method accepts a list, dictionary, or Series containing possible values of the DataFrame or Series. When called upon, it returns a Series of booleans, indicating whether an entry contained a value in the parameter passed into `isin()`.

```
# SELECT ID, Age FROM studentInfo
>>> studentInfo[["ID", "Age"]]
   ID  Age
0    0   20
1    1   21
2    2   18
3    3   22
4    4   19
5    5   20
6    6   20
7    7   19
8    8   20

# SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> mask = otherInfo["Financial_Aid"] == 'y'
>>> otherInfo[mask][["ID", "GPA"]]
```

```

      ID  GPA
0    0  3.8
3    3  3.9
7    7  3.4

# SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = "Sp"
>>> studentInfo[studentInfo["Class"].isin(['J', "Sp"])]["Name"]
0        Mylan
4        Jason
5        Remi
6        Matt
7   Alexander
Name: Name, dtype: object

```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function. `merge` takes the two `DataFrame` objects to join as parameters, as well as keyword arguments specifying the column on which to join, along with the type (left, right, inner, outer).

```

# SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = mathInfo.ID
>>> pd.merge(studentInfo, mathInfo, on="ID")
      ID  Name  Sex  Age  Class  Grade  Math_Major
0    0  Mylan    M   20    Sp     4.0        y
1    1  Regan    F   21    Se     3.0        n
2    3   Jess    F   22    Se     4.0        n
3    5   Remi    F   20      J     3.5        y
4    6    Matt    M   20      J     3.0        n

# SELECT GPA, Grade FROM otherInfo FULL OUTER JOIN mathInfo ON otherInfo.
# ID = mathInfo.ID
>>> pd.merge(otherInfo, mathInfo, on="ID", how="outer") [["GPA", "Grade"]]
      GPA  Grade
0  3.8    4.0
1  3.5    3.0
2  3.0    NaN
3  3.9    4.0
4  2.8    NaN
5  2.9    3.5
6  3.8    3.0
7  3.4    NaN
8  3.7    NaN

```

## More Datetime with Pandas

### Periods

A pandas `Timestamp` object represents a precise moment in time on a given day. Some data, however, is recorded over a time interval, and it wouldn't make sense to place an exact timestamp on any of the measurements. For example, a record of the number of steps walked in a day, box office earnings per week, quarterly earnings, and so on. This kind of data is better represented with the pandas `Period` object and the corresponding `PeriodIndex`.

The `Period` class accepts a `value` and a `freq`. The `value` parameter indicates the label for a given `Period`. This label is tied to the `end` of the defined `Period`. The `freq` indicates the length of the `Period` and in some cases can also indicate the offset of the `Period`. The default value for `freq` is 'M' for months. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 2.6.

```
# Creates a period for month of Oct, 2016.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time                      # The start and end times of the period
Timestamp('2016-10-01 00:00:00')      # are recorded as Timestamps.
>>> p1.end_time
Timestamp('2016-10-31 23:59:59.999999999')

# Represent the annual period ending in December that includes 10/03/2016.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2016-01-01 00:00:00')
> p2.end_time
Timestamp('2016-12-31 23:59:59.999999999')

# Get the weekly period ending on a Saturday that includes 10/03/2016.
>>> print(pd.Period("2016-10-03", freq="W-SAT"))
2016-10-02/2016-10-08
```

Like the `pd.date_range()` method, the `pd.period_range()` method is useful for generating a `PeriodIndex` for unindexed data. The syntax is essentially identical to that of `pd.date_range()`. When using `pd.period_range()`, remember that the `freq` parameter marks the end of the period. After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# Represent quarters from 2008 to 2010, with Q4 ending in December.
>>> pd.period_range(start="2008", end="2010-12", freq="Q-DEC")
PeriodIndex(['2008Q1', '2008Q2', '2008Q3', '2008Q4', '2009Q1', '2009Q2',
             '2009Q3', '2009Q4', '2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='period[Q-DEC]')

# Get every three months from March 2010 to the start of 2011.
>>> p = pd.period_range("2010-03", "2011", freq="3M")
>>> p
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'], dtype='period[3M]')

# Change frequency to be quarterly.
```

```
>>> q = p.asfreq("Q-DEC")
>>> q
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'], dtype='period[Q-DEC]')
```

The bounds of a `PeriodIndex` object can be shifted by adding or subtracting an integer. `PeriodIndex` will be shifted by  $n \times \text{freq}$ .

```
# Shift index by 1
>>> q -= 1
>>> q
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'], dtype='period[Q-DEC]')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so. The `how` parameter can be `start` or `end` and determines if the timestamp is the beginning or the end of the period. Similarly, you can switch from timestamps to periods.

```
# Convert to timestamp (last day of each quarter)
>>> q = q.to_timestamp(how="end")
>>> q
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
               dtype='datetime64[ns]', freq='Q-DEC')

>>> q.to_period("Q-DEC")
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

## Operations on Time Series

There are certain operations only available to Series and DataFrames that have a `DatetimeIndex`. A sampling of this functionality is described throughout the remainder of this lab.

### Slicing

Slicing is much more flexible in pandas for time series. We can slice by year, by month, or even use traditional slicing syntax to select a range of dates.

```
# Select all rows in a given year
>>> df["2010"]
          0         1
2010-01-01  0.566694  1.093125
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036

# Select all rows in a given month of a given year
>>> df["2012-01"]
          0         1
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
```

```

2012-01-03  1.436843  0.596143

# Select a range of dates using traditional slicing syntax
>>> df["2010-1-2": "2011-12-31"]
          0           1
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036
2011-01-01  0.300766  0.934895

```

## Resampling

Some datasets do not have datapoints at a fixed frequency. For example, a dataset of website traffic has datapoints that occur at irregular intervals. In situations like these, *resampling* can help provide insight on the data.

The two main forms of resampling are *downsampling*, aggregating data into fewer intervals, and *upsampling*, adding more intervals.

To downsample, use the `resample()` method of the `Series` or `DataFrame`. This method is similar to `groupby()` in that it groups different entries together. Then aggregation produces a new data set. The first parameter to `resample()` is an offset string from Table 2.6: '`D`' for daily, '`H`' for hourly, and so on.

```

>>> import numpy as np

# Get random data for every day from 2000 to 2010.
>>> dates = pd.date_range(start="2000-1-1", end="2009-12-31", freq='D')
>>> df = pd.Series(np.random(len(dates)), index=dates)
>>> df
2000-01-01    0.559
2000-01-02    0.874
2000-01-03    0.774
...
2009-12-29    0.837
2009-12-30    0.472
2009-12-31    0.211
Freq: D, Length: 3653, dtype: float64

# Group the data by year.
>>> years = df.resample('A')           # 'A' for "annual".
>>> years.agg(len)                 # Number of entries per year.
2000-12-31    366.0
2001-12-31    365.0
2002-12-31    365.0
...
2007-12-31    365.0
2008-12-31    366.0
2009-12-31    365.0
Freq: A-DEC, dtype: float64

```

```
>>> years.mean()                      # Average entry by year.
2000-12-31    0.491
2001-12-31    0.514
2002-12-31    0.484
...
2007-12-31    0.508
2008-12-31    0.521
2009-12-31    0.523
Freq: A-DEC, dtype: float64

# Group the data by month.
>>> months = df.resample('M')
>>> len(months.mean())                 # 12 months x 10 years = 120 months.
120
```

## Elementary Time Series Analysis

### Rolling Functions and Exponentially-Weighted Moving Functions

Many time series are inherently noisy. To analyze general trends in data, we use *rolling functions* and *exponentially-weighted moving (EWM)* functions. Rolling functions, or *moving window functions*, perform a calculation on a window of data. There are a few rolling functions that come standard with pandas.

#### Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the *rolling average*, which takes the average value over a window of data.

```
import matplotlib.pyplot as plt

# Generate a time series using random walk from a uniform distribution.
seed = 42
N = 10000
bias = 0.01

rng = np.random.default_rng(seed)
s = np.zeros(N)
s[1:] = rng.uniform(low=-1, high=1, size=N-1) + bias
s = pd.Series(s.cumsum(),
              index=pd.date_range("2015-10-20", freq='H', periods=N))

# Plot the original data together with a rolling average.
ax1 = plt.subplot(121)
s.plot(color="gray", lw=0.3, ax=ax1)
s.rolling(window=200).mean().plot(color='r', lw=1, ax=ax1)
ax1.legend(["Actual", "Rolling"], loc="lower right")
ax1.set_title("Rolling Average")
```

The function call `s.rolling(window=200)` creates a `pd.core.rolling.Window` object that can be aggregated with a function like `mean()`, `std()`, `var()`, `min()`, `max()`, and so on.

### Exponentially-Weighted Moving (EWM) Functions

Whereas a moving window function gives equal weight to the whole window, an *exponentially-weighted moving* function gives more weight to the most recent data points.

In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

$$z_i = \alpha \bar{x}_i + (1 - \alpha) z_{i-1},$$

where  $z_i$  is the value of the EWMA at time  $i$ ,  $\bar{x}_i$  is the average for the  $i$ -th window, and  $\alpha$  is the decay factor that controls the importance of previous data points. Notice that  $\alpha = 1$  reduces to the rolling average.

More commonly, the decay is expressed as a function of the window size. In fact, the `span` for an EWMA is nearly analogous to `window size` for a rolling average.

Notice the syntax for EWM functions is very similar to that of rolling functions.

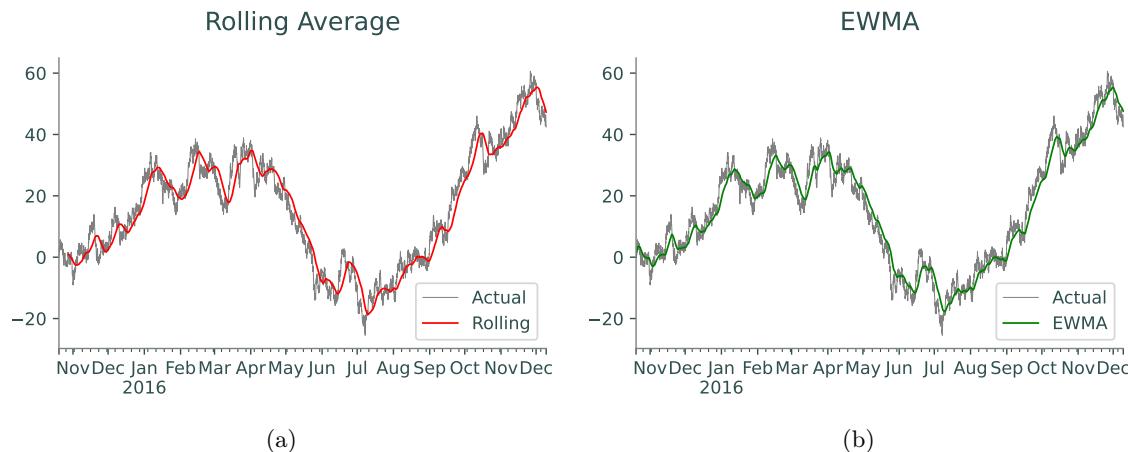


Figure 2.1: Rolling average and EWMA.

```
ax2 = plt.subplot(122)
s.plot(color="gray", lw=0.3, ax=ax2)
s.ewm(span=200).mean().plot(color='g', lw=1, ax=ax2)
ax2.legend(["Actual", "EWMA"], loc="lower right")
ax2.set_title("EWMA")
```

# 3

## Pandas 2: Plotting

**Lab Objective:** *Clear, insightful visualizations are a crucial part of data analysis. To facilitate quick data visualization, pandas includes several tools that wrap around matplotlib. These tools make it easy to compare different parts of a data set, explore the data as a whole, and spot patterns and correlations in the data.*

### NOTE

This lab is designed to be more flexible than other labs. You will be asked to make visualizations customized to your liking. As long as your plots clearly visualize the data, are properly labeled, etc., you will be graded generously.

## Overview of Plotting Tools

The main tool for visualization in pandas is the `plot()` method for `Series` and `DataFrames`. The method has a keyword argument `kind` that specifies the type of plot to draw. The valid options for `kind` are detailed below.

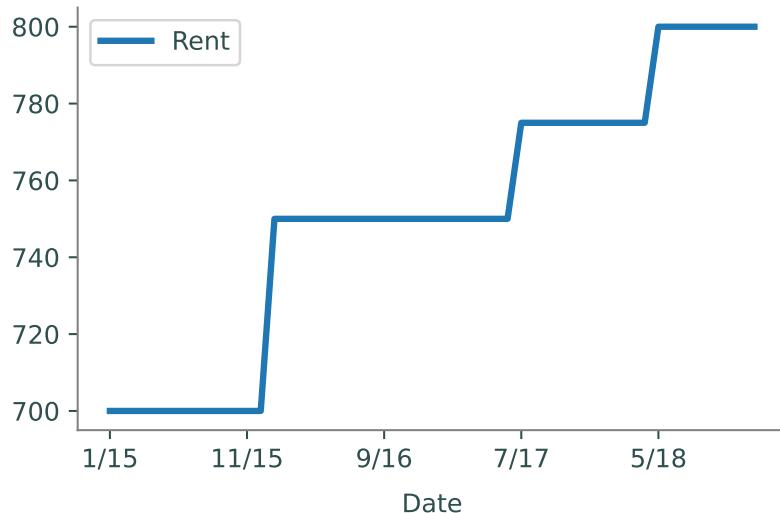
Plot Type	<code>plot()</code> ID	Uses and Advantages
Line plot	<code>"line"</code>	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	<code>"scatter"</code>	Compare exactly two data sets, independent of ordering
Bar plot	<code>"bar", "barch"</code>	Compare categorical or sequential data
Histogram	<code>"hist"</code>	Show frequencies of one set of values, independent of ordering
Box plot	<code>"box"</code>	Display min, median, max, and quartiles; compare data distributions
Hexbin plot	<code>"hexbin"</code>	2D histogram; reveal density of cluttered scatter plots

Table 3.1: Types of plots in pandas. The plot ID is the value of the keyword argument `kind`. That is, `df.plot(kind="scatter")` creates a scatter plot. The default `kind` is `"line"`.

The `plot()` method calls `plt.plot()`, `plt.hist()`, `plt.scatter()`, and other matplotlib plotting functions, but it also assigns axis labels, tick marks, legends, and a few other things based on the index and the data. Most calls to `plot()` specify the kind of plot and which `Series` to use as the `x` and `y` axes. By default, the `index` of the `Series` or `DataFrame` is used for the `x` axis.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

>>> budget = pd.read_csv("budget.csv", index_col="Date")
>>> budget.plot(y="Rent") # Plot rent against the index (date).
```



In this case, the call to the `plot()` method is essentially equivalent to the following code.

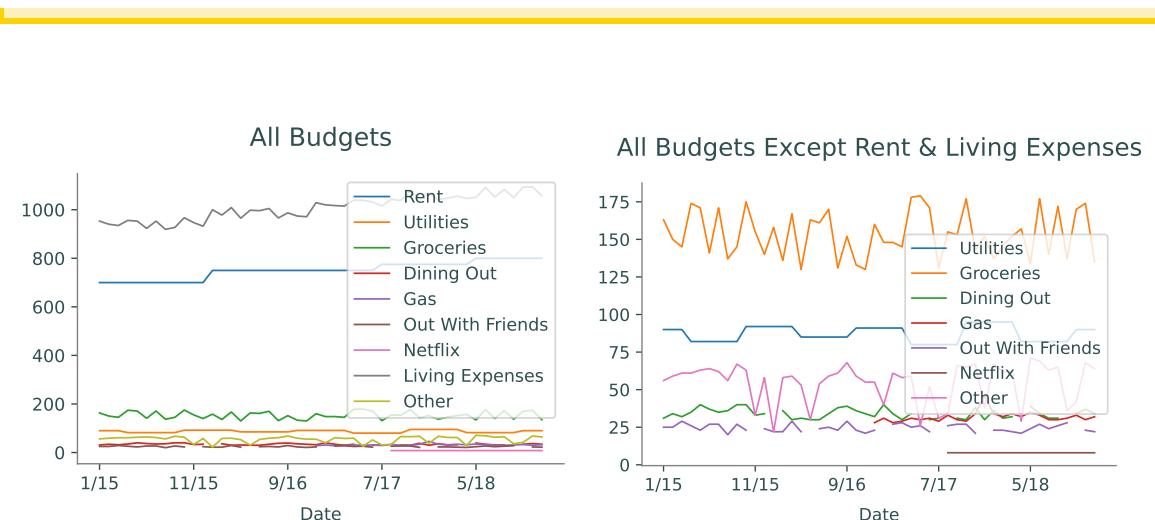
```
>>> plt.plot(budget.index, budget["Rent"], label="Rent")
>>> plt.xlabel(budget.index.name)
>>> plt.xlim(min(budget.index), max(budget.index))
>>> plt.legend(loc="best")
```

The `plot()` method also takes in many keyword arguments for matplotlib plotting and annotation functions. For example, setting `legend=False` disables the legend, providing a value for `title` sets the figure title, `grid=True` turns a grid on, and so on. For more customizations, see <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html>.

## Visualizing an Entire Data Set

A good way to start analyzing an unfamiliar data set is to visualize as much of the data as possible to determine which parts are most important or interesting. For example, since the columns in a `DataFrame` share the same index, the columns can all be graphed together using the index as the *x*-axis. By default, the `plot()` method attempts to plot **every Series** (column) in a `DataFrame`. This is especially useful with sequential data, like the budget data set.

```
# Plot all columns together against the index.
>>> budget.plot(title="All Budgets", linewidth=1)
>>> budget.drop(["Rent", "Living Expenses"], axis=1).plot(linewidth=1, title="↔
    All Budgets Except Rent & Living Expenses")
```



(a) All columns of the budget data set on the same figure, using the index as the *x*-axis.

(b) All columns of the budget data set except "**Living Expenses**" and "**Rent**".

Figure 3.1

While plotting every `Series` at once can give an overview of all the data, the resulting plot is often difficult for the reader to understand. For example, the budget data set has 9 columns, so the resulting figure, Figure 3.1a, is fairly cluttered.

One way to declutter a visualization is to examine less data. For example, the columns "**Living Expenses**" and "**Rent**" have values that are much larger than the other columns. Dropping these columns gives a better overview of the remaining data, as shown in Figure 3.1b.

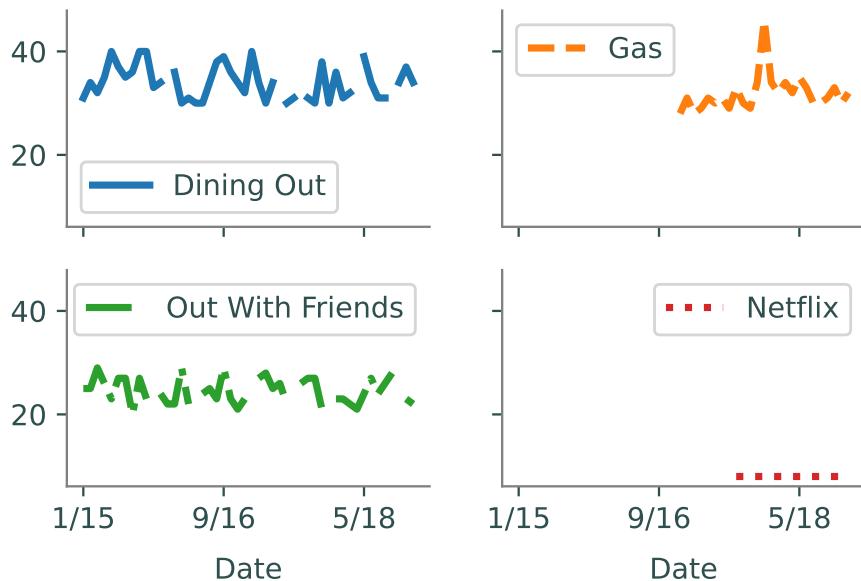
### ACHTUNG!

Often plotting all data at once is unwise because columns have **different units of measure**. Be careful not to plot parts of a data set together if those parts do not have the same units or are otherwise incomparable.

Another way to declutter a plot is to use subplots. To quickly plot several columns in separate subplots, use `subplots=True` and specify a shape tuple as the `layout` for the plots. Subplots automatically share the same *x*-axis. Set `sharey=True` to force them to share the same *y*-axis as well.

```
>>> budget.plot(y=["Dining Out", "Gas", "Out With Friends", "Netflix"],
...     subplots=True, layout=(2, 2), sharey=True,
...     style=['-', '--', '-.', ':'], title="Plots of Dollars Spent for ↵
... Different Budgets")
```

## Plots of Dollars Spent for Different Budgets



As mentioned previously, the `plot()` method can be used to plot different kinds of plots. One possible kind of plot is a histogram. Since plots made by the `plot()` method share an *x*-axis by default, histograms turn out poorly whenever there are columns with very different data ranges or when more than one column is plotted at once.

```
# Plot three histograms together.
>>> budget.plot(kind="hist", y=["Gas", "Dining Out", "Out With Friends"],
...     alpha=0.7, bins=10, title="Frequency of Amount (in dollars) Spent")

# Plot three histograms, stacking one on top of the other.
>>> budget.plot(kind="hist", y=["Gas", "Dining Out", "Out With Friends"],
...     bins=10, stacked=True, title="Frequency of Amount (in dollars) Spent")
```

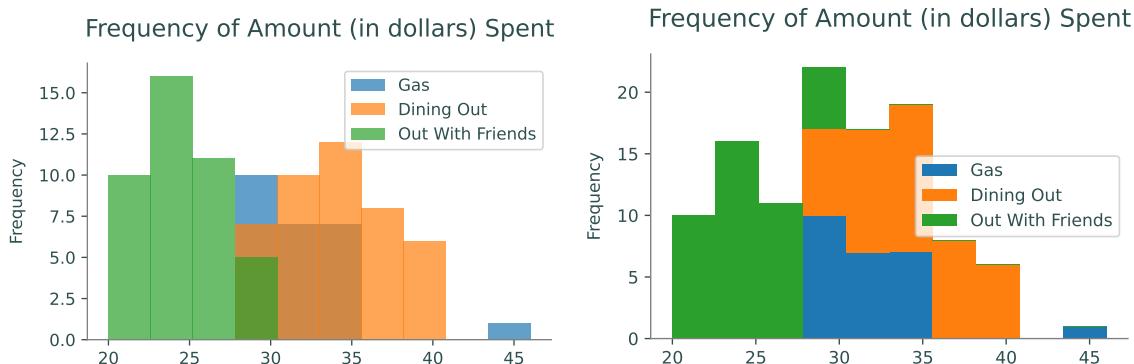


Figure 3.2: Two examples of histograms that are difficult to understand because multiple columns are plotted.

Thus, histograms are good for examining the distribution of a **single** column in a data set. For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter. Choose a number of bins that accurately represents the data; the wrong number of bins can create a misleading or uninformative visualization.

```
>>> budget[["Dining Out", "Gas"]].hist(grid=False, bins=10)
```

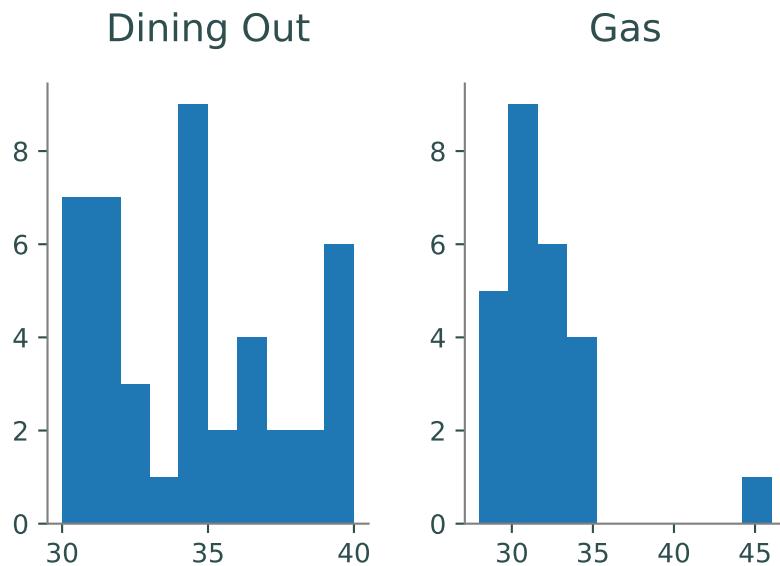


Figure 3.3: Histograms of "Dining Out" and "Gas".

**Problem 1.** Create 3 visualizations for the data in `crime_data.csv`. Make one of the visualizations a histogram. The visualizations should be well labeled and easy to understand.

## Patterns and Correlations

After visualizing the entire data set initially, a good next step is to closely compare related parts of the data. This can be done with different types of visualizations. For example, Figure 3.1b suggests that the "Dining Out" and "Out With Friends" columns are roughly on the same scale. Since this data is sequential (indexed by time), start by plotting these two columns against the index. Next, create a scatter plot of one of the columns versus the other to investigate correlations that are independent of the index. Unlike other types of plots, using `kind="scatter"` requires both `x` and `y` columns as arguments.

```
# Plot "Dining Out" and "Out With Friends" as lines against the index.
>>> budget.plot(y=["Dining Out", "Out With Friends"], title="Amount Spent on ←
    Dining Out and Out with Friends per Day")

# Make a scatter plot of "Dining Out" against "Out With Friends"
>>> budget.plot(kind="scatter", x="Dining Out", y="Out With Friends",
...     alpha=0.8, xlim=(0, max(budget["Dining Out"]) + 1),
...     ylim=(0, max(budget["Out With Friends"]) + 1),
...     title="Correlation between Dining Out and Out with Friends")
```

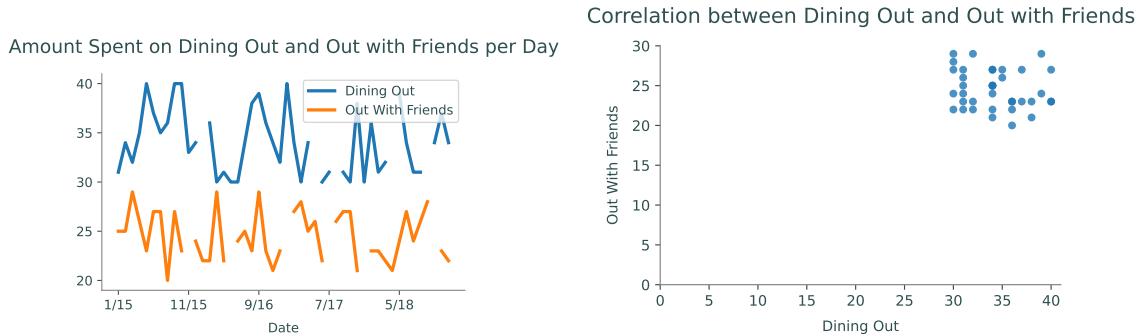


Figure 3.4: Correlations between "Dining Out" and "Out With Friends".

The first plot shows us that more money is spent on dining out than being out with friends overall. However, both categories stay in the same range for most of the data. This is confirmed in the scatter plot by the block in the upper right corner, indicating the common range spent on dining out and being out with friends.

### ACHTUNG!

When analyzing data, especially while searching for patterns and correlations, **always** ask yourself if the data makes sense and is trustworthy. What lurking variables could have influenced the data measurements as they were being gathered?

The crime data set from Problem 1 is somewhat suspect in this regard. The number of murders is likely accurate across the board, since murder is conspicuous and highly reported. However, the increase of rape incidents is probably caused both by the general rise in crime as well as an increase in the percentage of rape incidents being reported. Be careful about drawing conclusions for sensitive or questionable data.

Another useful visualization used to understand correlations in a data set is a scatter matrix. The function `pd.plotting.scatter_matrix()` produces a table of plots where each column is plotted against each other column in separate scatter plots. The plots on the diagonal, instead of plotting a column against itself, displays a histogram of that column. This provides a very quick method for an initial analysis of the correlation between different columns.

```
>>> pd.plotting.scatter_matrix(budget[['Living Expenses", "Other"]])
```

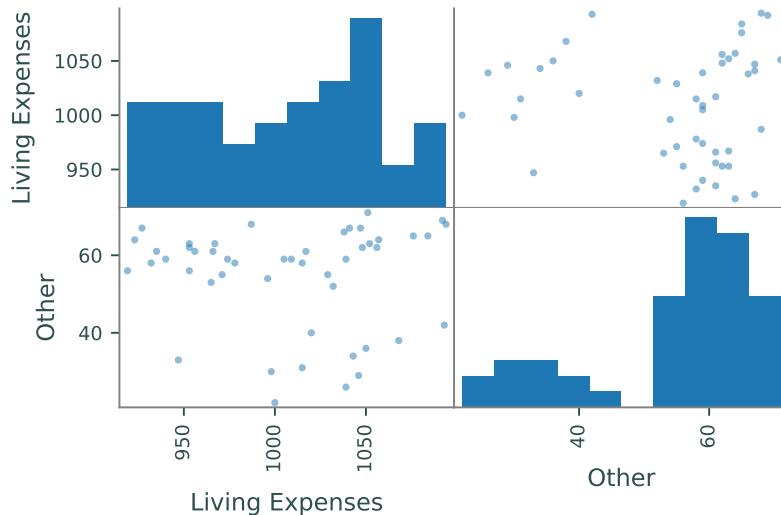


Figure 3.5: Scatter matrix comparing "Living Expenses" and "Other".

## Bar Graphs

Different types of graphs help to identify different patterns. Note that the data set `budget` gives monthly expenses. It may be beneficial to look at one specific month. Bar graphs are a good way to compare small portions of the data set.

As a general rule, horizontal bar charts (`kind="barh"`) are better than the default vertical bar charts (`kind="bar"`) because most humans can detect horizontal differences more easily than vertical differences. If the labels are too long to fit on a normal figure, use `plt.tight_layout()` to adjust the plot boundaries to fit the labels in.

```
# Plot all data for the last month in the budget
>>> budget.iloc[-1, :].plot(kind="barh")
>>> plt.tight_layout()

# Plot all data for the last month without "Rent" and "Living Expenses"
>>> budget.drop(["Rent", "Living Expenses"], axis=1).iloc[-1, :].plot(kind="barh")
>>> plt.tight_layout()
```

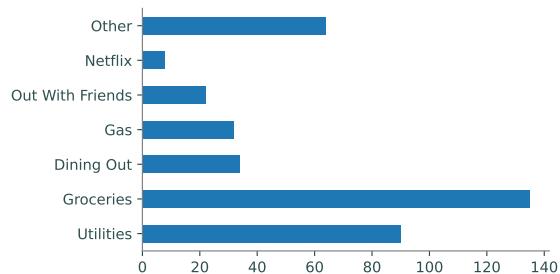
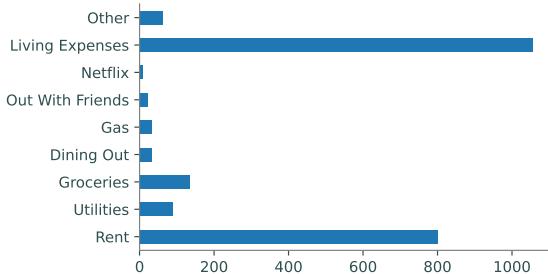


Figure 3.6: Bar graphs showing expenses paid in the last month of `budget`.

**Problem 2.** Using `crime_data.csv`, plot the trends between `Larceny` and the following variables:

1. `Violent`
2. `Burglary`
3. `Aggravated Assault`

Make sure each graph is clearly labeled and readable. One of these variables does *not* have a linear trend with `Larceny`. Return a string identifying this variable (You're welcome to hard-code this value after observing the data).

## Distributional Visualizations

While histograms are good at displaying the distributions for one column, a different visualization is needed to show the distribution of an entire set. A *box plot*, sometimes called a “cat-and-whisker” plot, shows the five number summary: the minimum, first quartile, median, third quartile, and maximum of the data. Box plots are useful for comparing the distributions of relatable data. However, box plots are a basic summary, meaning that they are susceptible to miss important information such as how many points were in each distribution.

```
# Compare the distributions of four columns.
```

```
>>> budget.plot(kind="box", y=["Gas", "Dining Out", "Out With Friends", "Other"])
# Compare the distributions of all columns but "Rent" and "Living Expenses".
>>> budget.drop(["Rent", "Living Expenses"], axis=1).plot(kind="box",
...     vert=False)
```

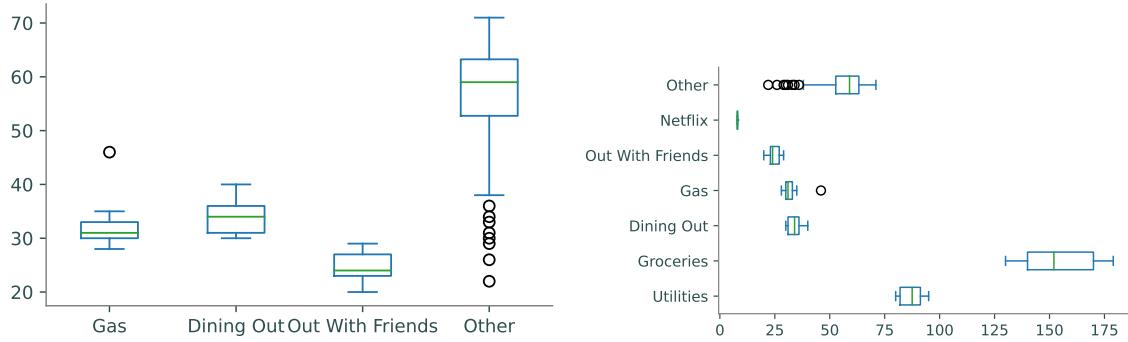


Figure 3.7: Vertical and horizontal box plots of budget dataset.

## Hexbin Plots

A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially creating a 2-dimensional histogram.

The file `satact.csv` contains 700 self reported scores on the SAT Verbal, SAT Quantitative and ACT, collected as part of the Synthetic Aperture Personality Assessment (SAPA) web based personality assessment project. The obvious question with this data set is “how correlated are ACT and SAT scores?” The scatter plot of ACT scores versus SAT Quantitative scores, Figure 3.8a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 3.8b, reveals the **frequency** of points in binned regions.

```
>>> satact = pd.read_csv("satact.csv", index_col="ID")
>>> list(satact.columns)
['gender', 'education', 'age', 'ACT', 'SATV', 'SATQ']

# Plot the ACT scores against the SAT Quant scores in a regular scatter plot.
>>> satact.plot(kind="scatter", x="ACT", y="SATQ", alpha=.8)

# Plot the densities of the ACT vs. SATQ scores with a hexbin plot.
>>> satact.plot(kind="hexbin", x="ACT", y="SATQ", gridsize=20)
```

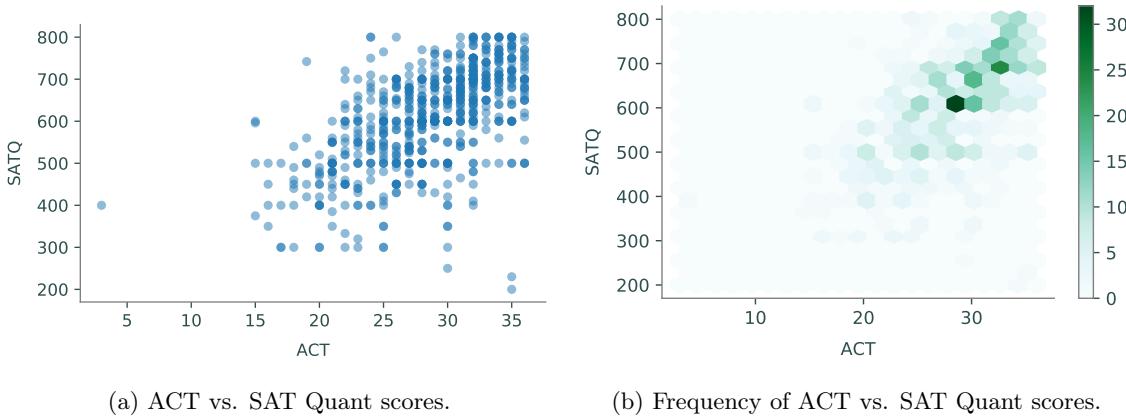


Figure 3.8: Scatter plots and hexbin plot of SAT and ACT scores.

Just as choosing a good number of `bins` is important for a good histogram, choosing a good `gridsize` is crucial for an informative hexbin plot. A large `gridsize` creates many small bins and a small `gridsize` creates fewer, larger bins.

#### NOTE

Since hexbins are based on frequencies, they are prone to being misleading if the dataset is not understood well. For example, when plotting information that deals with geographic position, increases in frequency may be results in higher populations rather than the actual information being plotted.

See <http://pandas.pydata.org/pandas-docs/stable/visualization.html> for more types of plots available in Pandas and further examples.

**Problem 3.** Use `crime_data.csv` to display the following distributions.

1. The distributions of `Burglary`, `Violent`, and `Vehicle Theft` as box plots,
2. The distribution of `Vehicle Thefts` against `Robbery` as a hexbin plot.

As usual, all plots should be labeled and easy to read.

Hint: To get the x-axis label to display, you might need to set the `sharex` parameter of `plot()` to `False`.

## Principles of Good Data Visualization

Data visualization is a powerful tool for analysis and communication. When writing a paper or report, the author must make many decisions about how to use graphics effectively to convey useful information to the reader. Here we will go over a simple process for making deliberate, effective, and efficient design decisions.

## Attention to Detail

Consider the plot in Figure 3.9. It is a scatter plot of positively correlated data of some kind, with `temp`-likely temperature-on the  $x$  axis and `cons` on the  $y$  axis. However, the picture is not really communicating anything about the dataset. It has not specified the units for the  $x$  or the  $y$  axis, nor does it tell what `cons` is. There is no title, and the source of the data is unknown.

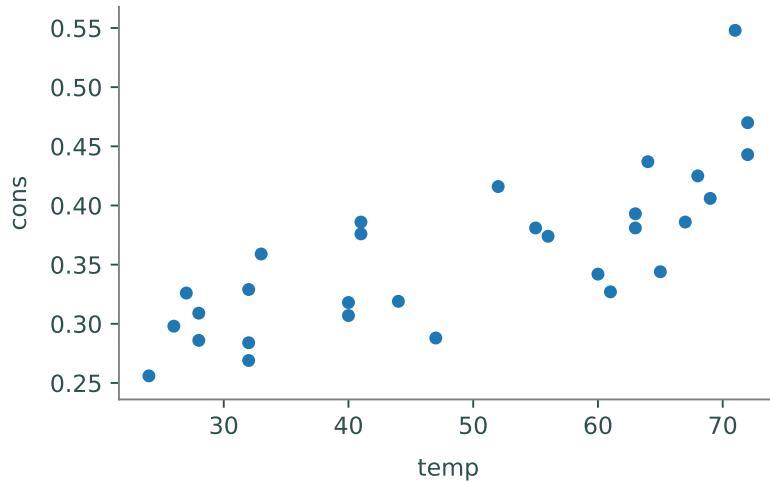


Figure 3.9: Non-specific data.

## Labels and Citations

In a homework or lab setting, we sometimes (mistakenly) think that it is acceptable to leave off appropriate labels, legends, titles, and sourcing. In a published report or presentation, this kind of carelessness is confusing at best and, when the source is not included, even plagiaristic. Data needs to be explained in a useful manner that includes all of the vital information.

Consider again Figure 3.9. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> from pydataset import data
>>> icecream = data("Icecream")
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

This code produces the rather substandard plot in Figure 3.9. Examining the source of the dataset can give important details to create better plots. When plotting data, make sure to understand what the variable names represent and where the data was taken from. Use this information to create a more effective plot.

The ice cream data used in Figure 3.9 is better understood with the following information:

1. The dataset details ice cream consumption via 30 four-week periods from March 1951 to July 1953 in the United States.
2. `cons` corresponds to “consumption of ice cream per capita” and is measured in pints.
3. `income` is the family’s weekly income in dollars.

4. `price` is the price of a pint of ice cream.
5. `temp` corresponds to temperature, degrees Fahrenheit.
6. The listed source is: “Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.”

This information gives important details that can be used in the following code. As seen in previous examples, pandas automatically generates legends when appropriate. Pandas also automatically labels the  $x$  and  $y$  axes, however our data frame column titles may be insufficient. Appropriate titles for the  $x$  and  $y$  axes must also list appropriate units. For example, the  $y$  axis should specify that the consumption is in units of *pints per capita*, in place of the ambiguous label `cons`.

```
>>> icecream = data("Icecream")
# Set title via the title keyword argument
>>> icecream.plot(kind="scatter", x="temp", y="cons",
...     title="Ice Cream Consumption in the U.S., 1951-1953")
# Override pandas automatic labeling using xlabel and ylabel
>>> plt.xlabel("Temp (Fahrenheit)")
>>> plt.ylabel("Consumption per capita (pints)")
```

To add the necessary text to the figure, use either `plt.annotate()` or `plt.text()`. Alternatively, add text immediately below wherever the figure is displayed. The first two parameters of `plt.text` are the  $x$  and  $y$  coordinates to place the text. The third parameter is the text to write. For instance, using `plt.text(0.5, 0.5, "Hello World")` will center the Hello World string in the axes.

```
>>> plt.text(20, .1, r"Source: Hildreth, C. and J. Lu (1960) \emph{Demand}
...     "relations with autocorrelated disturbances}\nTechnical Bulletin No"
...     "2765, Michigan State University.", fontsize=7)
```

Both of these methods are imperfect but can normally be easily replaced by a caption attached to the figure. Again, we reiterate how important it is that you source any data you use; failing to do so is plagiarism.

Finally, we have a clear and demonstrative graphic in Figure 3.10.

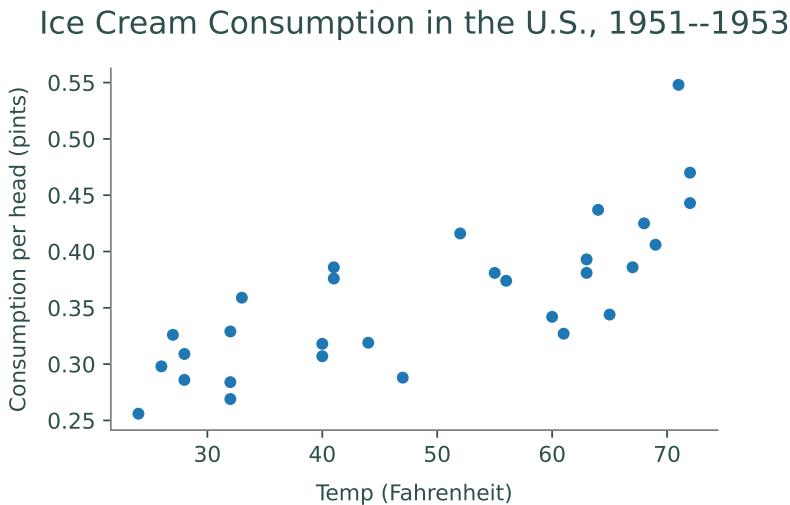


Figure 3.10: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

### ACHTUNG!

Visualizing data can inherit many biases of the visualizer and as a result can be intentionally misleading. Examples of this include, but are not limited to, visualizing subsets of data that do not represent the whole of the data and having purposely misconstrued axes. Every data visualizer has the responsibility to avoid including biases in their visualizations to ensure data is being represented informatively and accurately.

**Problem 4.** The dataset `college.csv` contains information from 1995 on universities in the United States. To access information on variable names, go to <https://cran.r-project.org/web/packages/ISLR/ISLR.pdf>. Create 3 plots that compare variables or universities. These plots should answer questions about the data (e.g. What is the distribution of graduation rates? Do schools with lower student to faculty ratios have higher tuition costs? etc.). These three plots should be easy to understand and have clear titles and variable names. In addition, the dataset should be cited in your first plot as follows:

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) An Introduction to Statistical Learning with applications in R, [www.StatLearning.com](http://www.StatLearning.com), Springer-Verlag, New York

This problem is intended to give you flexibility to create your own visualizations that answer your own questions. As long as your plots are clear and include all the information listed above, you will be graded generously!



# 4

# Pandas 3: Grouping

**Lab Objective:** *Many data sets contain categorical values that naturally sort the data into groups. Analyzing and comparing such groups is an important part of data analysis. In this lab we explore pandas tools for grouping data and presenting tabular data more compactly, primarily through groupby and pivot tables.*

## Groupby

The file `mammal_sleep.csv`<sup>1</sup> contains data on the sleep cycles of different mammals, classified by order, genus, species, and diet (carnivore, herbivore, omnivore, or insectivore). The "`sleep_total`" column gives the total number of hours that each animal sleeps (on average) every 24 hours. To get an idea of how many animals sleep for how long, we start off with a histogram of the "`sleep_total`" column.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

# Read in the data and print a few random entries.
>>> msleep = pd.read_csv("mammal_sleep.csv")
>>> msleep.sample(5)
   name   genus   vore      order  sleep_total  sleep_rem  sleep_cycle
51  Jaguar  Panthera  carni  Carnivora       10.4        NaN        NaN
77  Tenrec    Tenrec  omni  Afrosoricida     15.6        2.3        NaN
10   Goat     Capri  herbi  Artiodactyla      5.3        0.6        NaN
80   Genet    Genetta  carni  Carnivora       6.3        1.3        NaN
33  Human      Homo  omni   Primates        8.0        1.9        1.5

# Plot the distribution of the sleep_total variable.
>>> msleep.plot(kind="hist", y="sleep_total", title="Mammalian Sleep Data")
>>> plt.xlabel("Hours")
```

<sup>1</sup>Proceedings of the National Academy of Sciences, 104 (3):1051–1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia. Available in `pydataset` (with a few more columns) under the key "`msleep`".

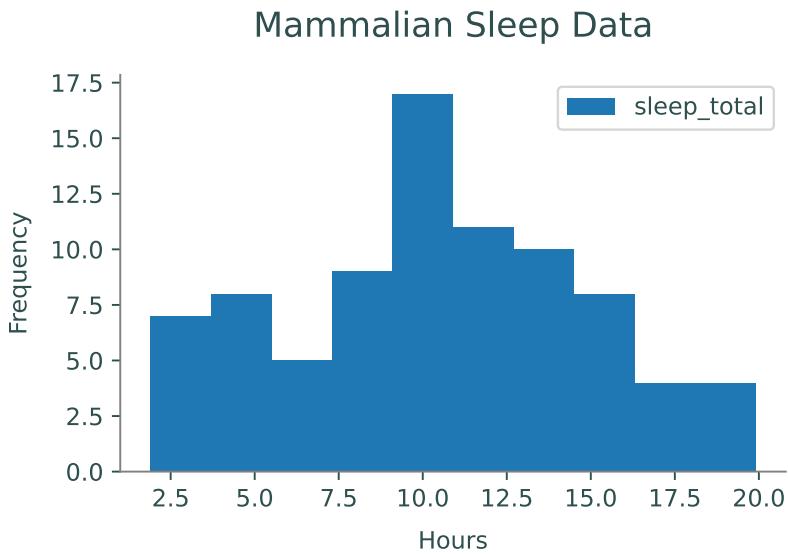


Figure 4.1: "`sleep_total`" frequencies from the mammalian sleep data set.

While this visualization is a good start, it doesn't provide any information about how different kinds of animals have different sleeping habits. How long do carnivores sleep compared to herbivores? Do mammals of the same genus have similar sleep patterns?

A powerful tool for answering these kinds of questions is the `groupby()` method of the pandas `DataFrame` class, which partitions the original `DataFrame` into groups based on the values in one or more columns. The `groupby()` method does **not** return a new `DataFrame`; it returns a pandas `GroupBy` object, an interface for analyzing the original `DataFrame` by groups.

For example, the columns "`genus`", "`vore`", and "`order`" in the mammal sleep data all have a discrete number of categorical values that could be used to group the data. Since the "`vore`" column has only a few unique values, we start by grouping the animals by diet.

```
# List all of the unique values in the "vore" column.
>>> set(msleep["vore"])
{nan, 'herbi', 'omni', 'carni', 'insecti'}

# Group the data by the "vore" column.
>>> vores = msleep.groupby("vore")
>>> list(vores.groups)
['carni', 'herbi', 'insecti', 'omni']           # NaN values for vore were dropped.

# Get a single group and sample a few rows. Note vore="carni" in each entry.
>>> vores.get_group("carni").sample(5)
   name    genus     vore     order  sleep_total  sleep_rem  sleep_cycle
80  Genet  Genetta    carni  Carnivora       6.3        1.3         NaN
50   Tiger  Panthera    carni  Carnivora      15.8        NaN         NaN
8     Dog    Canis     carni  Carnivora      10.1        2.9       0.333
0  Cheetah  Acinonyx    carni  Carnivora      12.1        NaN         NaN
82  Red fox   Vulpes    carni  Carnivora       9.8        2.4       0.350
```

As shown above, `groupby()` is useful for filtering a DataFrame by column values; the command `df.groupby(col).get_group(value)` returns the rows of `df` where the entry of the `col` column is `value`. The real advantage of `groupby()`, however, is how easily it compares groups of data. Standard DataFrame methods like `describe()`, `mean()`, `std()`, `min()`, and `max()` all work on GroupBy objects to produce a new data frame that describes the statistics of each group.

```
# Get averages of the numerical columns for each group.
>>> vores.mean(numeric_only=True)
      sleep_total  sleep_rem  sleep_cycle
vore
carni        10.379     2.290      0.373
herbi        9.509     1.367      0.418
insecti      14.940     3.525      0.161
omni         10.925     1.956      0.592

# Get more detailed statistics for "sleep_total" by group.
>>> vores[["sleep_total"]].describe()
   count    mean     std    min    25%    50%    75%    max
vore
carni    19.0  10.379  4.669   2.7   6.25  10.4  13.000  19.4
herbi    32.0   9.509  4.879   1.9   4.30  10.3  14.225  16.6
insecti    5.0  14.940  5.921   8.4   8.60  18.1  19.700  19.9
omni     20.0  10.925  2.949   8.0   9.10  9.9  10.925  18.0
```

Multiple columns can be used simultaneously for grouping. In this case, the `get_group()` method of the GroupBy object requires a tuple specifying the values for each of the grouping columns.

```
>>> msleep_small = msleep.drop(["sleep_rem", "sleep_cycle"], axis=1)
>>> vores_orders = msleep_small.groupby(["vore", "order"])
>>> vores_orders.get_group(("carni", "Cetacea"))
      name      genus  vore  order  sleep_total
30    Pilot whale  Globicephalus  carni  Cetacea      2.7
59  Common porpoise    Phocoena  carni  Cetacea      5.6
79  Bottle-nosed dolphin    Tursiops  carni  Cetacea      5.2
```

**Problem 1.** Read in the data `college.csv` containing information on various United States universities in 1995. To access information on variable names, go to <https://cran.r-project.org/web/packages/ISLR/ISLR.pdf>. Use a `groupby` object to group the colleges by private and public universities. Read in the data as a DataFrame object and use `groupby` and `describe` to examine the following columns by group:

1. Student to faculty ratio
2. Percent of students from the top 10% of their high school class
3. Percent of students from the top 25% of their high school class

Determine whether private or public universities have a higher mean for each of these columns. For the type of university with the higher mean, save the values of the describe function on said column as an array using `.values`. Return a tuple with these arrays in the order described above.

For example, if we were comparing whether the average number of professors with PhDs was higher at private or public universities, we would find that public universities have a higher average, and we would return the following array:

```
array([212., 76.83490566, 12.31752531, 33., 71., 78.5 , 86., 103.])
```

## Visualizing Groups

There are a few ways that `groupby()` can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time using the `plot` method. The following visualization improves on Figure 4.1 by grouping mammals by their diets.

```
# Plot histograms of "sleep_total" for two separate groups.
>>> vores.get_group("carni").plot(kind="hist", y="sleep_total", legend=False,
                                 title="Carnivore Sleep Data")
>>> plt.xlabel("Hours")
>>> vores.get_group("herbi").plot(kind="hist", y="sleep_total", legend=False,
                                 title="Herbivore Sleep Data")
>>> plt.xlabel("Hours")
```

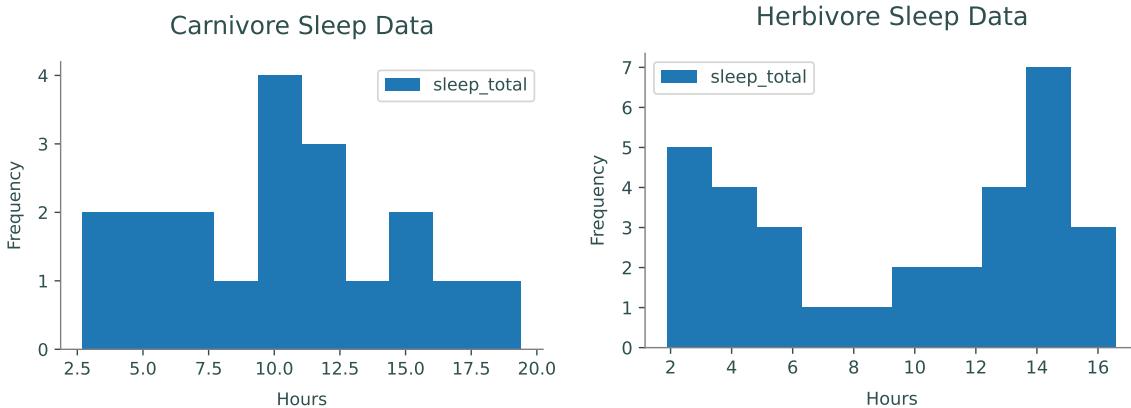
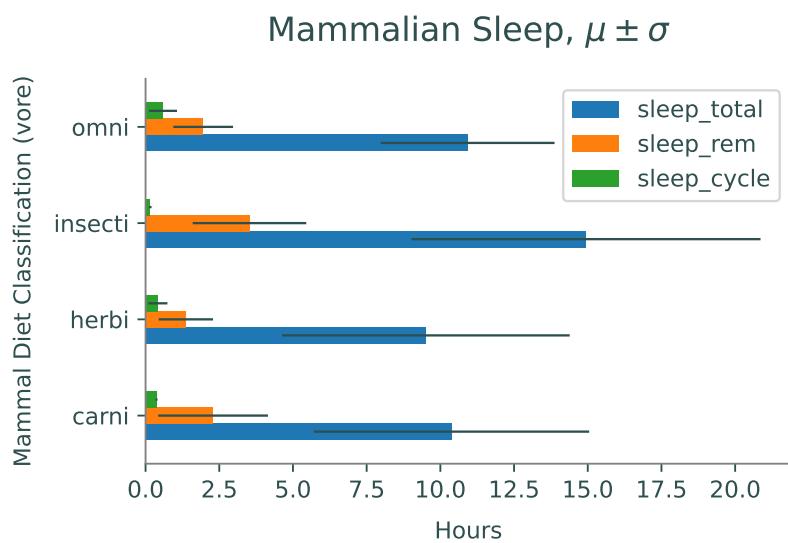


Figure 4.2: "`sleep_total`" histograms for two groups in the mammalian sleep data set.

The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

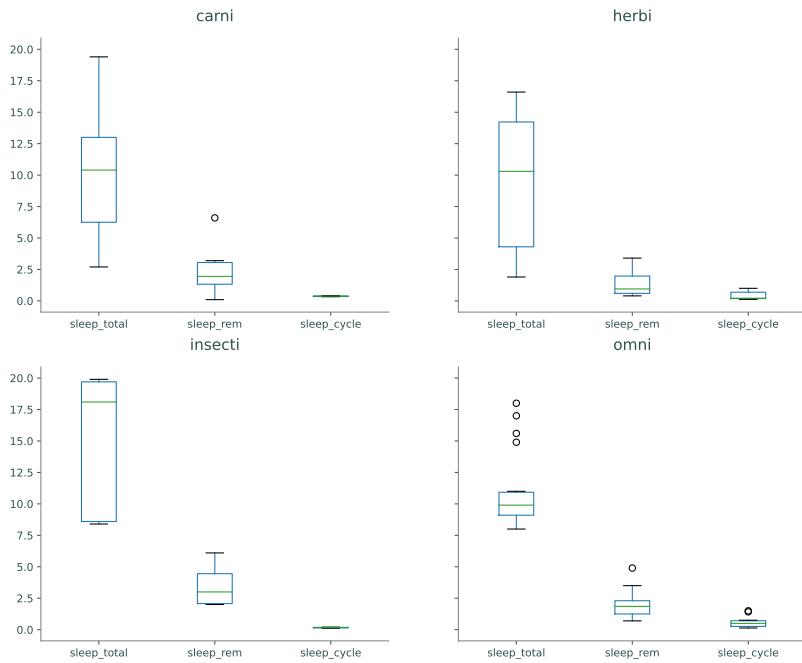
```
>>> cols = ["sleep_total", "sleep_rem", "sleep_cycle"]
>>> vores[cols].mean().plot(kind="barh", xerr=vores[cols].std(),
... title="Mammalian Sleep, $\mu\pm\sigma$")
```

```
>>> plt.xlabel("Hours")
>>> plt.ylabel("Mammal Diet Classification (vore)")
```



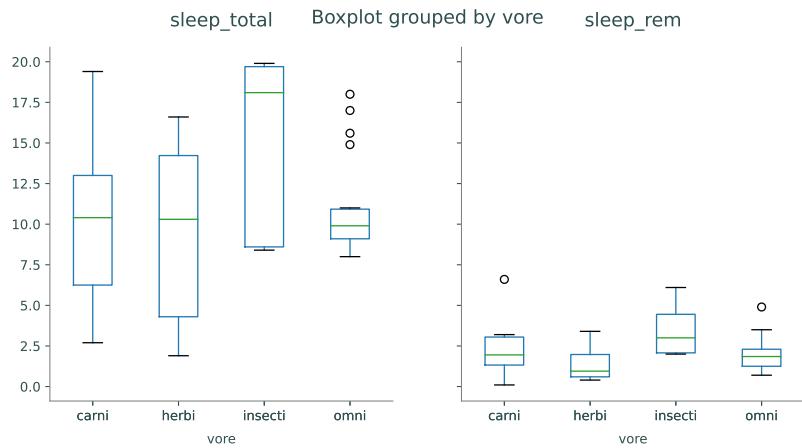
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

```
# Use GroupBy.boxplot() to generate one box plot per group.
>>> vores.boxplot(grid=False)
>>> plt.tight_layout()
```



Alternatively, the `boxplot()` method of the `DataFrame` class creates one subplot **per column**, plotting each of the columns as a box plot. Specify the `by` keyword to group the data appropriately.

```
# Use DataFrame.boxplot() to generate one box plot per column.
>>> msleep.boxplot(["sleep_total", "sleep_rem"], by="vore", grid=False)
```



Like `groupby()`, the `by` argument can be a single column label or a list of column labels. Similar methods exist for creating histograms (`GroupBy.hist()` and `DataFrame.hist()` with `by` keyword), but generally box plots are better for comparing multiple distributions.

**Problem 2.** Create visualizations that give relevant information answering the following questions (using `college.csv`):

1. How do the number of applicants, number of accepted students, and number of enrolled students compare between private and public universities?
2. How does the range of money spent on room and board compare between private and public universities?

## Pivot Tables

One of the downfalls of `groupby()` is that a typical `GroupBy` object has too much information to display coherently. A *pivot table* intelligently summarizes the results of a `groupby()` operation by aggregating the data in a specified way. The standard tool for making a pivot table is the `pivot_table()` method of the `DataFrame` class. As an example, consider the "`HairEyeColor`" data set from `pydataset`.

```
>>> from pydataset import data
>>> hec = data("HairEyeColor")                      # Load and preview the data.
>>> hec.sample(5)
   Hair    Eye     Sex  Freq
3  Red  Brown   Male    10
1 Black  Brown   Male    32
14 Brown  Green   Male    15
31 Red  Green Female     7
21 Black  Blue Female     9

>>> for col in ["Hair", "Eye", "Sex"]:
...     print(f"{col}: {", ".join(set(str(x) for x in hec[col]))}")
...
Hair: Brown, Black, Blond, Red
Eye: Brown, Blue, Hazel, Green
Sex: Male, Female
```

There are several ways to group this data with `groupby()`. However, since there is only one entry per unique hair-eye-sex combination, the data can be completely presented in a pivot table.

```
>>> hec.pivot_table(values="Freq", index=["Hair", "Eye"], columns="Sex")
Sex      Female  Male
Hair  Eye
Black Blue        9    11
        Brown       36    32
        Green        2     3
        Hazel       10    10
Blond Blue       64    30
        Brown       4     3
        Green       8     8
        Hazel       5     5
```

Brown	Blue	34	50
	Brown	66	53
	Green	14	15
	Hazel	29	25
Red	Blue	7	10
	Brown	16	10
	Green	7	7
	Hazel	7	7

Listing the data in this way makes it easy to locate data and compare the female and male groups. For example, it is easy to see that brown hair is more common than red hair and that about twice as many females have blond hair and blue eyes as males.

Unlike "HairEyeColor", many data sets have more than one entry in the data for each grouping. An example in the previous dataset would be if there were two or more rows in the original data for females with blond hair and blue eyes. To construct a pivot table, data of similar groups must be *aggregated* together in some way.

By default entries are aggregated by averaging the non-null values. You can use the keyword argument `aggfunc` to choose among different ways to aggregate the data. For example, if you use `aggfunc="min"`, the value displayed will be the minimum of all the values. Other arguments include `"max"`, `"std"` for standard deviation, `"sum"`, or `"count"` to count the number of occurrences. You also may pass in any function that reduces to a single float, like `np.argmax` or even `np.linalg.norm` if you wish. A list of functions can also be passed into the `aggfunc` keyword argument.

Consider the Titanic data set found in `titanic.csv`<sup>2</sup>. For this analysis, take only the "Survived", "Pclass", "Sex", "Age", "Fare", and "Embarked" columns, replace null age values with the average age, then drop any rows that are missing data. To begin, we examine the average survival rate grouped by sex and passenger class.

```
>>> titanic = pd.read_csv("titanic.csv")
>>> titanic = titanic[["Survived", "Pclass", "Sex", "Age", "Fare", "Embarked"]]
>>> titanic["Age"] = titanic["Age"].fillna(titanic["Age"].mean(),)

>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass")
Pclass      1.0      2.0      3.0
Sex
female    0.965   0.887   0.491
male      0.341   0.146   0.152
```

### NOTE

The `pivot_table()` method is a convenient way of performing a potentially complicated `groupby()` operation with aggregation and some reshaping. The following code is equivalent to the previous example.

```
>>> titanic.groupby(["Sex", "Pclass"])["Survived"].mean().unstack()
Pclass      1.0      2.0      3.0
```

<sup>2</sup>There is a "Titanic" data set in `pydataset`, but it does not contain as much information as the data in `titanic.csv`.

```
Sex
female  0.965  0.887  0.491
male    0.341  0.146  0.152
```

The `stack()`, `unstack()`, and `pivot()` methods provide more advanced shaping options.

Among other things, this pivot table clearly shows how much more likely females were to survive than males. To see how many entries fall into each category, or how many survived in each category, aggregate by counting or summing instead of taking the mean.

```
# See how many entries are in each category.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                      aggfunc="count")
Pclass   1.0   2.0   3.0
Sex
female   144   106   216
male     179   171   493

# See how many people from each category survived.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                      aggfunc="sum")
Pclass     1.0     2.0     3.0
Sex
female   139.0   94.0   106.0
male     61.0    25.0    75.0
```

**Problem 3.** The file `Ohio_1999.csv` contains data on workers in Ohio in the year 1999. Use pivot tables to answer the following questions:

1. Which race/sex combination has the highest `Usual Weekly Earnings` in total?
2. Which race/sex combination has the lowest cumulative `Usual Hours Worked`?
3. What race/sex combination has the highest average `Usual Hours Worked`?

Return a tuple for each question (in order of the questions) where the first entry is the numerical code corresponding to the race and the second entry is corresponding to the sex. You may hard code each tuple as long as you also provide the working code that gave you the solutions.

Some useful keys in understanding the data are as follows:

1. In column `Sex`, {1: `male`, 2: `female`}.
2. In column `Race`, {1: `White`, 2: `African-American`, 3: `Native American/Eskimo`, 4: `Asian`}.

## Discretizing Continuous Data

In the Titanic data, we examined survival rates based on sex and passenger class. Another factor that could have played into survival is age. Were male children as likely to die as females in general? We can investigate this question by *multi-indexing*, or pivoting, on more than just two variables, by adding in another index.

In the original dataset, the "Age" column has a floating point value for the age of each passenger. If we add "Age" as another pivot, then the table would create a new row for **each** age present. Instead, we partition the "Age" column into intervals with `pd.cut()`, thus creating a categorical that can be used for grouping. Notice that when creating the pivot table, the index uses the categorical `age` instead of the column name "Age".

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(0, 4], (0, 4], (0, 4], (0, 4], (4, 8], (4, 8], (4, 8]]
Categories (2, interval[int64, right]): [(0, 4] < (4, 8]]

# Partition the passengers into 3 categories based on age.
>>> age = pd.cut(titanic["Age"], [0, 12, 18, 80])

>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="mean")
Pclass      1.0    2.0    3.0
Sex   Age
female (0, 12]  0.000  1.000  0.467
          (12, 18]  1.000  0.875  0.607
          (18, 80]  0.969  0.871  0.475
male   (0, 12]  1.000  1.000  0.343
          (12, 18]  0.500  0.000  0.081
          (18, 80]  0.322  0.093  0.143
```

From this table, it appears that male children (ages 0 to 12) in the 1st and 2nd class were very likely to survive, whereas those in 3rd class were much less likely to. This clarifies the claim that males were less likely to survive than females. However, there are a few oddities in this table: zero percent of the female children in 1st class survived, and zero percent of teenage males in second class survived. To further investigate, count the number of entries in each group.

```
>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="count")
Pclass      1.0  2.0  3.0
Sex   Age
female (0, 12]    1   13   30
          (12, 18]   12    8   28
          (18, 80]  131   85  158
male   (0, 12]    4   11   35
          (12, 18]    4   10   37
          (18, 80]  171  150  421
```

This table shows that there was only 1 female child in first class and only 10 male teenagers in second class, which sheds light on the previous table.

## ACHTUNG!

The previous pivot table brings up an important point about partitioning datasets. The Titanic dataset includes data for about 1300 passengers, which is a somewhat reasonable sample size, but half of the groupings include less than 30 entries, which is **not** a healthy sample size for statistical analysis. Always carefully question the numbers from pivot tables before making any conclusions.

Pandas also supports multi-indexing on the columns. As an example, consider the price of a passenger tickets. This is another continuous feature that can be discretized with `pd.cut()`. Instead, we use `pd.qcut()` to split the prices into 2 equal quantiles. Some of the resulting groups are empty; to improve readability, specify `fill_value` as the empty string or a dash.

```
# pd.qcut() partitions entries into equally populated intervals.
>>> pd.qcut([1, 2, 5, 6, 8, 3], 2)
[(0.999, 4.0], (0.999, 4.0], (4.0, 8.0], (4.0, 8.0], (4.0, 8.0], (0.999, 4.0]]
Categories (2, interval[float64]): [(0.999, 4.0] < (4.0, 8.0]]

# Cut the ticket price into two intervals (cheap vs expensive).
>>> fare = pd.qcut(titanic["Fare"], 2)
>>> titanic.pivot_table(values="Survived",
    index=["Sex", "age"], columns=[fare, "Pclass"],
    aggfunc="count", fill_value='-')
Fare      (-0.001, 14.454]          (14.454, 512.329]
Pclass
Sex   Age
female (0, 12]           -   -    7           1   13   23
                  (12, 18]          -   4   23           12   4   5
                  (18, 80]          -  31  101          131  54  57
male   (0, 12]           -   -    8           4   11   27
                  (12, 18]          -   5   26           4   5   11
                  (18, 80]          8   94  350          163  56   70
```

Not surprisingly, most of the cheap tickets went to passengers in 3rd class.

**Problem 4.** Use the employment data from Ohio in 1999 to answer the following questions:

1. The column `Educational Attainment` contains numbers 0-46. Any number less than 39 means the person did not get any form of degree. 39-42 refers to either a high-school or associate's degree. A number greater than or equal to 43 means the person got at least a bachelor's degree. Out of these categories, which degree type is the most common among the workers in this dataset?
2. Partition the `Age` column into 6 equally-sized groups using `pd.qcut()`. Which interval has the highest average `Usual Hours Worked`?

3. Using the partitions from the first two parts, what age/degree combination has the lowest yearly salary on average?

Return the answer to each question (in order) as an `Interval`. For part three, the answer should be a tuple where the first entry is the `Interval` of the age and the second is the `Interval` of the degree.

An `Interval` is the object returned by `pd.cut()` and `pd.qcut()`. These can also be obtained from a pivot table, as in the example below.

```
>>> # Create pivot table used in last example with titanic dataset
>>> table = titanic.pivot_table(values="Survived",
                                 index=[age], columns=[fare, "Pclass"],
                                 aggfunc="count")
>>> # Get index of maximum interval
>>> table.sum(axis=1).idxmax()
Interval(0, 12, closed='right')
```

**Problem 5.** Examine the college dataset using `pivot tables` and `groupby` objects. Determine the answer to the following questions. If the answer is yes, save the answer as `True`. If the answer is no, save the answer as `False`. For the last question, save the answer as a string giving your explanation. Return a tuple containing your answers to the questions in order.

1. Partition `perc.alumni` into evenly spaced intervals of 20% using `pd.cut()`. Does the number of both private and public universities decrease as the percentage of alumni that donate increases, as expected?
2. Partition `Grad.Rate` into evenly spaced intervals of 20% using `pd.cut()`. Is the partition with the greatest number of schools the same for private and public universities?
3. Create a column that gives the acceptance rate of each university (using columns `Accept` and `Apps`). Partition this column into evenly spaced intervals of 25% using `pd.cut()`. Is it true that the partition with the least average number of students from the top 10 percent of their high school class is the same for both private and public universities?
4. Use the same partition as part 3. The average percentage of students admitted from the top 10 percent of their high school class is very high in private universities with the lowest acceptance rates (< 25% acceptance rate). Why is this *not* a good conclusion to draw solely from this dataset? Use only the data to explain why; do not extrapolate.

# 5

# Information Theory and Wordle

**Lab Objective:** *Use the information theory concept of entropy to create an algorithm for playing the popular word game Wordle.*

## Wordle

Wordle is a word game<sup>1</sup> where you have 6 guesses to guess a five-letter word. Every time a guess is made, you receive some information about how close your guess is to the correct answer. Letters in the guess that are in the correct location are colored green; letters that are present in the secret word but not in the correct location are colored yellow; and letters that aren't present in the secret word are colored gray. An example game is given in Figure 5.1.



Figure 5.1: An example game of Wordle. Here, the secret word is “train.” Green tiles mean the letter is in the correct location; yellow tiles mean the letter is in the secret word but not at that location; and grey tiles mean the letter is not in the secret word.

In the official version, the secret word is chosen at random from a fixed list of 2309 words. Additionally, there is a list of 12953 words that are allowed to be used as guesses; the guess we make cannot be any arbitrary string of 5 characters, but must always be one of these words. While it is possible to only select guesses that might be the secret word, we can often get more information by making other guesses.

---

<sup>1</sup>It was particularly popular on the internet in 2022.

**Problem 1.** Write a function `get_guess_result()` that accepts a guess and the secret word, and returns the colors of the guess as a list. Label correct letters with the number 2, letters in the wrong location with 1, and incorrect letters with 0.

There are some technicalities with how the guess is colored when multiple of the same letter are present. In order to get these cases correct, have your function follow the following rules:

1. All letters in the guess that are correct in the correct location are marked green.
2. Any other letters in the guess that are in the secret word but not in the right location are marked yellow.
3. However, there will not be more copies of a letter marked yellow or green than there are copies of that letter in the secret word. For instance, if the secret word has one letter `e` and the guess has three, only one of them will be marked yellow or green. The letters are marked yellow from left to right.
4. All other letters are marked gray.

Here are some examples you can use to test your code:

```
>>> get_guess_result("excel", "boxed")
[0, 1, 0, 2, 0]
>>> get_guess_result("stare", "train")
[0, 1, 2, 1, 0]
>>> get_guess_result("green", "pages")
[1, 0, 0, 2, 0]
>>> get_guess_result("abate", "vials")
[0, 0, 2, 0, 0]
>>> get_guess_result("robot", "older")
[1, 1, 0, 0, 0]
```

Hint: Find some way to keep track of which letters in the secret word have been matched to. Since strings are immutable, it may also be helpful to turn the guess and secret word into lists if you need to modify them.

## Computing Guess Results

We will be creating an algorithm to play Wordle. The lists of possible secret words and allowed guesses are in the files `possible_secret_words.txt` and `allowed_guesses.txt`, respectively. These can be loaded using the provided function `load_words()`.

As part of this algorithm, we will need to use the guess results for every pair of possible secret word and allowed guess. Rather than computing these each time, we will compute these once and save the results in an array. For convenience, we have provided this array for you in `all_guess_results.npy`. You can unzip this file using `tar -xf all_guess_results.npy.tar.gz` in the terminal. Once the file has been unzipped, it can be loaded using the function `np.load()`.

Each row of this array corresponds to one of the allowed guess, and each column corresponds to a secret word, in the same order as the wordlists. To make certain computations later work better, each guess has been condensed to a single ternary (base 3) number. This is done by letting each element of the result represent a digit in base 3. For instance, the list  $[1,0,2,2,1]$  becomes the number  $1 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 2 \cdot 3^3 + 1 \cdot 3^4 = 154$ . For an example of using the array, let  $i=1388$  be the index of a given guess (“boxes”) and  $j=1914$  be the index of a given secret word (“steel”). Then we have the following:

```
# The results of guessing "boxes" for every secret word
>>> all_guess_results[i]
array([ 1, 109, 28, ..., 28, 108, 6])
# The results of every guess for the secret word "steel"
>>> all_guess_results[:,j]
array([ 54, 9, 0, ..., 0, 135, 0])
# The result of guessing "boxes" for the secret word "steel"
>>> all_guess_results[i, j]
135
# 135 is equivalent to [0, 0, 0, 2, 1]
```

Our objective is to create some strategy to play Wordle as effectively as possible. Simply choosing the word that is most likely to be the secret word is ineffective, as there is no reason to prefer any word over another as long as both are consistent with the information we have. A much better strategy is to maximize the amount of information each of our guesses gives us, which we will quantify by using entropy<sup>2</sup>.

## Information and Entropy

*Entropy* is the expected amount of information we would gain by knowing the result of a random variable. A natural way to define the information of an event  $A$  is as  $-\log_2 P(A)$ .<sup>3</sup> The entropy of a random variable  $X$ , which we denote  $H(X)$ , is then defined as

$$H(X) = \mathbb{E}[-\log_2 P(X = x)] = - \sum_x P(X = x) \log_2 P(X = x).$$

A loose interpretation is that if a random variable has lower entropy, then we know more about what its value will be even if we haven’t observed it yet, and observing it usually will give little information. At one extreme, if a discrete random variable has zero entropy, then it is necessarily constant. On the other hand, if a random variable has higher entropy, then we know less about its result and observing it typically will give more information. If we know a random variable lives in a certain set, the highest possible entropy it can have is if it is uniformly distributed.

---

<sup>2</sup>This method was explained by Grant Sanderson on his YouTube channel 3Blue1Brown. The video can be found here.

<sup>3</sup>This choice of definition has a number of desirable properties for information: information is non-negative, the information of two independent events is the sum of their individual informations, and information is a continuous function of the probability of an event. In fact, this is the *only* function with this property, up to choice of logarithm base; refer to the Volume 3 textbook for more details. The base-2 logarithm is often used because it represents the number of bits needed to encode the information.

For Wordle, since we don't know the secret word, it is reasonable to consider it as a random variable. This gives the secret word a value of entropy, which we can use to choose a guess that is likely to give more information. Denote the secret word as  $W$  and the result of making a guess  $g$  as  $R(g)$ . For any guess, the result  $R(g)$  of the guess is entirely determined by  $W$ ; since we don't know the secret word, this makes  $R(g)$  also a random variable.

There are two approaches we can take to make a strategy out of this. First, we can try to minimize the entropy of the variable  $W|R(g)$ . This essentially is trying to find the guess that will on average minimize how much we don't know about the secret word after we know the result of the guess. Second, we can try to maximize the entropy of the variable  $R(g)$ . This amounts to finding which guess is expected to give the most information.

These two approaches are in fact equivalent, as

$$H(W|R(g)) = H((W, R(g))) - H(R(g)) = H(W) - H(R(g)),$$

where  $H((W, R(g)))$  denotes the entropy of the joint random variable  $(W, R(g))$  (*not* the cross entropy). To see this equality, note that for random variables  $X, Y$  we have

$$-\log_2 P(X|Y) = -\log_2 \frac{P(X, Y)}{P(Y)} = -\log_2 P(X, Y) + \log_2 P(Y).$$

Taking the expectation of both sides implies that

$$H(X|Y) = H((X, Y)) - H(Y).$$

Additionally, the value of  $R(g)$  is completely determined by  $W$ , so  $H((W, R(g))) = H(W)$ . The entropy of  $R(g)$  ends up being more straightforward to calculate, so this is the approach we take for the remainder of the lab.

We now seek to calculate the entropy of  $R(g)$ , the result of the guess, for each guess  $g$  we can make. This is given by

$$\begin{aligned} H(R(g)) &= -\sum_r P(R(g) = r) \log_2 P(R(g) = r) \\ &= -\sum_r P(R(g, W) = r) \log_2 P(R(g, W) = r). \end{aligned}$$

Since we assumed a uniform distribution over the set of possible secret words, the probability  $P(R(g, W) = r)$  is the proportion of secret words that yield the same result  $r$  given the same guess  $g$ .

To find the entropy of a guess, we thus need only to compute the probability of each unique guess result, and then apply the equation above. This sum will need to be evaluated for each individual guess that we can make.

As an example, suppose that we know the secret word is one of the words "boney", "disco", "marsh", "stock", or "visor", and we are evaluating the guess "boxes". The result of this guess for each of these words is as follows:

Secret word	Guess result
boney	(2,2,0,2,0)
disco	(0,1,0,0,1)
marsh	(0,0,0,0,1)
stock	(0,1,0,0,1)
visor	(0,1,0,0,1)

There are three distinct possible results: (2,2,0,2,0), with probability  $\frac{1}{5}$ ; (0,1,0,0,1), with probability  $\frac{3}{5}$ ; and (0,0,0,0,1), with probability  $\frac{1}{5}$ . Using the above formula gives the entropy of this guess as

$$-\frac{1}{5} \log_2 \frac{1}{5} - \frac{3}{5} \log_2 \frac{3}{5} - \frac{1}{5} \log_2 \frac{1}{5} \approx 1.3710$$

**Problem 2.** Write a function that accepts the array of all guess results (as in `all_guess_results.npy`) and calculates the entropy of each guess. Return the guess with the highest entropy.

Hint: `np.unique` with the argument `return_counts=True` will return an array with the number of occurrences of each of the different values in a one-dimensional array. By looping over each allowed guesses, you can use this function to compute the entropy quickly. Beware that applying this function directly to multidimensional arrays results in different behavior, however.

After we make a guess, we want to find the posterior distribution for the secret word given the guesses we've made. Bayes' Rule gives

$$P(W = w|R(g) = r) = \frac{P(R(g) = r|W = w)P(W = w)}{P(R(g) = r)}.$$

First, we look at the term  $P(R(g) = r|W = w)$ . If we know the secret word  $W$ , then for any guess  $g$ , the result  $R(g)$  is uniquely determined. Thus, this probability is either 0 or 1, depending on whether the guess result we observed is the result that would be seen if  $w$  is the secret word. For instance, with the secret word  $w$  = "steel" and the guess  $g$  = "boxes", we have

$$P(R(g) = r|W = w) = \begin{cases} 1 & r = [0, 0, 0, 2, 1] \\ 0 & \text{otherwise} \end{cases};$$

that is, the only value of  $r$  for which the probability is not zero is  $r = [0, 0, 0, 2, 1]$ , which is the result of making that guess.

Now,  $P(W = w)$  is a constant, and  $P(R(g) = r)$  is constant for all secret words that have  $P(R(g) = r|W = w) \neq 0$ , since these all have the same value of  $R(g)$ . So, the posterior distribution is just a uniform distribution over the set of possible secret words that give  $R(g_{\text{made}}) = r_{\text{observed}}$ , i.e. the same result for our guess as we observed. Finding the optimal next guess to make is then equivalent to repeating the same process as before with a smaller initial list of possible secret words.

**Problem 3.** Create a function that filters the list of possible secret words after making a guess.

Accept the array of all guess results (as in `all_guess_results.npy`), the list of allowed guesses, the list of possible secret words, the guess that was made, and the guess's result (as a list of integers). Return a filtered list of possible words that are still possible after knowing the result of a guess. Also return a filtered version of the array of all guess results that only contains the results for the secret words still possible after making the guess. This smaller array will be used in later steps of the game.

If our guess is "boxes" and the guess result is `[0,0,0,2,1]`, then the list of remaining words should have length 47 and the array of guess results should have shape `(12953, 47)`.

Hint: to find the index of a word in either of the wordlists, use the `.index()` function.

Hint part 2: The most efficient way to do this problem is with *boolean masking*. If `A` is any numpy array and `mask` is a 1-D array of True/False values, then `A[mask]` will return the portion of `A` where `mask` is true. This can be used even if `A` is multidimensional, and on dimensions other than the first; for instance, `A[:,mask]` will use the mask for the second dimension of the array.

### NOTE

Note that although we filter down the list of possible secret words, we do not do anything similar for the list of allowed guesses. As the game goes on and we make more guesses, the list of words that could still be the secret word shrinks, while the list of words we are allowed to guess stays the same. Sometimes words that we know cannot possibly be the secret word might give us more information than words which might be the secret word, so it can be beneficial to guess them anyways.

Before we assemble our algorithm for playing Wordle, we would like a benchmark. A simple strategy to compare to is to select an allowed guess at random until we know the secret word.

**Problem 4.** The file `wordle.py` contains a class called `WordleGame` that can be used to simulate games of Wordle.<sup>a</sup> Instantiate one of these, use the `start_game()` function to start a game, and use the `make_guess()` function to make a guess.

Write a function that accepts a `WordleGame` and starts and plays a game using the strategy of randomly selecting words. At each step of the game:

- If we know the secret word (our list of possible guesses has length 1), guess that word.
- Otherwise, choose a guess at random from the list of allowed guesses.
- Filter the list of possible words using your function from Problem 3 to only those that are still possible knowing the result of the guess.
- Repeat until the secret word has been guessed.

Use `game.is_finished()` to check if the game has been finished. Return the number of guesses needed to guess the secret word (including guessing the word, not just determining it).

To visualize this algorithm, pass the argument `display=True`, and the `WordleGame` will print out each word as it is guessed.

---

<sup>a</sup>This class uses the `colorama` package to format terminal output. If needed, it can easily be installed with `pip install colorama`.

**Problem 5.** Write a function that accepts a `WordleGame` object and starts and plays a game using the strategy of maximizing the entropy of each guess. At each step:

- If we know the secret word (our list of possible secret words has length 1), guess that word
- Otherwise, compute the entropies using your function from Problem 2, and make the guess that has the highest entropy
- Filter the possible secret words using your function from Problem 3 to only those that we still know are possible
- Repeat until the secret word has been guessed

Use `game.is_finished()` to check at each step if the game has been finished. Return the number of guesses needed to guess the secret word.

**Problem 6.** Write a function that accepts an integer  $n$  and simulates that many games of Wordle using each of the above algorithms. Return the average number of guesses each required to find the secret word. Compare their performance; the approach using the entropy should require about half as many guesses on average.

The `WordleGame` object also has a version you can play in the terminal, which can be started using the `play_game_interactive()` method. You can use this to also compare your own performance to that of your algorithm.



# 6

# GeoPandas

**Lab Objective:** *GeoPandas is a package designed to organize and manipulate geographic data. It combines the data manipulation tools of pandas with the geometric capabilities of the Shapely package. In this lab, we explore the basic data structures of GeoSeries and GeoDataFrames and their functionalities.*

## Installation

GeoPandas is a new package designed to combine the functionality of pandas with Shapely, a package used for geometric manipulation. Using GeoPandas with geographic data is very useful as it allows the user to not only compare numerical data, but also geometric attributes. GeoPandas can be installed via pip:

```
>>> pip install geopandas
```

## GeoSeries

A GeoSeries is a pandas Series where each entry is a set of geometric objects. There are three classes of geometric objects inherited from the Shapely package:

1. Points / Multi-Points
2. Lines / Multi-Lines
3. Polygons / Multi-Polygons

A point is used to identify objects like coordinates, where there is one small instance of the object. A line could be used to describe objects such as roads. A polygon could be used to identify regions, such as a country. Multipoints, multilines, and multipolygons contain lists of points, lines, and polygons, respectively.

Since each object in the GeoSeries is also a Shapely object, the GeoSeries inherits many methods and attributes of Shapely objects. Some of the key attributes and methods are listed in Table 6.1. These attributes and methods can be used to calculate distances, find the sizes of countries, and determine whether coordinates are within country's boundaries. The example below uses the attribute `bounds` to find the maximum and minimum coordinates of Egypt in the GeoDataFrame `worldmap.gpkg`.

Method/Attribute	Description
<code>distance(other)</code>	returns minimum distance from GeoSeries to <code>other</code>
<code>contains(other)</code>	returns <code>True</code> if shape contains <code>other</code>
<code>intersects(other)</code>	returns <code>True</code> if shape intersects <code>other</code>
<code>area</code>	returns shape area
<code>convex_hull</code>	returns convex shape around all points in the object
<code>bounds</code>	returns the bounding x- and y-coordinates of the object

Table 6.1: Attributes and Methods for GeoSeries

```
>>> import geopandas as gpd
>>> world = gpd.read_file(worldmap.gpkg)
# Get GeoSeries for Egypt
>>> egypt = world[world["SOVEREIGNT"]=="Egypt"]

# Find bounds of Egypt
>>> egypt.bounds
```

# Creating GeoDataFrames

The main structure used in GeoPandas is a GeoDataFrame, which is similar to a pandas DataFrame. A GeoDataFrame has one special column called `geometry`, which must be a GeoSeries. This GeoSeries column is used when a spatial method, like `distance()`, is used on the GeoDataFrame. Therefore all attributes and methods used for GeoSeries can also be used on GeoDataFrame objects.

A GeoDataFrame can be made from a pandas DataFrame. One of the columns in the DataFrame should contain geometric information. That column can be converted to a GeoSeries using the `apply()` method. At this point, the Pandas DataFrame can be cast as a GeoDataFrame. Assign which column will be the `geometry` using either the `geometry` keyword in the constructor or the `set_geometry()` method afterwards.

```

...
    "Longitude": [126.98, -77.04, 28.04]})

# Create geometry column
>>> df["Coordinates"] = list(zip(df.Longitude, df.Latitude))

# Make geometry column Shapely objects
>>> df["Coordinates"] = df["Coordinates"].apply(Point)

# Cast as GeoDataFrame
>>> gdf = gpd.GeoDataFrame(df, geometry="Coordinates")

# Equivalently, specify the geometry after construction
# Note that set_geometry() returns a new GeoDataFrame
>>> gdf = gpd.GeoDataFrame(df)
>>> gdf = gdf.set_geometry("Coordinates")

# Display the GeoDataFrame
>>> gdf
      City      Country  Latitude  Longitude           Coordinates
0    Seoul  South Korea     37.57    126.98  POINT (126.98000 37.57000)
1      Lima        Peru    -12.05    -77.04  POINT (-77.04000 -12.05000)
2  Johannesburg  South Africa    -26.20     28.04  POINT (28.04000 -26.20000)

# Create a polygon with all three cities as points
>>> city_polygon = Polygon(list(zip(df.Longitude, df.Latitude)))

```

A `GeoDataFrame` can also be made directly from a dictionary. If the dictionary already contains geometric objects, the corresponding column can be directly set as the `geometry` in the constructor. Otherwise, a column containing geometry data can be created as in the above example and then set as the `geometry` with the `set_geometry()` method.

```

# Both of these methods create the same GeoDataFrame as above
# Directly create the GeoDataFrame from the dictionary
>>> gdf = gpd.GeoDataFrame({"City": ["Seoul", "Lima", "Johannesburg"],
...                           "Country": ["South Korea", "Peru", "South Africa"],
...                           "Latitude": [37.57, -12.05, -26.20],
...                           "Longitude": [126.98, -77.04, 28.04]})

# Create geometry column and set as the geometry
>>> gdf["Coordinates"] = list(zip(gdf.Longitude, gdf.Latitude))
>>> gdf["Coordinates"] = gdf["Coordinates"].apply(Point)
# inplace=True modifies gdf itself rather than returning a copy
>>> gdf.set_geometry("Coordinates", inplace=True)

# Equivalently, using a dictionary that already contains geometry objects
>>> gdf = gpd.GeoDataFrame({"City": ["Seoul", "Lima", "Johannesburg"],
...                           "Country": ["South Korea", "Peru", "South Africa"],
...                           "Coordinates": [Point(126.98, 37.57),
...                                         Point(-77.04, -12.05), Point(28.04, -12.05)],
...                           geometry="Coordinates"})

```

Method/Attribute	Description
<code>abs()</code>	returns series/dataframe with absolute numeric value of each element
<code>add(other)</code>	returns addition of dataframe and <code>other</code> element-wise
<code>affine_transform(matrix)</code>	returns <code>GeoSeries</code> with translated geometries
<code>append(other)</code>	returns new object with appended rows of <code>other</code> to the end of caller
<code>dot(other)</code>	returns dataframe of matrix multiplication with <code>other</code>
<code>equals(other)</code>	tests if the two objects contain the same elements

Table 6.2: Attributes and Methods for GeoDataFrame

**NOTE**

Longitude is the angular measurement starting at the Prime Meridian,  $0^\circ$ , and going to  $180^\circ$  to the east and  $-180^\circ$  to the west. Latitude is the angle between the equatorial plane and the normal line at a given point; a point along the Equator has latitude 0, the North Pole has latitude  $+90^\circ$  or  $90^\circ N$ , and the South Pole has latitude  $-90^\circ$  or  $90^\circ S$ .

## Plotting GeoDataFrames

Information from a GeoDataFrame is plotted based on the geometry column. Data points are displayed as geometry objects. The following example plots the shapes in the `world` GeoDataFrame.

```
# Plot world GeoDataFrame
>>> world.plot()
```

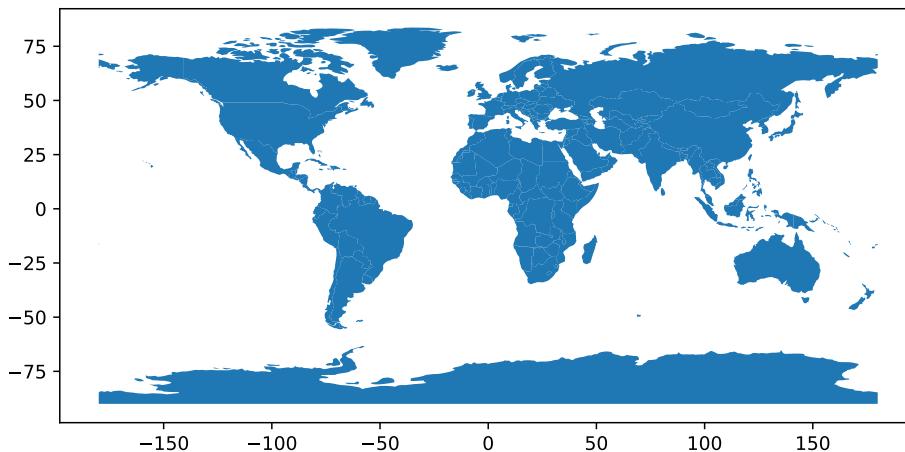


Figure 6.1: World map

Multiple GeoDataFrames can be plotted at once. This can be done by setting one GeoDataFrame as the base of the plot and ensuring that each layer uses the same axes. In the following example, the file `airports.csv`, containing the coordinates of world airports, is loaded into a GeoDataFrame and plotted on top of the boundary of the `world` GeoDataFrame.

```
# Set outline of world countries as base
>>> fig, ax = plt.subplots(1, figsize=(10, 4))
>>> base = world.boundary.plot(edgecolor="black", ax=ax, linewidth=1)

# Load airport data and convert to a GeoDataFrame
>>> airports = pd.read_csv("airports.csv")
>>> airports["Coordinates"] = list(zip(airports.Longitude, airports.Latitude))
>>> airports["Coordinates"] = airports.Coordinates.apply(Point)
>>> airports = gpd.GeoDataFrame(airports, geometry="Coordinates")

# Plot airports on top of world map
>>> airports.plot(ax=base, marker='o', color="green", markersize=1)
>>> ax.set_xlabel("Longitude")
>>> ax.set_ylabel("Latitude")
>>> ax.set_title("World Airports")
```

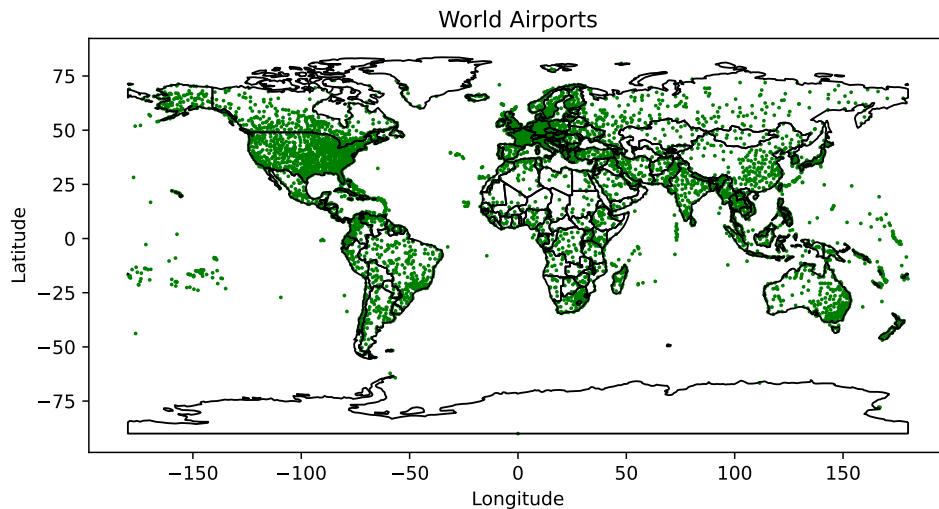


Figure 6.2: Airport map

**Problem 1.** The file `worldmap.gpkg.zip` contains data of the worldmap used in above examples.<sup>a</sup> After unzipping, use the command `geopandas.read_file("worldmap.gpkg")` to create a GeoDataFrame of this information.

Then, read in the file `airports.csv` as a pandas DataFrame. Create three convex hulls around the three sets of airports listed below. This can be done by passing in lists of the airports' coordinates (Longitude and Latitude zipped together) to a `shapely.geometry.Polygon` object.

Finally, create a new GeoDataFrame using a dictionary with key "`geometry`" and with a list of these three Polygons as the value. Plot this GeoDataFrame, and then plot the outlined world map on top of it.

- Maio Airport, Scatsta Airport, Stokmarknes Skagen Airport, Bekily Airport, K. D. Matanzima Airport, RAF Ascension Island
- Oiapoque Airport, Maio Airport, Zhezkazgan Airport, Walton Airport, RAF Ascension Island, Usiminas Airport, Pilojo Osvaldo Marques Dias Airport
- Zhezkazgan Airport, Khanty Mansiysk Airport, Novy Urengoy Airport, Kalay Airport, Biju Patnaik Airport, Walton Airport

<sup>a</sup>Source: <https://www.naturalearthdata.com/downloads/110m-cultural-vectors/>

#### NOTE

.gpkg files are actually structured as a directory that contains several files that each contain parts of the data. For instance, `worldmap.gpkg` consists of the files `worldmap.cpg`, `worldmap.dbf`, `worldmap.prj`, `worldmap.shp`, and `worldmap.shx`. Be sure that these files are placed directly in the first level of `worldmap.gpkg`, and not in further subdirectories.

## Working with GeoDataFrames

As previously mentioned, GeoDataFrames contain many of the functionalities of pandas DataFrames. For example, to create a new column, define a new column name in the GeoDataFrame with the needed information for each GeoSeries.

```
# Create column in the world GeoDataFrame for GDP_PER_CAPITA
>>> world["GDP_PER_CAP"] = world.GDP_MD / world.POP_EST
```

GeoDataFrames can utilize many pandas functionalities, and they can also be parsed by geometric manipulations. For example, a useful way to index GeoDataFrames is with the `cx` indexer. This splits the GeoDataFrame by the coordinates of each geometric object. It is used by calling the method `cx` on a GeoDataFrame, followed by a slicing argument, where the first element refers to the longitude and the second refers to latitude.

```
# Create a GeoDataFrame containing the northern hemisphere
>>> north = world.cx[:, 0:]

# Create a GeoDataFrame containing the southeastern hemisphere
>>> south_east = world.cx[0:, :0]
```

GeoSeries objects in a GeoDataFrame can also be dissolved, or merged, together into one GeoSeries based on their geometry data. For example, all countries on one continent could be merged to create a GeoSeries containing the information of that continent. The method designed for this is called `dissolve`. It receives two parameters, `by` and `aggfunc`. `by` indicates which column to dissolve along, and `aggfunc` tells how to combine the information in all other columns. The default `aggfunc` is `first`, which returns the first application entry. In the following example, we use `sum` as the `aggfunc` so that each continent is the combination of its countries.

```
>>> world = world[["CONTINENT", "geometry", "GDP_PER_CAP"]]

# Dissolve world GeoDataFrame by continent
>>> continent = world.dissolve(by="CONTINENT", aggfunc="sum")
```

## Projections and Coloring

When plotting, GeoPandas uses the CRS (coordinate reference system) of a GeoDataFrame. This reference system indicates how coordinates should be spaced on a plot. Two of the most commonly used CRSs are EPSG:4326 and EPSG:3395. EPSG:4326 is the standard latitude-longitude projection used by GPS. EPSG:3395, also known as the Mercator projection, is the standard navigational projection.

When creating a new GeoDataFrame, it is important to set the `crs` attribute of the GeoDataFrame. This allows any plots to be shown correctly. Furthermore, GeoDataFrames being layered need to have the same CRS. To change the CRS, use the method `to_crs()`.

```
# Check CRS of world GeoDataFrame
>>> print(world.crs)
EPSG:4326

# Change CRS of world to Mercator
# inplace=True ensures that we modify world instead of returning a copy
>>> world.to_crs(3395, inplace=True)
>>> print(world.crs)
EPSG:3395
```

GeoPandas accepts many different CRSs; a reference can be found at [www.spatialreference.org](http://www.spatialreference.org). Additionally, inspecting a given CRS object in the terminal without using `print()` or `str()` can be used to get additional information about a specific CRS:<sup>1</sup>

```
>>> world.crs
<Projected CRS: EPSG:3395>
Name: WGS 84 / World Mercator
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: World between 80°S and 84°N.
```

---

<sup>1</sup>This can also be accomplished using `print(repr(crs))`.

```
- bounds: (-180.0, -80.0, 180.0, 84.0)
Coordinate Operation:
- name: World Mercator
- method: Mercator (variant A)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

GeoDataFrames can also be plotted using the values in the other attributes of the GeoSeries. The map plots the color of each geometry object according to the value of the column selected. This is done by passing in the parameter `column` into the `plot()` method.

```
>>> fig, ax = plt.subplots(1, figsize=(10, 4))
# Plot world based on gdp
>>> world.plot(column="GDP_MD", cmap="OrRd", legend=True, ax=ax)
>>> ax.set_title("World Map based on GDP")
>>> ax.set_xlabel("Longitude")
>>> ax.set_ylabel("Latitude")
>>> plt.show()
```

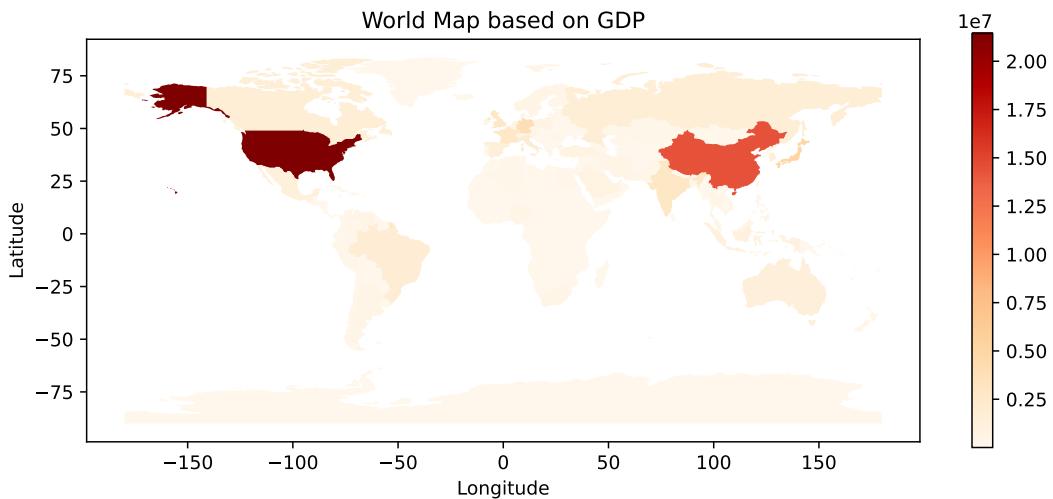


Figure 6.3: World Map Based on GDP

**Problem 2.** The file `county_data.gpkg.zip` contains information about US counties.<sup>a</sup> After unzipping, use the command `geopandas.read_file("county_data.gpkg")` to create a GeoDataFrame of this information. Each county's shape is stored in the `geometry` column. Use this to plot the boundaries of all US counties two times, first using the default CRS and then using EPSG:5071.

Next, create a new GeoDataFrame that combines (`dissolve`) all counties within each state (`by="STATEFP"`). Drop regions with the following STATEFP codes: 02, 15, 60, 66, 69, 72, 78. Plot the boundary of this GeoDataFrame to see an outline of all 48 contiguous states. Ensure a CRS of EPSG:5071.

<sup>a</sup>Source: [http://www2.census.gov/geo/tiger/GENZ2016/shp/cb\\_2016\\_us\\_county\\_5m.zip](http://www2.census.gov/geo/tiger/GENZ2016/shp/cb_2016_us_county_5m.zip)

## Merging GeoDataFrames

Just as multiple pandas DataFrames can be merged, multiple GeoDataFrames can be merged with attribute joins or spatial joins. An attribute join is similar to a merge in pandas. It combines two GeoDataFrames on a column (not the geometry column) and then combines the rest of the data into one GeoDataFrame.

```
>>> world = gpd.read_file(worldmap.gpkg)
>>> cities = gpd.read_file(cities.gpkg)

# Create subsets of the world and cities GeoDataFrames
>>> world = world[["CONTINENT", "SOVEREIGNT", "SOV_A3"]]
>>> cities = cities[["NAME", "SOV_A3"]]

# Merge the GeoDataFrames on their SOV_A3 code
>>> countries = world.merge(cities, on="SOV_A3")
```

A spatial join merges two GeoDataFrames based on their geometry data. The function used for this is `sjoin`. `sjoin` accepts two GeoDataFrames and then direction on how to merge. It is imperative that two GeoDataFrames have the same CRS. In the example below, we merge using an `inner` join with the option `intersects`. The `inner` join means that we will only use keys in the intersection of both geometry columns, and we will retain only the left geometry column. `intersects` tells the GeoDataFrames to merge on GeoSeries that intersect each other. Other options include `contains` and `within`.

```
# Combine countries and cities on their geographic location
>>> countries = gpd.sjoin(world, cities, how="inner", predicate="intersects")
```

**Problem 3.** Load in the file `nytimes.csv`<sup>a</sup> as a Pandas DataFrame. This file includes county-level data for the cumulative cases and deaths of Covid-19 in the US, starting with the first case in Snohomish County, Washington, on January 21, 2020.

Merge the county GeoDataFrame from `county_data.gpkg` with the `nytimes` DataFrame on the county `fips` codes (a FIPS code is a 5-digit unique identifier for geographic locations). Note that the `fips` column of the `nytimes` DataFrame stores entries as floats, but the county GeoDataFrame stores FIPS codes as strings, with the first two digits in the `STATEFP` column and the last three digits in the `COUNTYFP` column. Thus, you will need to add these two columns together and then convert them into floats so they can be merged with the `fips` column in the `nytimes` DataFrame.

Drop the regions from the county GeoDataFrame with the same STATEFP codes as in Problem 2. Also, make sure to change the CRS of the county GeoDataFrame to EPSG:5071 before you merge the two DataFrames (this will make the code run much faster).

Plot the cases from March 21, 2020, and then plot your state outline map from Problem 2 on top of that (with a CRS of EPSG:5071). Include a colorbar using the arguments `legend=True` and `cmap="plasma_r"` in the `plot` function. Finally, print out the name of the county with the most cases on March 21, 2020, along with its case count.

Hint: every state should have multiple covid cases.

<sup>a</sup>Source: <https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv>

## Logarithmic Plotting Techniques

The color scheme of a graph can also help to communicate information clearly. A good list of available colormaps can be found at [https://matplotlib.org/3.2.1/gallery/color/colormap\\_reference.html](https://matplotlib.org/3.2.1/gallery/color/colormap_reference.html). Note also that you can reverse any colormap by adding `_r` to the end. The following example demonstrates some plotting features, using country GDP as in Figure 6.3.

```
>>> fig, ax = plt.subplots(1, figsize=(10, 4))
>>> world.plot(column="GDP_MD", cmap="plasma_r",
...             ax=ax, legend=True, edgecolor="gray")

# Add title and remove axis tick marks
>>> ax.set_title("GDP on Linear Scale")
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

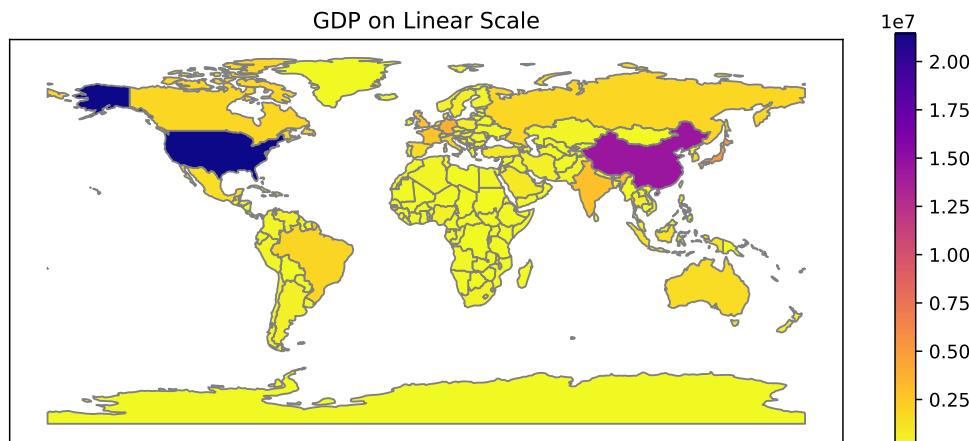


Figure 6.4: World map showing country GDP

Sometimes data can be much more informative when plotted on a logarithmic scale. See how the world map changes when we add a `norm` argument in the code below. Depending on the purpose of the graph, Figure 6.5 may be more informative than Figure 6.4.

```
>>> from matplotlib.colors import LogNorm
>>> from matplotlib.cm import ScalarMappable
>>> fig, ax = plt.subplots(1, figsize=(10, 7))

# Set the norm using data bounds
>>> data = world.GDP_MD
>>> norm = LogNorm(vmin=min(data), vmax=max(data))

# Plot the graph using the norm
>>> world.plot(column="GDP_MD", cmap="plasma_r", ax=ax,
...             edgecolor="gray", norm=norm)

# Create a custom colorbar
>>> cbar = fig.colorbar(ScalarMappable(norm=norm, cmap="plasma_r"),
...                      ax=ax, orientation="horizontal", pad=0, label="GDP")

>>> ax.set_title("GDP on a Log Scale")
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

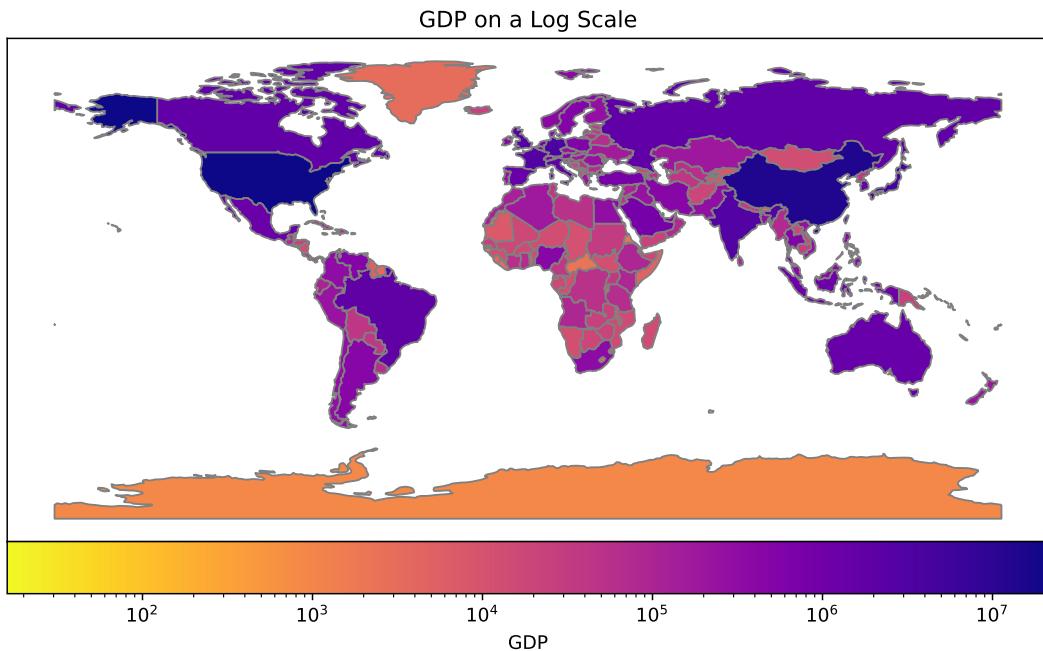


Figure 6.5: World map showing country GDP using a log scale

**Problem 4.** As in Problem 3, plot your state outline map from Problem 2 on top of a map of Covid-19 cases from March 21, 2020 (each with a CRS of EPSG:5071). This time, however, use a log scale. Pick a good colormap (the counties with the most cases should generally be darkest) and be sure to display a colorbar.

**Problem 5.** In this problem, you will create an animation of the spread of Covid-19 through US counties from January 21, 2020, through June 21, 2020. You will use the same GeoDataFrame you used in Problems 3 and 4 (with a CRS of EPSG:5071). Use a log scale and a good colormap, and be sure that you're using the same norm and colorbar for the whole animation.

As a reminder, below is a summary of what you will need in order to animate this map. You may also find it helpful to refer to the animation section included with the Volume 4 lab manual.

1. Set up your figure and norm. Be sure to use the highest case count for your `vmax` so that the scale remains uniform.
2. Write your `update` function. This should plot the cases from a given day as well as the state boundaries.
3. Set up your colorbar. Do this outside the `update` function to avoid adding a new colorbar each day.
4. Create a `FuncAnimation` object. Check to make sure everything displays properly before you save it.
5. Save the animation to a file, and embed it into the notebook.

# 7

# PCA, LSI, and Scikit-Learn

**Lab Objective:** *Understand the basics of principal component analysis and latent semantic indexing. Learn more about scikit-learn and implement a machine learning pipeline.*

## Principal Component Analysis

Principal Component Analysis (PCA) is a multivariate statistical tool used to change the basis of a set of samples from the basis of original features (which may be correlated) into a basis of uncorrelated variables called the *principal components*. It is a direct application of the singular value decomposition (SVD). The first principal component will account for the greatest variance in the samples, the second principal component will be orthogonal to the first and account for the second greatest variance, etc. By projecting the samples onto the space spanned by the first few principal components, we can reduce the dimensionality of the data while preserving most of the variance.

Take a matrix  $X$  with samples as rows and features as columns. The first step in PCA is to pre-process the data, which usually includes translating the columns of  $X$  to have mean 0. Some datasets require additional scaling based on variance and units of measurement. Call the new pre-processed matrix  $Y$ .

We next compute the truncated SVD of our centered data,  $Y = U\Sigma V^T$ , where the columns of  $V$  are the principal components and form an orthonormal basis for the space spanned by the samples. The variance captured by each principal component can be calculated by the equation below, where  $\sigma_i$  is the  $i$ -th nonzero singular value and there are  $k$  total singular values.

$$\frac{\sigma_i^2}{\sum_{j=1}^k \sigma_j^2} \quad (7.1)$$

In general, we are only interested in the first several principal components. But just how many principal components should we keep? One method is to keep the first two principal components so that we can project the data into 2-dimensional space. Another is to only keep the set of principal components accounting for a certain percentage of the variance, using the equation above.

Once we have decided how many principal components to keep (say the first  $l$ ), we can project the samples from the original feature space onto the principal component space by computing

$$\hat{Y} = U_{:,l}\Sigma_{:l,:l} = YV_{:,l}$$

**Problem 1.** The breast cancer dataset from scikit-learn has 569 samples with 30 features each. Each sample is labeled as 0 (malignant) or 1 (benign). With 30 features, this data can't be directly visualized, so we will use PCA to graph the first two principal components, which account for nearly all of the variance in the data.

Write a function that performs PCA on the breast cancer dataset using the SVD as described above. Graph the first two principal components, with the first along the  $x$ -axis. Your graph should resemble Figure 7.1 below. Include the proportion of the total variance that the first two principal components capture in the graph title, calculated with Equation 7.1.

You can load this dataset using the following code:

```
>>> cancer = sklearn.datasets.load_breast_cancer()
>>> X = cancer.data
>>> y = cancer.target # Class labels (0 or 1)
```

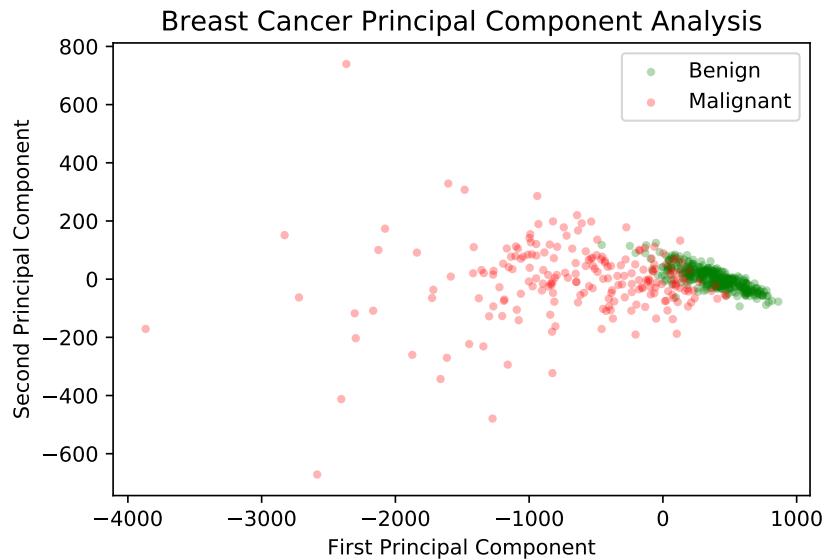


Figure 7.1: First two principal components of the transformed breast cancer data

## Latent Semantic Indexing

*Latent Semantic Indexing* (LSI) is an application of PCA to the realm of natural language processing. In particular, LSI employs the SVD to reduce the dimensionality of a large corpus of text documents in order to enable us to evaluate the similarity between two documents. Many information-retrieval systems used in government and in industry are based on LSI.

To motivate the problem, suppose we have a large collection of documents about various topics. How can we find an article about BYU? We might consider simply choosing the article that contains the acronym the greatest number of times, but this is a crude method. A better way is to use a form of PCA on the collection of documents.

In order to do so, we need to represent the documents as numerical vectors. A standard way of doing this is to define an ordered set of words occurring in the collection of documents (called the *vocabulary*) and then to represent each document as a vector of word counts from the vocabulary. More formally, let our vocabulary be  $V = \{w_1, w_2, \dots, w_m\}$ . Then a document is a vector  $x = (x_1, x_2, \dots, x_m) \in \mathbb{R}^m$  such that  $x_i$  is the number of occurrences of word  $w_i$  in the document. In this setup, we represent the entire collection of  $m$  documents as an  $n \times m$  matrix  $X$ , where  $m$  is the number of vocabulary words and  $n$  is the number of documents in our collection, so each row is a document vector. As expected, we let  $X_{i,j}$  be the number of times term  $j$  occurs in document  $i$ . Note that  $X$  is often a sparse matrix, as any single document likely does not contain most of the vocabulary words. This mode of representation is called the *bag of words* model for documents.

We calculate the SVD of  $X$  *without* centering or scaling the data so that we may retain the sparsity. This is unique to this particular problem. We now have  $X = U\Sigma V^\top$ . If we are keeping  $l$  principal components, we can represent the corpus of documents by the matrix

$$\hat{X} = U_{:,l}\Sigma_{l,:}V_{:,l} = X V_{:,l}$$

Note that  $\hat{X}$  will no longer be a sparse matrix, but will have dimension  $n \times l$ .

Now that we have our documents represented in terms of the first  $l$  principal components, we can find the similarity between two documents. Our measure for similarity is simply the cosine of the angle between the vectors; a small angle (large cosine) indicates greater similarity, while a large angle (small cosine) indicates greater dissimilarity. Recall that we can use the inner product to find the cosine of the angle between two vectors. Under this metric, the similarity between document  $i$  and document  $j$  (represented by the  $i$ -th and  $j$ -th row of  $\hat{X}$ , notated  $\hat{X}_i$  and  $\hat{X}_j$ , respectively) is just

$$\frac{\langle \hat{X}_i, \hat{X}_j \rangle}{\|\hat{X}_i\| \|\hat{X}_j\|}.$$

To find the document most similar to document  $i$ , we simply compute

$$\operatorname{argmax}_{j \neq i} \frac{\langle \hat{X}_i, \hat{X}_j \rangle}{\|\hat{X}_i\| \|\hat{X}_j\|}.$$

**Problem 2.** Create a function `similar` that takes in a numpy array `Xhat` and an index `i` and returns the indices of the most similar and the least similar documents.

Hint: `np.argsort` may be useful for finding which are the most and least similar. Note that every document will have a similarity score of 1 with itself, so be careful not to return a document as its own closest document.

To test your code, use the following matrix:

```
X = np.array([
    [0.78, 0.14, 0.12, 0.],
    [0.64, 0.97, 0., 0.],
    [0., 0., 0.63, 0.46],
    [0., 0.84, 0.6, 0.],
    [0.29, 0.89, 0.51, 0.],
    [0.77, 0., 0.27, 0.2],
    [0.86, 0.47, 0., 0.06],
    [0.89, 0., 0., 0.]])
```

])

With `i=4` your function should output the following:

```
>>> print(similar(4, X))
(3, 7)
```

## Application: State of the Union

We now discuss some practical issues involved in creating the bag of words representation  $X$  from the raw text. Our dataset will consist of the US State of the Union addresses from 1945 through 2013, each contained in a separate text file in the folder `Addresses`. We would like to avoid loading in all of the text into memory at once, and so we will *stream* the documents one at a time.

The first thing we need to establish is the vocabulary set, i.e. the set of unique words that occur throughout the collection of documents. A Python set object automatically preserves the uniqueness of the elements, so we will create a set and then iteratively read through the documents, adding the unique words of each document to the set. As we read in each document, we will remove punctuation and numerical characters and convert everything to lower case. The following code, found in the function `document_converter()`, will accomplish this task.

```
# Get list of filepaths to each text file in the folder.
folder = "./Addresses/"
paths = sorted([folder+p for p in os.listdir(folder) if p.endswith(".txt")])

# Helper function to get list of words in a string.
def extractWords(text):
    ignore = string.punctuation + string.digits
    cleaned = "".join([t for t in text.strip() if t not in ignore])
    return cleaned.lower().split()

# Initialize vocab set, then read each file and add to the vocab set.
vocab = set()
for p in paths:
    with open(p, 'r', encoding="utf8") as infile:
        for line in infile:
            vocab.update(extractWords(line)) # Union sets together
```

We now have a set containing all of the unique words in the corpus. However, many of the most common words do not provide important information. We call these *stop words*. Examples in English include *the*, *a*, *an*, *and*, *I*, *we*, *you*, *it*, *there*, etc; a list of common English stop words is given in `stopwords.txt`. We remove the stop words from our vocabulary set as follows and then fix an ordering to the vocabulary by creating a dictionary with key-value pairs of the form (word, index).

```
# Load stopwords
with open("stopwords.txt", 'r', encoding="utf8") as f:
    stops = set([w.strip().lower() for w in f.readlines()])
```

```
# Remove stopwords from vocabulary, create ordering
vocab = {w:i for i, w in enumerate(vocab.difference(stops))}
```

We are now ready to create the word count vectors for each document, and we store these in a sparse matrix  $X$ . It is convenient to use the `Counter` object from the `collections` module, as this object automatically counts the occurrences of each distinct element in a list.

```
from collections import Counter

counts = []      # holds the entries of X
doc_index = []   # holds the row index of X
word_index = []  # holds the column index of X

# Iterate through the documents
for doc, p in enumerate(paths):
    with open(p, 'r', encoding="utf8") as f:
        # Create the word counter.
        ctr = Counter()
        for line in f:
            ctr.update(extractWords(line))
        # Iterate through the word counter, storing counts.
        for word, count in ctr.items():
            if word in vocab:
                word_index.append(vocab[word])
                counts.append(count)
                doc_index.append(doc)

# Create sparse matrix holding these word counts.
X = sparse.csr_array((counts, [doc_index, word_index]),
                      shape=(len(paths), len(vocab)), dtype=float)
```

**Problem 3.** Applying the techniques of LSI discussed above to the word count matrix  $X$ , created with the `document_converter()` function, and keeping the first 7 principal components, write a function that takes in the path to a single State of the Union address `speech` and returns a tuple of the addresses that are most and least similar to `speech`.

For Ronald Reagan's 1984 speech, the input would be `"./Addresses/1984-Reagan.txt"`, and your output should be `("1988-Reagan", "1946-Truman")`. Be sure to format the strings properly. Also run your algorithm on Clinton's 1993 speech, and display your output.

Since  $X$  is a sparse matrix, you will need to use the SVD method found in `scipy.sparse.linalg`. This method operates slightly differently than the SVD method found in `scipy.linalg`, so be sure to read the documentation. Also pass the argument `random_state=28` into this function for consistency.

The simple bag of words representation is a bit crude, as it fails to consider how some words may be more important than others in determining the similarity of documents. Words appearing in few documents tend to provide more information than words occurring in every document. For example, while the word *war* might not be considered a stop word, it is likely to appear in many more addresses than the word *Afghanistan*. Two speeches sharing the word *Afghanistan* are probably more closely related than two speeches sharing the word *war*. So while  $X_{i,j}$  is a good measure of the importance of term  $j$  in document  $i$ , we also need to consider some kind of global weight for each term  $j$ , indicating how important the term is over the entire collection. There are a number of different weights we could choose; we will employ the following approach. Define

$$p_{i,j} = \frac{X_{i,j}}{\sum_{\bar{j}} X_{i,\bar{j}}}.$$

We then let

$$g_j = 1 + \sum_{i=1}^m \frac{p_{i,j} \log(p_{i,j} + 1)}{\log m},$$

where  $m$  is the number of documents in the collection. We call  $g_j$  the *global weight* of term  $j$ . We replace each term frequency in the matrix  $X$  by weighting it globally. Specifically, we define a matrix  $A$  with entries

$$A_{i,j} = g_j \log(X_{i,j} + 1).$$

We can now perform LSI on the matrix  $A$ , whose entries are both locally and globally weighted.

**Problem 4.** Use the equation above to create the function `weighted_document_converter()` to calculate the sparse matrix  $A$ . Similar to the function `document_converter()`, this function should return  $A$  and a list of file paths.

Hint: the function `np.log1p`, which calculates  $\log(1+x)$ , can be applied to a sparse matrix without losing sparsity.

## Scikit-Learn

Scikit-learn is one of the fundamental tools Python offers for machine learning. It includes classifiers, such as `RandomForestClassifier` and `KNeighborsClassifier`, as well as transformers, which preprocess data before classification. In the remainder of this lab, we will discuss transformers, validation tools, how to find optimal hyperparameters, and how to build a machine learning pipeline.

## Transformers

A scikit-learn *transformer* processes data to make it better suited for classification. This may involve shifting or scaling data, dropping columns, replacing missing values, and so on. The function from Problem 4 is an example of a transformer, as is PCA.

NOTE

A *hyperparameter* is not dependent on data. Hyperparameters are declared in the constructor `__init__()`, before data is even passed in. Parameters set during the `fit()` method are often called *model parameters* and do depend on specific data. For example, a `StandardScaler` transformer shifts and scales data to have a mean of 0 and a standard deviation of 1.

Scikit-learn's transformers have three main methods: `fit_transform()`, which fits model parameters and also transforms given data; `fit()`, which sets model parameters but does not perform a transformation; and `transform()`, which transforms data according to pre-fitted model parameters. Model parameters are fitted according to training data, and they are not refitted to testing data, so a `StandardScaler` will shift and scale testing data according to the mean and variance of the training data; the transformed test data likely will not have mean 0 and variance 1.

Scikit-learn has a built-in PCA package. Its hyperparameters include the desired number of principal components and the type of SVD solver to use. Its `fit_transform()` method takes in an array of data and returns the decomposition with `n_components`.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=5) # Create the PCA transformer with hyperparameters
>>> Xhat = pca.fit_transform(X) # Fit the transformer and transform the data
```

For our particular application, however, since our matrix is sparse and Scikit-learn's PCA class does not accept sparse matrices, we need to use their `TruncatedSVD` class instead. The syntax for this class is identical to the above.

#### NOTE

An interesting observation can be made by repeating Problem 1 using scikit-learn's PCA package. The resulting graph will have the *x*-axis flipped, because a matrix's singular value decomposition is unique up to multiplication by -1.

**Problem 5.** Repeat Problem 3 using your weighted document converter function and scikit-learn's `sklearn.decomposition.TruncatedSVD` class. For consistency, also pass the argument `random_state=74` in the constructor to this class.

For Bill Clinton's 1993 speech, your code should return ("`1992-Bush`", "`1946-Truman`"). Also run the algorithm on Reagan's 1984 speech, and display the results.

## Validation Tools

We now turn our attention from transformers to classifiers. A *classifier* is trained to predict how a new sample should be classified or labeled. Knowing how to determine whether or not a classifier performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the possible labels are only 0 or 1.

The `score()` method of a scikit-learn classifier returns the *accuracy* of the model, or the percent of labels predicted correctly. However, accuracy isn't always the best measure of success. Consider the *confusion matrix* for a classifier, the matrix where the  $(i, j)$ th entry is the number of samples with actual label  $i$  but that are classified with label  $j$ . Call the class with label 0 the *negatives* and the class with label 1 the *positives*. Then the confusion matrix is as follows.

$$\begin{array}{cc} & \text{Predicted: 0} & \text{Predicted: 1} \\ \text{Actual: 0} & \left[ \begin{array}{cc} \text{True Negatives (TN)} & \text{False Positives (FP)} \end{array} \right] \\ \text{Actual: 1} & \left[ \begin{array}{cc} \text{False Negatives (FN)} & \text{True Positives (TP)} \end{array} \right] \end{array}$$

With this terminology, we define the following metrics.

- *Accuracy*:  $\frac{TN + TP}{TN + FN + FP + TP}$ , the percent of labels predicted correctly.
- *Precision*:  $\frac{TP}{TP + FP}$ , the percent of predicted positives that are actually correct.
- *Recall*:  $\frac{TP}{TP + FN}$ , the percent of actual positives that are predicted correctly.

Precision is useful in situations where false positives are dangerous or costly, while recall is important when avoiding false negatives takes priority. For example, an email spam filter should avoid filtering out an email that isn't actually spam; here a false positive is more dangerous, so precision is a valuable metric for the filter. On the other hand, recall is more important in disease detection: it is better to test positive and not have the disease than to test negative when the disease is actually present. Focusing on a single metric often leads to skewed results, so the following metric is also common.

$$F_\beta \text{ Score} : (1 + \beta^2) \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \frac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2FN}.$$

Choosing  $\beta < 1$  weighs precision more than recall, while  $\beta > 1$  prioritizes recall over precision. The choice of  $\beta = 1$  yields the common  $F_1$  score, which weighs precision and recall equally. This is an important alternative to accuracy when, for example, the training set is heavily unbalanced with respect to the class labels.

Scikit-learn implements all of these metrics in `sklearn.metrics`. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. We will be using the function `classification_report()`, which returns precision, recall, and  $F_1$  scores for each label. Each row in the report corresponds to a specific label and gives the scores with its label as the "positive" classification. For example, in binary classification, the row corresponding to 1 gives the scores as they would normally be calculated, with 1 as "positive."

```

>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.metrics import confusion_matrix, classification_report
>>> from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)

# Fit the estimator to training data and predict the test labels.
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> knn.fit(X_train, y_train)
>>> knn_predicted = knn.predict(X_test)

# Compute the confusion matrix by comparing actual labels to predicted labels.
>>> CM = confusion_matrix(y_test, knn_predicted)
>>> CM
array([[44,  5],
       [10, 84]])

# Get precision, recall, and F1 scores all at once.
# The row labeled 1 gives these scores as we normally calculate them.
>>> print(classification_report(y_test, knn_predicted))
      precision    recall  f1-score   support

          0       0.81      0.90      0.85       49
          1       0.94      0.89      0.92       94

   accuracy                           0.90      143
  macro avg       0.88      0.90      0.89      143
weighted avg       0.90      0.90      0.90      143

```

**Problem 6.** For this problem, use the cancer dataset from Problem 1 to compare a `RandomForestClassifier` and a `KNeighborsClassifier`, using the default parameters for each with `random_state=43` for the `RandomForestClassifier`.

Use `train_test_split()` with `random_state=2` to split up the data. Fit the classifiers with the training set and predict the labels for the testing set. Print out a classification report for each classifier, making sure to clearly label which report corresponds to which classifier.

Write a few sentences explaining which of these classifiers would be better to use in this situation and why, using the information from the report as evidence. Remember that in this dataset, the label 1 means benign and 0 means malignant.

## Grid Search

Finding the optimal hyperparameters for a given model is a challenging and active area of research.<sup>1</sup> However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with a classifier, a dictionary of hyperparameters, and some validation parameters. When its `fit()` method is called, it tests the given classifier with every possible hyperparameter combination.

For example, a `KNeighborsClassifier` has a few important hyperparameters that can have a significant impact on the speed and accuracy of the model. These include `n_neighbors`, the number of nearest neighbors allowed to vote, and `weights`, which specifies a strategy for weighting the distances between points. The code box below tests various combinations of these hyperparameters.

The cost of a grid search rapidly increases as the hyperparameter space grows. However, the outcomes of each trial are completely independent of each other, so the problem of training each classifier is *embarrassingly parallel*, meaning the trials can easily be computed simultaneously. To parallelize the grid search over  $n$  CPU cores, set the `n_jobs` parameter to  $n$ , or set it to  $-1$  to divide the labor between as many cores as are available.

```
>>> from sklearn.model_selection import GridSearchCV

>>> knn = KNeighborsClassifier()
# Specify values for certain hyperparameters
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...                 "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, scoring="f1", n_jobs=-1)

# Run the actual search. This may take some time.
>>> knn_gs.fit(X_train, y_train)

# After fitting, you can access data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_, sep='\n')
{'n_neighbors': 5, 'weights': 'uniform'}
0.9532526583188765

# Access the model
>>> knn_gs.best_estimator_
KNeighborsClassifier(weights='distance')
```

In some circumstances, the parameter grid can be organized in a way that eliminates redundancy. For example, with a `RandomForestClassifier`, you could test each `max_depth` argument with entirely different sets of values for `min_samples_leaf`. To specify certain combinations of parameters, enter the parameter grid as a list of dictionaries.

**Problem 7.** Do a grid search on the breast cancer dataset using a `RandomForestClassifier`. Modify at least three parameters in your grid. Use `scoring="f1"` for the `GridSearchCV` object. Fit your model with the same train-test split as in Problem 6. Print out the best parameters and the best score.

---

<sup>1</sup>Intelligent hyperparameter selection is sometimes called *metalearning*.

Next, use the `GridSearchCV` object to predict labels for your test set. Print out a confusion matrix using these values.

### ACHTUNG!

Because grid searches can be computationally expensive to perform, it is recommended to debug your code using a small subset of data to ensure everything is functioning correctly before performing a grid search on the full data.

## Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* chains together one or more transformers and one estimator (such as a classifier) into a single object, complete with `fit()` and `predict()` methods. This simplifies and automates the machine learning process so that when you get new data or make changes to various functions and features, you can easily rerun the new version from beginning to end.

The following example demonstrates how to use a pipeline with a `StandardScaler` transformer and a `KNeighborsClassifier`. Like classifiers, pipelines have `fit()`, `predict()`, and `score()` methods. Each member of the pipeline is declared as a tuple where the first element is a string naming the step and the second is the actual transformer or classifier.

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.pipeline import Pipeline

# Chain together a StandardScaler transformer and a KNN classifier.
>>> pipe = Pipeline([('scaler', StandardScaler()), # "scaler" is the step name
...                   ('knn', KNeighborsClassifier())]) # "knn" is the step name
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972
```

Since `Pipeline` objects follow `fit()` and `predict()` conventions, they can be used with tools like `GridSearchCV`. To specify which hyperparameters belong to which steps of the pipeline, precede each hyperparameter name with `<stepname>__`. For example, `knn__n_neighbors` corresponds to the `n_neighbors` hyperparameter of the pipeline step labeled `knn`.

```
# Create the Pipeline, labeling each step.
>>> pipe = Pipeline([('scaler', StandardScaler()),
...                   ('knn', KNeighborsClassifier())])

# Specify the hyperparameters to test for each step.
>>> pipe_param_grid = {"scaler__with_mean": [True, False],
...                     "scaler__with_std": [True, False],
...                     "knn__n_neighbors": [2, 3, 4, 5, 6],
...                     "knn__weights": ["uniform", "distance"]}
```

```
# Pass the Pipeline object to the GridSearchCV and fit it to the data.  
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,  
...                           n_jobs=-1).fit(X_train, y_train)  
  
>>> print(pipe_gs.best_params_, pipe_gs.best_score_, sep='\n')  
{'knn__n_neighbors': 6, 'knn__weights': 'distance',  
 'scaler__with_mean': True, 'scaler__with_std': True}  
0.971830985915493
```

Pipelines can also be used to compare different transformations or estimators. For example, a pipeline can end in either a `KNeighborsClassifier()` or a classifier called `SVC()`, even though they have different hyperparameters. Like before, you can use a list of dictionaries to specify the specific combinations of the hyperparameter space.

**Problem 8.** The breast cancer dataset has 30 features. By using PCA, we can drastically reduce the dimensionality while still retaining predictive power.

Create a pipeline with a `StandardScaler`, `PCA`, and a `KNeighborsClassifier`. Use the same train-test split as before. Do a grid search on this pipeline, modifying at least six hyperparameters and using `scoring="f1"`. Use no more than 5 principal components. Print out your best parameters and best score. Attain a score of at least .96.

*Hint:* The documentation for `StandardScaler`, `PCA`, and `KNeighborsClassifier` can be found at these links.





# K-Means Clustering

**Lab Objective:** *Clustering is the one of the main tools in unsupervised learning—machine learning problems where the data comes without labels. In this lab we implement the k-means algorithm, a simple and popular clustering method, and apply it to geographic clustering and color quantization.*

## Clustering

In this lab, we will analyze a few different datasets from Scikit-Learn’s library and use the *k-means* algorithm. Figure 8.1 is a graph of the iris dataset. As a human, it is easy to identify the two distinct groups of data. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*. The *k-means* algorithm is a simple way of helping computers see the group distinctions.

The objective of clustering is to find a partitions of the data such that points in the same subset will be “close” according to some metric. The metric used will likely depend on the data, but some obvious choices include Euclidean distance and angular distance. Throughout this lab, we will use the metric  $d(x, y) = \|x - y\|_2$ , the Euclidean distance between  $x$  and  $y$ , unless we specify a different metric to be used.

More formally, suppose we have a collection of  $\mathbb{R}^K$ -valued observations  $X = \{x_1, x_2, \dots, x_n\}$ . Let  $N \in \mathbb{N}$  and let  $\mathcal{S}$  be the set of all  $N$ -partitions of  $X$ , where an  $N$ -partition is a partition with exactly  $N$  nonempty elements. We can represent a typical partition in  $\mathcal{S}$  as  $S = \{S_1, S_2, \dots, S_N\}$ , where

$$X = \bigcup_{i=1}^N S_i$$

and

$$|S_i| > 0, \quad i = 1, 2, \dots, N.$$

We seek the  $N$ -partition  $S^*$  that minimizes the within-cluster sum of squares, i.e.

$$S^* = \arg \min_{S \in \mathcal{S}} \sum_{i=1}^N \sum_{x_j \in S_i} \|x_j - \mu_i\|_2^2,$$

where  $\mu_i$  is the mean of the elements in  $S_i$ , i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

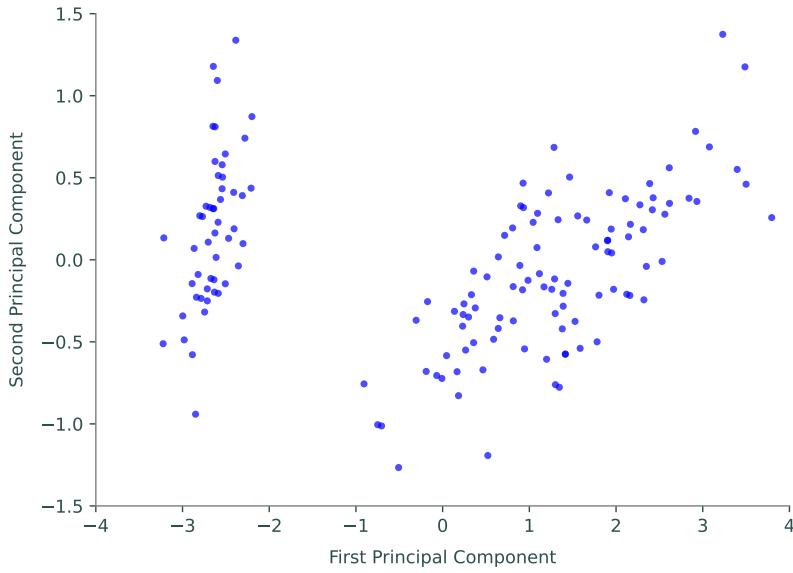


Figure 8.1: The first two principal components of the iris dataset.

## The K-Means Algorithm

Finding the global minimizing partition  $S^*$  is generally intractable since the set of partitions can be very large indeed, but the *k-means* algorithm is a heuristic approach that can often provide reasonably accurate results.

We begin by specifying an initial cluster mean  $\mu_i^{(1)}$  for each  $i = 1, \dots, N$ . This can be done by random initialization, or according to some heuristic. For each iteration, we adopt the following procedure. Given a current set of cluster means  $\mu^{(t)}$ , we find a partition  $S^{(t)}$  of the observations such that

$$S_i^{(t)} = \{x_j : \|x_j - \mu_i^{(t)}\|_2^2 \leq \|x_j - \mu_l^{(t)}\|_2^2, l = 1, \dots, N\}.$$

We then update our cluster means by computing for each  $i = 1, \dots, N$ . We continue to iterate in this manner until the partition ceases to change.

Figure 8.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

The algorithm can be summarized as follows.

1. From the data points, choose  $k$  initial cluster centers.
2. For  $i = 0, \dots, \text{max\_iter}$ :
  - (a) Assign each data point to the cluster center that is closest, forming  $k$  clusters.
  - (b) Recompute the cluster centers as the means of the new clusters.
  - (c) If the old cluster centers and the new cluster centers are sufficiently close, terminate early.

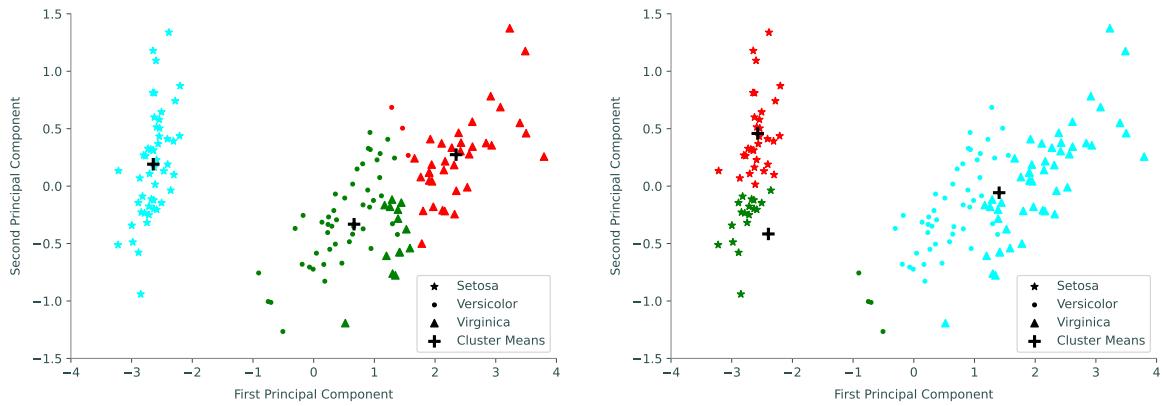


Figure 8.2: Two different *k-means* clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

**Problem 1.** Write a `KMeans` class for doing basic *k-means* clustering. Implement the following methods.

1. `__init__()`: Accept a number of clusters `k`, a maximum number of iterations, and a convergence tolerance. Store these as attributes.
2. `fit()`: Accept an  $m \times n$  matrix `X` of  $m$  data points with  $n$  features. Choose `k` random rows of `X` as the initial cluster centers. Run the *k-means* iteration until consecutive centers are within the convergence tolerance, or until iterating the maximum number of times. Save the cluster centers as attributes.  
If a cluster is empty, reassign the cluster center as a random row of `X`.
3. `predict()`: Accept an  $l \times n$  matrix `X` of data. Return an array of  $l$  integers where the  $i$ th entry indicates which cluster center the  $i$ th row of `X` is closest to.
4. `plot()`: Accept an  $l \times n$  matrix `X` of  $l$  data points and an array `y` of  $l$  integers representing the labels. Plot each data point from the matrix, colored by cluster, as well as the cluster centers. Note that in this case,  $n = 2$ .

Test your class on the iris data set (with `k = 3`) after reducing the data to two principal components. Plot the data and cluster centers, coloring the data by cluster.

Hint: There are various ways to color data points given their cluster labels. A simple way is to create a list of color names, and then set each cluster color by indexing the list using the cluster labels as indices. Another way (more adaptable to a varying number of clusters) is to use `plt.scatter` and set the colors `c` equal to the cluster labels `y` (i.e. `c=y`), and then provide a colormap (i.e. `cmap="jet"`).

## Fire Station Placement

When urban planners are making plans for a city, there are many city elements to consider. One of which is the locations of the fire stations that will service the city. When choosing a suitable location for the city, urban planners look at the current building locations, the roads nearby each location, prior traffic history and the areas of potential growth. We will simplify this complex problem by only taking into account the distances from each building to the nearest fire station (see Additional Material for a harder version of this problem).

Using another data set from SKLearn, we can get the data from the 1990 US Census for California housing based on the blocks of the residents. This has been saved in `sacramento.npy` and can be accessed by using the `np.load()` function. This file contains demographic data for each block in Sacramento and nearby cities. The eight columns in the file are: median block income, median house age in the block, average number of rooms, average number of bedrooms, average house occupancy, latitude and longitude.

There are couple ways for a fire station to be optimally placed. The stations could be placed to minimize the average distance to each house. Another option is to minimize the distance to the farthest house in each group. For this problem, minimize the distance to the farthest house in each group.

**Problem 2.** Using the Methods you wrote in Problem 1, add a parameter `p` to your class that denotes the norm and defaults to 2. Save `p` as an attribute to be used in your `fit()` and `predict()` functions.

Using the latitude and longitude data in `sacramento.npy`, find the optimal placement for 12 fire stations. Plot the longitude and latitude data points colored by cluster as well as the cluster centers. Make plots for 3 different values of `p` to find the optimal locations for the fire stations. In a Markdown cell, report which norm you found to be the best at keeping the maximum distance small.

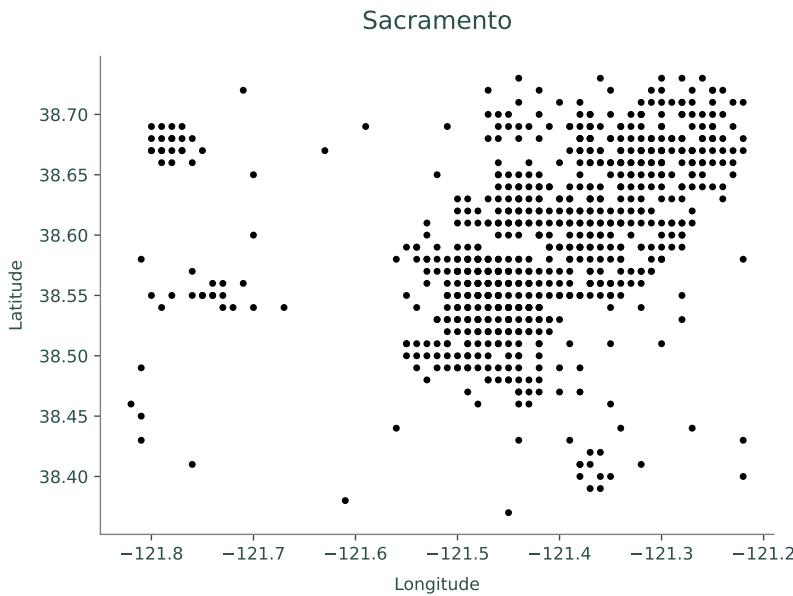


Figure 8.3: Sacramento Housing Data (1990 US Census).

## Detecting Active Earthquake Regions

Suppose we are interested in learning about which regions are prone to experience frequent earthquake activity. We could make a map of all earthquakes over a given period of time and examine it ourselves, but this, as an unsupervised learning problem, can be solved using our *k-means* clustering tool.

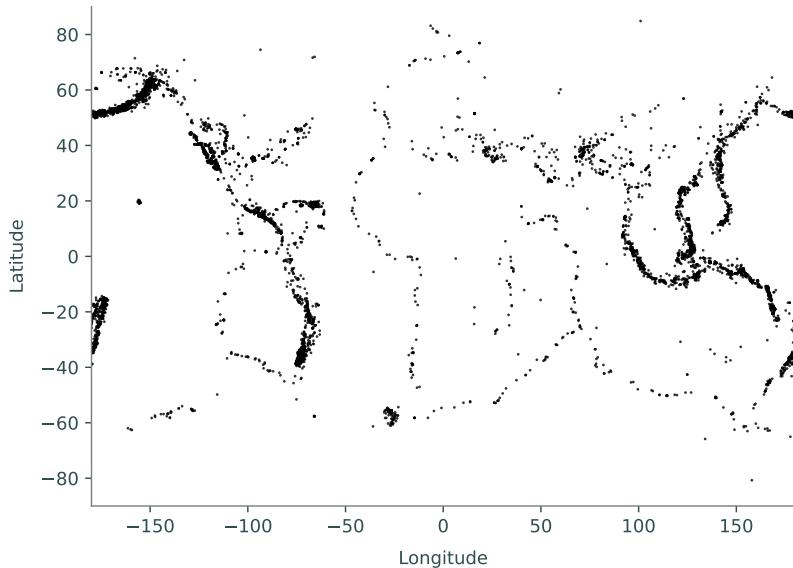


Figure 8.4: Earthquake epicenters over a 6 month period.

The file `earthquake_coordinates.npy` contains earthquake data throughout the world from January 2010 through June 2010. Each row represents a different earthquake; the columns are scaled longitude and latitude measurements. We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in  $\mathbb{R}^2$  with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. Instead, latitude and longitude should be viewed in *spherical coordinates* in  $\mathbb{R}^3$ , which could then be clustered.

A simple way to accomplish this transformation is to first transform the latitude and longitude values to spherical coordinates, and then to Euclidean coordinates. Recall that a spherical coordinate in  $\mathbb{R}^3$  is a triple  $(r, \theta, \varphi)$ , where  $r$  is the distance from the origin,  $\theta$  is the radial angle in the  $xy$ -plane from the  $x$ -axis, and  $\varphi$  is the angle from the  $z$ -axis. In our earthquake data, once the longitude is converted to radians it is an appropriate  $\theta$  value; the latitude needs to be offset by  $90^\circ$  degrees, then converted to radians to obtain  $\varphi$ . For simplicity, we can take  $r = 1$ , since the earth is roughly a sphere. We can then transform to Euclidean coordinates using the following relationships.

$$\theta = \frac{\pi}{180} (\text{longitude}) \quad \varphi = \frac{\pi}{180} (90 - \text{latitude})$$

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} & x &= r \sin \varphi \cos \theta \\ \varphi &= \arccos \frac{z}{r} & y &= r \sin \varphi \sin \theta \\ \theta &= \arctan \frac{y}{x} & z &= r \cos \varphi \end{aligned}$$

There is one last issue to solve before clustering. Each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. Therefore, the cluster centers should also have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth, and the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors.

**Problem 3.** Add a keyword argument `normalize=False` to your `KMeans` constructor. Modify `fit()` so that if `normalize` is `True`, the cluster centers are normalized at each iteration.

Cluster the earthquake data in three dimensions by converting the data from raw data to spherical coordinates to euclidean coordinates on the sphere.

1. Convert longitude and latitude to radians, then to spherical coordinates.  
(Hint: `np.deg2rad()` may be helpful.)
2. Convert the spherical coordinates to euclidean coordinates in  $\mathbb{R}^3$ .
3. Use your `KMeans` class with normalization to cluster the euclidean coordinates with `k = 15`.
4. Translate the cluster center coordinates back to spherical coordinates, then to degrees.  
Transform the cluster means back to latitude and longitude coordinates.  
(Hint: use `np.arctan2()` for arctangent so that the correct quadrant is chosen).
5. Plot the data, coloring by cluster. Also mark the cluster centers.

With 15 clusters, your plot should resemble Figure 8.5.

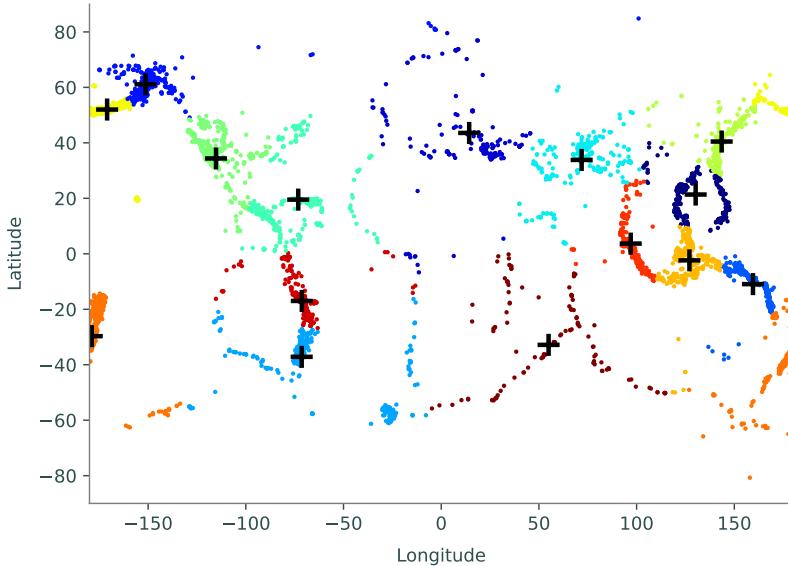


Figure 8.5: Earthquake epicenter clusters with  $k = 15$ .

## Color Quantization

The  $k$ -means algorithm uses the euclidean metric, so it is natural to cluster geographic data. However, clustering can be done in any abstract vector space. The following application is one example.

Images are usually represented on computers as 3-dimensional arrays. Each 2-dimensional layer represents the red, green, and blue color values, so each pixel on the image is really a vector in  $\mathbb{R}^3$ . Clustering the pixels in  $RGB$  space leads a one kind of image segmentation that facilitate memory reduction.

Reading: [https://en.wikipedia.org/wiki/Color\\_quantization](https://en.wikipedia.org/wiki/Color_quantization)

**Problem 4.** Write a function that accepts an image array (of shape  $(m, n, 3)$ ), an integer number of clusters  $k$ , and an integer number of samples  $S$ . Reshape the image so that each row represents a single pixel. Choose  $S$  pixels to train a  $k$ -means model on with  $k$  clusters. Make a copy of the original picture where each pixel has the same color as its cluster center. Return the new image. For this problem, you may use `sklearn.cluster.KMeans` instead of your `KMeans` class from Problem 1.

Test your function on the six provided NASA images.

## Additional Material

### Spectral Clustering

We now turn to another method for solving a clustering problem, namely that of Spectral Clustering. It can cluster data not just by its location on a graph, but can even separate shapes that overlap others into distinct clusters. It does so by utilizing the spectral properties of a Laplacian matrix. Different types of Laplacian matrices can be used. In order to construct a Laplacian matrix, we first need to create a graph of vertices and edges from our data points. This graph can be represented as a symmetric matrix  $W$  where  $w_{ij}$  represents the edge from  $x_i$  to  $x_j$ . In the simplest approach, we can set  $w_{ij} = 1$  if there exists an edge and  $w_{ij} = 0$  otherwise. However, we are interested in the similarity of points, so we will weight the edges by using a *similarity measure*. Points that are similar to one another are assigned a high similarity measure value, and dissimilar points a low value. One possible measure is the *Gaussian similarity function*, which defines the similarity between distinct points  $x_i$  and  $x_j$  as

$$s(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

for some set value  $\sigma$ .

Note that some similarity functions can yield extremely small values for dissimilar points. We have several options for dealing with this possibility. One is simply to set all values which are less than some  $\varepsilon$  to be zero, entirely erasing the edge between these two points. Another option is to keep only the  $T$  largest-valued edges for each vertex. Whichever method we choose to use, we will end up with a weighted *similarity matrix*  $W$ . Using this we can find the diagonal *degree matrix*  $D$ , which gives the number of edges found at each vertex. If we have the original fully-connected graph, then  $D_{ii} = n - 1$  for each  $i$ . If we keep the  $T$  highest-valued edges,  $D_{ii} = T$  for each  $i$ .

As mentioned before, we may use different types of Laplacian matrices. Three such possibilities are:

1. The *unnormalized Laplacian*,  $L = D - W$
2. The *symmetric normalized Laplacian*,  $L_{sym} = I - D^{-1/2}WD^{-1/2}$
3. The *random walk normalized Laplacian*,  $L_{rw} = I - D^{-1}W$ .

Given a similarity measure, which type of Laplacian to use, and the desired number of clusters  $k$ , we can now proceed with the Spectral Clustering algorithm as follows:

- Compute  $W$ ,  $D$ , and the appropriate Laplacian matrix.
- Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of the Laplacian matrix.
- Set  $U = [u_1, \dots, u_k]$ , and if using  $L_{sym}$  or  $L_{rw}$  normalize  $U$  so that each row is a unit vector in the Euclidean norm.
- Perform  $k$ -means clustering on the  $n$  rows of  $U$ .
- The  $n$  labels returned from your `kmeans` function correspond to the label assignments for  $x_1, \dots, x_n$ .

As before, we need to run through our *k-means* function multiple times to find the best measure when we use random initialization. Also, if you normalize the rows of  $U$ , then you will need to set the argument `normalize = True`.

You can use the following function declaration to implement the Spectral Clustering Algorithm by calling your `kmeans` function.

```

def specClus(measure, Laplacian, args, arg1=None, kitters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1, 2, 3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass "data" into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kitters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -----
    labels : ndarray of shape (n,)
        The i-th entry is an integer in [0, n_clusters-1] indicating
        which cluster the i-th row of data belongs to.
    """
    pass

```

We now need a way to test our code. The website <http://cs.joensuu.fi/sipu/datasets/> contains many free data sets that will be of use to us. Scroll down to the “Shape sets” heading, and download some of the datasets found there to use for trial datasets.

You can use the following function declaration to create a function that will return the accuracy of your spectral clustering implementation.

```

def test_specClus(location, measure, Laplacian, args, arg1=None, kitters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    location : string
        The location of the dataset to be tested.
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1, 2, 3}

```

```

Which Laplacian matrix to use. 1 corresponds to the unnormalized,
2 to the symmetric normalized, 3 to the random walk normalized.

args : tuple
    The arguments as they were passed into your k-means function,
    consisting of (data, n_clusters, init, max_iter, normalize). Note
    that you will not pass "data" into your k-means function.

arg1 : None, float, or int
    If Laplacian==1, it should remain as None
    If Laplacian==2, the cut-off value, epsilon.
    If Laplacian==3, the number of edges to retain, T.

kitters : int
    How many times to call your kmeans function to get the best
    measure.

Returns
-----
accuracy : float
    The percent of labels correctly predicted by your spectral
    clustering function with the given arguments (the number
    correctly predicted divided by the total number of points).

"""
pass

```

## Fire Station Placement II

In Problem 2 we looked at choosing the best location for a fire station. However, because we looked at the city of Sacramento where the geography doesn't role in choosing a location, we didn't need to double check that there is a place for the station. The `sanfrancisco.npy` data is organized the same way as `sacramento.py`, as this also comes from the SKLearn California Housing Module. Doing the same method as before will give us groups of houses, however, the group centers may be in the middle of the bay. When implementing this problem, perform a check on the centers to make sure they are not in water. The file `bayboundary.npy` gives a rough outline of where the bay is. The `bayboundary.npy` has only 2 columns, longitude and latitude. Using the boundaries set, make sure that the chosen centers are on land and not on water.

As an additional exercise, import and parse the data from the `bayboundary.npy` and the `sanfrancisco.npy` files. Using either the algorithm that you wrote in Problem 1 or the *k-means* algorithm in the SK Learn library, find the optimal locations for the 12 fire stations.

After the algorithm has finished running, check to see if the new coordinates are on land. Return the graph of the clusters, the centers (the fire station locations) as different colors.

# 9

# Random Forests

**Lab Objective:** *Understand how to build and use a classification tree and a random forest.*

## Classification Trees

Decision Classification trees are a class of decision trees used in a wide variety of settings where labeled training data is available. The desired outcome is a model that can accurately assign labels to unlabeled data. Decision trees are widely used because they have a fast run time, low computation cost, and can handle irrelevant, missing, and noisy data easily.

We begin with a dataset of samples, such as information about customers from a certain store. Each sample contains a variety of features, such as if the individual is married or has children. The sample also has a classification label, such as whether or not the person made a specific purchase.

A classification tree is composed of many *nodes*, which ask a question (i.e. “Is income  $\geq 85$ ?”) and then split the data based on the answers. If the response is **True**, then the sample is “pushed” down the tree to the left child node. If the response is **False**, then the sample is “pushed” down the tree to the right child node. A *leaf* node is a node that has no child node. Upon arrival at a leaf, an unlabeled sample is labeled with the classification that matches the majority of labeled samples at that leaf.

Table 9.1 includes information about 10 individuals and an indicator of whether or not they made a certain purchase. To simplify construction of the tree, all data is numeric, so 1=Yes and 0=No for yes/no questions.

Suppose we wanted to guess whether a single college student making under \$30,000 would purchase this item. Starting at the top of the tree, we compare our sample to the question and first choose the right branch, and then we compare with the second question and choose the right branch again. Now we reach a leaf with the dictionary `{0:1}`. The key 0 corresponds to the label, and the value 1 means one of our original samples is at this leaf with that label. Since 100% of samples at this leaf are labeled with 0, our new sample college student will be predicted to share the label 0.

If we arrived instead at a leaf with the dictionary `{0:1, 1:4}`, then one of our original samples at this leaf would be labeled 0 and four would be labeled 1, so the majority vote would assign the label 1 to our new sample.

Married (Y/N)	Children	Income (\$1000)	Purchased (Y/N)
0	5	125	0
1	0	100	0
0	0	70	0
1	3	120	0
0	0	95	1
1	0	60	0
0	2	220	1
0	0	85	1
1	0	75	0
0	0	90	1

Table 9.1: Customer data with 3 features (Married, Children, Income) and a label (Purchase) indicating whether or not the customer bought the item.

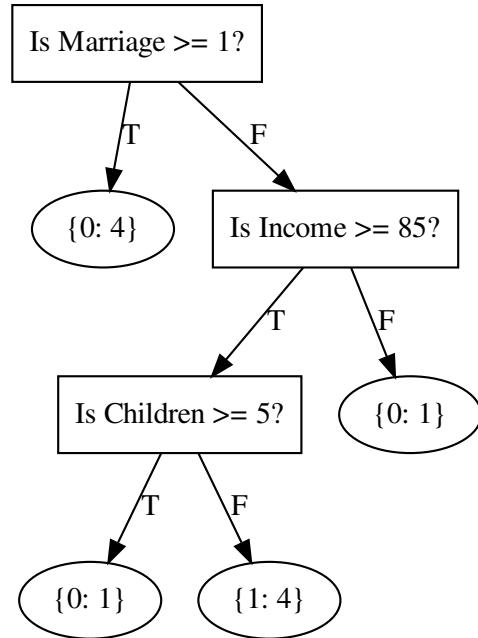


Figure 9.1: A classification tree built using Table 9.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

**Problem 1.** At each node in a classification tree, a question determines which branch a sample belongs to. The `Question` class has attributes `column` and `value`. Write a `match` method for the `Question` class that accepts a sample and returns either `True` or `False`. A sample will be in the form of an array, so in the example above, a single college student with no children making \$20,000 would be represented by the array `[0, 0, 20]`. The method should determine if the sample's feature located at index `column` is greater than or equal to `value`. Notice that this method will only handle one feature of one sample at a time.

Next, write a `partition()` function that partitions the samples (rows) of a dataset for a given `Question` into two `numpy` arrays, `left` and `right`, returned in that order. The array `left` will contain the samples that the `match` method returned as `True`, and the array `right` will contain the samples that the `match` method returned as `False`. If `left` or `right` is empty, still return them as (2-D size zero) arrays.

Hint: if `n` is the length of each sample, `left.reshape(-1, n)` (and similar for `right`) will make the final arrays the correct size even if they're empty.

The file `animals.csv` contains information about 7 features for 100 animals. The last column, the class labels, indicates whether or not an animal lives in the ocean. You may use this file to test your functions.

```
>>> import numpy as np
# Load in the data
>>> animals = np.loadtxt("animals.csv", delimiter=',')
# Load in feature names
>>> features = np.loadtxt("animal_features.csv", delimiter=',', dtype=str,
...                         comments=None)
# Load in sample names
>>> names = np.loadtxt('animal_names.csv', delimiter=',', dtype=str)

# initialize question and test partition function
>>> question = Question(column=1, value=3, feature_names=features)
>>> left, right = partition(animals, question)
>>> print(len(left), len(right))
62 38

>>> question = Question(column=1, value=75, feature_names=features)
>>> left, right = partition(animals, question)
>>> print(len(left), len(right))
0 100
```

## Optimal Split

To use the `partition()` function from Problem 1, we need to know which question to ask at each node. Usually, the question is determined by the split that maximizes either the Gini impurity or the information gain (which itself uses the Gini impurity). For this lab, we will use the information gain.

Gini impurity measures how often a sample would be mislabeled based on the distribution of labels. It is a measure of homogeneity of labels, so it is 0 when all samples at a node have the same label.

**Definition 9.1.** Let  $D$  be a dataset with  $K$  different class labels and  $N$  different samples. Let  $N_k$  be the number of samples labeled class  $k$  for each  $1 \leq k \leq K$ . We define the Gini impurity to be

$$G(D) = 1 - \sum_{k=1}^K \left( \frac{N_k}{N} \right)^2.$$

Information gain is based on the concept of information theory entropy. It measures the difference between two probability distributions. If the distributions are equal, then the information gain is 0. We will use a modified version of information gain for simplicity:

**Definition 9.2.** Let  $s_D(p, x) = D_1, D_2$  be a partition of data  $D$ . We define the information gain of this partition to be

$$I(s_D(p, x)) = G(D) - \sum_{i=1}^2 \frac{|D_i|}{|D|} \cdot G(D_i)$$

where  $|D|$  represents the number of samples (or rows) in  $D$ .

The provided function `info_gain()` can be used to compute the information gain of a partition of a dataset. The optimal split of data at a node can be chosen by finding the question whose partition maximizes the information gain.

Sometimes the partition to split on may separate the data into very small subsets with only a few samples each. This can make the classification tree vulnerable to overfitting and noisy data. For this reason, we will include an argument to specify the smallest allowable leaf size, or the minimum number of samples at any node. A reasonable minimum number depends on the size of the whole dataset, so a dataset with 10,000 samples would have a larger minimum leaf size than our first example with only 10 samples.

To find the optimal split, begin by instantiating `best_gain` to 0 and `best_question` to `None`. For each unique value in each feature (column) of the dataset, instantiate a `Question` object with the column and value, then use `partition()` to split the dataset into left and right partitions. If either the left or right partition has fewer samples than the smallest allowable leaf size (called `min_samples_leaf`), then discard this split and iterate to the next one. If the left and right partitions are ok, then calculate the `info_gain()` of these two partitions. If this `info_gain()` is greater than `best_gain`, then set `best_gain` and `best_question` equal to `info_gain()` and the corresponding `Question`, respectively. After checking all possible partitions (every column and row), return `best_gain` and `best_question`.

**Problem 2.** Write a function `find_best_split()` that computes the optimal split of a dataset by following the directions given above. Recall that the final column of the dataset contains the class labels, which has no questions associated with it, so do not iterate through the final column. Include a minimum leaf size argument `min_samples_leaf` defaulting to 5.

Return the information gain and corresponding question for the best split, in that order. If two splits have the same information gain, choose the first split. Iterate through each feature (column) of the dataset, then through each unique value (row) that the feature takes on; that is, iterate through columns in the outer loop and rows in the inner loop. If no partitions are possible due to `2*min_samples_leaf`, return `None` for the question.

You should get the following output for the animals dataset.

```
# Test your function
>>> find_best_split(animals, features)
(0.12259833679833687, Is # legs/tentacles >= 2.0?)
```

## Building the Tree

Once the optimal split for a node is determined, the node needs to be defined as either a Leaf node or a Decision node. As described earlier, leaf nodes have no children, and the classification of samples are determined in leaf nodes. If the optimal split returns a left and right tree, then the node is a decision node and has a question associated with it to determine which path a sample should follow. The next two problems will walk through building a classification tree using the functions and classes from the previous problems.

**Problem 3.** The class `Leaf` is instantiated with data containing samples in that leaf. In the constructor, save an attribute `prediction` as a dictionary of how many samples in the leaf belong to each unique class label.

Hint: remember the provided function `class_counts()`.

Write the class `Decision_Node`. This class should have three attributes: an associated `Question`, a left branch, and a right branch. The branches will be `Leaf` or `Decision_Node` objects. Name these three attributes `question`, `left`, and `right`.

In addition to having a minimum leaf size, it's also important to have a maximum depth for trees. Without restricting the depth, the tree can become very large; if there is no minimum leaf size, it can be one less than the number of training samples. Limiting the depth can stop the tree from having too many splits, preventing it from becoming too complex and overfitting the training data. On the other hand, it's also important to not have too shallow of a tree because then the tree will underfit the data.

**Problem 4.** Write a function `build_tree()` that uses your previous functions to build a classification tree. Include a minimum leaf argument defaulting to 5 and a maximum depth argument defaulting to 4. Start counting depth at 0. For comparison, the tree in Figure 9.1 has depth 3. We will build this tree recursively as follows:

- If the number of samples (rows) in the given data is less than twice `min_samples_leaf` (i.e., it can't be split again), then return the data as a `Leaf` and that's it.

- Otherwise, the data can be split, so find the optimal gain and corresponding question using the function `find_best_split()`.
- If the optimal gain is 0 (i.e., if the partition is already optimal), or if `current_depth` is greater than or equal to `max_depth` (i.e., the tree is already too deep), return the data as a `Leaf` and that's it.
- If the node isn't a `Leaf`, then it must be a `Decision_Node`.
  - Use `partition()` to split the data into left and right partitions.
  - Next, recursively define the right branch and left branch of the tree by calling `build_tree()` on each of the left and right partitions with `current_depth` incremented by 1.
  - Finally, return a `Decision_Node` object using the optimal question found earlier and the left and right branches of the tree.

The function `draw_tree()` is provided to allow you to save a pdf image to view a specified trained tree. In order to use this function, you must successfully install the `graphviz` package.

`graphviz` has two parts: an external program, and a Python package that interfaces with that program. To install both of these parts, refer to the section in Additional Materials. (Both of these are installed by `install_dependencies.sh`.)

With `graphviz` installed, you can test your `build_tree` function as follows:

```
>>> my_tree = build_tree(animals, features)
>>> draw_tree(my_tree)
```

The resulting tree should have 8 question nodes, 9 leaf nodes, and a total of 5 rows of nodes (including the lowermost leaves). If `draw_tree` returns an error about pdf being an unrecognized file type, try running the command `dot -c` in your terminal.

## Predicting

It's important to test your tree to ensure that it predicts class labels fairly accurately and so that you can adjust the minimum leaf and maximum depth parameters as needed. It is customary to randomly assign some of your labeled data to a training set that you use to fit your tree and then use the rest of your data as a testing set to check accuracy.

**Problem 5.** Write a function `predict_tree()` that returns the predicted class label for a single new sample given a trained tree called `my_tree`. This function will be implemented recursively in order to traverse the branches and reach a `Leaf` node. Use the `isinstance()` function to determine if the current node (`my_tree`) is of type `Leaf`; if it is, return the label that corresponds with the most samples in the `Leaf`.

If the given tree is not a `Leaf`, then it is a `Decision_Node` with left and right children nodes. If the `my_tree.question.match` method is `True` with the given sample, then recursively call `predict_tree()` with `my_tree.left`. Otherwise, recursively call `predict_tree()` with `my_tree.right`.

Hint: an easy way to get the most common value at a Leaf node is to call `max()` on `my_tree.prediction` using the keyword argument `key=my_tree.prediction.get`. Remember to return something from both branches of your function.

Next, write a function `analyze_tree()` that accepts a labeled dataset (with the labels in the last column, as in `animals.csv`) and a trained classification tree. The function should compare the actual label of each sample (row) with its predicted label using `predict_tree()`, and it should return the proportion of samples that the tree labels correctly.

Test your function with the `animals.csv` file. Shuffle the dataset with `np.random.shuffle()` and use 80 samples to train your classification tree. Use the other 20 samples as the test set to see how accurately your tree classifies them. Your tree should be able to classify this set with roughly 80% accuracy on average, given the default parameters.

## Random Forest

As noted, one of the main issues with Decision Trees is their tendency to overfit. Random forests are a way of mitigating overfitting that cannot be fixed by restricting the tree depth and leaf size. A *random forest* is just what it sounds like—a collection of trees. Each tree is trained randomly, meaning that at each node, only a small, randomly-chosen subset of the features is available to determine the next split. The size of this subset should be small relative to the total number of features present. Let  $n$  be the total number of features in the dataset. A common method which we will use here is to split on  $\sqrt{n}$  features (rounding down where applicable).

When predicting the label of a new sample, each trained tree in the forest casts a vote, determined as above, and the sample is labeled according to the majority vote of the trees.

**Problem 6.** Add an argument `random_subset` to `find_best_split()` and `build_tree()`, defaulting to `False`, that indicates whether or not the tree should be trained randomly. When `True`, each node should be restricted to a random combination of  $\sqrt{n}$  (rounded down) features to use in its split, where  $n$  is the total number of features (note that class labels are not features). This will require the function `find_best_split()` to be altered so that it only iterates through a random combination of  $\sqrt{n}$  features (columns).

Next, write a function `predict_forest()` that accepts a new sample and a trained forest (as a list of trees). It should iterate through each tree, finding the label assigned to the sample by calling `predict_tree()`. Then, it should return the label predicted by the majority of the trees.

Finally, write a function `analyze_forest()` that accepts a labeled dataset (with the labels in the last column, as in `animals.csv`) and a trained forest. The function should compare the actual label of each sample (row) with its predicted label using `predict_forest()`, and it should return the accuracy of the forest's predictions.

Test your functions on the `animals.csv` dataset. Visualize your trees using `draw_tree()`, verifying that they are different every time. Compare the results to the non-randomized version.

## Scikit-Learn

Finally, we'll compare our implementation to scikit-learn's `RandomForestClassifier`. Rather than accepting all the data as a single array, as in our implementation, this package accepts the feature data as the first argument and all of the labels as the second argument.

```
>>> from sklearn.ensemble import RandomForestClassifier

# Create the forest with the appropriate arguments and 200 trees
>>> forest = RandomForestClassifier(n_estimators=200, max_depth=4,
...                                 min_samples_leaf=5)

# Shuffle the data
>>> shuffled = np.random.permutation(animals)
>>> train = shuffled[:80]
>>> test = shuffled[80:]

# Fit the model to your data, passing the labels in as the second argument
>>> forest.fit(train[:, :-1], train[:, -1])

# Test the accuracy with the testing set
>>> forest.score(test[:, :-1], test[:, -1])
0.85
```

**Problem 7.** The file `parkinsons.csv` contains annotated speech data from people with and without Parkinson's Disease. The first column is the subject ID, columns 2-27 are various features, and the last column is the label indicating whether or not the subject has Parkinson's. You will need to remove the first column so your forest doesn't use participant ID to predict class labels. Feature names are contained in the file `parkinsons_features.csv`.

Write a function to compare your forest implementation to the package from scikit-learn. Because of the size of this dataset, we will only use a small portion of the samples and build a very simple forest. Randomly select 130 samples. Use 100 in training your forest and 30 more in testing it. Build 5 trees for your forest (as a list) using `min_samples_leaf=15` and `max_depth=4` for each tree. Time how long it takes to train and analyze your forest.

Repeat this with scikit-learn's package, using the same 100 training samples and 30 test samples. Set `n_estimators=5`, `min_samples_leaf=15`, and `max_depth=4`.

Then, using scikit-learn's package, run the whole `parkinsons.csv` dataset, using the default parameters. Use 80% of the data to train the forest and the other 20% to test it.

Return three tuples: the accuracy and time of your implementation, the accuracy and time of scikit-learn's package, and then the accuracy and time of scikit-learn's package using the entire dataset.

## Additional Materials

### Installing Graphviz

The Python package can be installed using `pip install graphviz`. To install the external program, on Windows in the WSL terminal and on Linux use the following:

```
sudo apt-get install graphviz
```

On Mac:

```
brew install graphviz
```

For additional options for the external program, refer to the instructions at <https://graphviz.org/download/>.



# 10

## Data Cleaning

**Lab Objective:** *The quality of a data analysis or model is limited by the quality of the data used. In this lab we learn techniques for cleaning data, creating features, and determining feature importance.*

Data cleaning is the process of identifying and correcting bad data. This could be data that is missing, duplicated, irrelevant, inconsistent, incorrect, in the wrong format, or otherwise does not make sense. Though it can be tedious, data cleaning is the most important step of data analysis. Without accurate and legitimate data, any results or conclusions are suspect and may be incorrect.

We will demonstrate common issues with data and how to correct them using the following dataset. It consists of family members and some basic details.

```
# Example dataset
>>> df = pd.read_csv('toy_dataset.csv')

>>> df
      Name  Age      name        DOB Marital_Status
0    John Doe   30    john  01/01/2010     Divorcee
1    Jane Doe   29    jane  12/02/1990     Divorced
2  Jill smith   40     NaN  03/04/1980    married
3  Jill smith   40    jill  03/04/1980    married
4  jack smith  100    jack   4/4/1980  married
5 Jenny Smith    5     NaN  05/05/2015      NaN
6 JAMES Smith    2     NaN  20/06/2018   single
7       Rover    2     NaN  05/05/2018      NaN

      Height  Weight  Marriage_Len      Spouse
0     72.0     175          5      NaN
1      5.5     125          5  John Doe
2     64.0     120         10  Jack Smith
3     64.0     120         NaN  jack smith
4      1.8     220         10  jill smith
5    105.0      40         NaN      NaN
```

6	27.0	25	Not Applicable	NaN
7	36.0	50	NaN	NaN

## Data Type

We can check the data type in Pandas using `dtype`. A `dtype` of `object` means that the data in that column contains either strings or mixed `dtypes`. These fields should be investigated to determine if they contain mixed datatypes. In our toy example, we would expect that `Marriage_Len` is numerical, so an object `dtype` is suspicious. Looking at the data, we see that James has Not Applicable, which is a string.

```
# Check validity of data
# Check Data Types
>>> df.dtypes
Name          object
Age           int64
name          object
DOB           object
Marital_Status object
Height        float64
Weight         int64
Marriage_Len  object
Spouse         object
dtype: object
```

## Duplicates

Duplicates can be easily identified in Pandas using the `duplicated()` function. When no parameters are passed, it returns a DataFrame of the first duplicates. We can identify rows that are duplicated in only some columns by passing in the column names. The `keep` parameter has three possible values, first, last, and False. False keeps all duplicated values, while first and last keep only the first and last instances, respectively.

```
# Display duplicated rows
>>> df[df.duplicated()]
Empty DataFrame
Columns: [Name, Age, name, DOB, Marital_Status, Height, Weight, Marriage_Len, ←
           Spouse]
Index: []

# Display rows that have duplicates in some columns
>>> df[df.duplicated(['Name', 'DOB', 'Marital_Status'], keep=False)]
      Name  Age   name      DOB  Marital_Status  Height  Weight  ←
      Marriage_Len    Spouse
2  Jill  smith    40  NaN  03/04/1980     married    64.0     120  ←
10 Jack  Smith
```

3 Jill smith 40 jill 03/04/1980	married	64.0	120	←
NaN jack smith				

## Range

We can check the range of values in a numeric column using the `min` and `max` attributes. If a column corresponds to the temperature in Salt Lake City, measured in degrees Farenheit, then a value over 110 or below 0 should make you suspicious, since those would be extreme values for Salt Lake City. In fact, checking the all-time temperature records for Salt Lake shows that the values in this column should never be more than 107 and never less than  $-30$ . Any values outside that range are almost certainly errors and should probably be reset to `NaN`, unless you have special information that allows you to impute more accurate values.

Other options for looking at the values include line plots, histograms, and boxplots. Some other useful Pandas commands for evaluating the breadth of a dataset include `df.unique()` (which returns a series giving the name of each column and the number of unique values in each column), `pd.unique()` (which returns an array of the unique values in a series), and `value_counts()` (which counts the number of instances of each unique value in a column, like a histogram).

```
# Count the number of unique values in each column
>>> df.unique()
Name          7
Age           6
name          4
DOB           7
Marital_Status 5
Height         7
Weight         7
Marriage_Len   3
Spouse         4
dtype: int64

# Print the unique Marital_Status values
>>> pd.unique(df['Marital_Status'])
array(['Divorcee', 'Divorced', 'married', 'married', nan, 'single'],
      dtype=object)

# Count the number of each Marital_Status values
>>> df['Marital_Status'].value_counts()
Marital_Status
married    2
Divorcee   1
Divorced   1
married   1
single    1
Name: count, dtype: int64
```

## Missing Data

The percentage of missing data is the completeness of the data. All uncleaned data will have missing values, but datasets with large amounts of missing data, or lots of missing data in key columns, are not going to be as useful. Pandas has several functions to help identify and count missing values. In Pandas, all missing data is considered a `NaN` and does not affect the `dtype` of a column. `df.isna()` returns a boolean DataFrame indicating whether each value is missing. `df.notnull()` returns a boolean DataFrame with `True` where a value is not missing. Information on how to deal with missing data is described in later sections.

```
# Count number of missing data points in each column
>>> df.isna().sum()
Name          0
Age           0
name          4
DOB           0
Marital_Status 2
Height         0
Weight         0
Marriage_Len   3
Spouse         4
dtype: int64
```

## Consistency

Consistency measures how cohesive the data is, both within the dataset and across multiple datasets. For example, in our toy dataset **Jack Smith** is 100 years old, but his birth year is 1980. Data is inconsistent across datasets when the data points should be the same and are different. This could be due to incorrect entries or syntax errors.

It is also important to be consistent in units of measure. Looking at the `Height` column in our dataset, we see values ranging from 1.8 to 105. This is likely the result of different units of measure. It is also important to be consistent across multiple datasets.

All features should also have a consistent type and standard formatting (like capitalization). Syntax errors should be fixed, and white space at the beginning and ends of strings should be removed. Some data might need to be padded so that it's all the same length.

Method	Description
<code>series.str.lower()</code>	Convert to all lower case
<code>series.str.upper()</code>	Convert to all upper case
<code>series.str.strip()</code>	Remove all leading and trailing white space
<code>series.str.lstrip()</code>	Remove leading white space
<code>series.str.replace(" ","")</code>	Remove all spaces
<code>series.str.pad()</code>	Pad strings

Table 10.1: Pandas String Formatting Methods

**Problem 1.** The `g_t_results.csv` file is a set of parent-reported scores on their child's Gifted and Talented tests. The two tests, OLSAT and NNAT, are used by NYC to determine if children are qualified for gifted programs. The OLSAT Verbal has 16 questions for Kindergarteners and 30 questions for first, second, and third graders. The NNAT has 48 questions. Each test assigns 1 point to each question asked (so there are no non integer scores). Using this dataset, answer the following questions.

1. What column has the highest number of null values and what percent of its values are null? Print the answer as a tuple with (column name, percentage). Make sure the second value is a percent.
2. List the columns that should be numeric that aren't. Print the answer as a tuple.
3. How many third graders have scores outside the valid range for the OLSAT Verbal Score? Print the answer
4. How many data values are missing (NaN)? Print the number.

## Cleaning

There are many aspects and methods of cleaning; here are a few ways of doing so.

### Unwanted Data

Removing unwanted data typically falls into two categories, duplicated data and irrelevant data. Irrelevant data consists of observations that don't fit the specific problem you are trying to solve or don't have enough variation to affect the model. We can drop duplicated data using the `duplicated()` function described above with `drop()` or `drop_duplicates()`.

### Missing Data

Some commonly suggested methods for handling data are removing the missing data and setting the missing values to some value based on other observations. However, missing data can be informative and removing or replacing missing data erases that information. Removing missing values from a dataset might result in losing significant amounts of data or even in a less accurate model. Retaining the missing values can help increase accuracy.

We have several options to deal with missing data:

- Dropping missing data is the easiest method. Dropping rows or columns should only be done if there is less than 90% – 95% of the data available. If dropping missing data is inappropriate, you may instead choose to estimate the missing values which can be done by solving the mean, mode, median, randomly choosing from a distribution, linear regression, or hot-decking, to name a few.
- Hot-decking is when you fill in the data based on similar observations. It can be applied to numerical and categorical data, unlike many of the other options listed above. Sequential hot-decking sorts the column with missing data based on an auxiliary column and then fills in the data with the value from the next available data point. K-Nearest Neighbors can also be used to identify similar data points.

- The last option is to flag the data as missing. This retains the information from missing data and removes the missing data (by replacing it). For categorical data, simply replace the data with a new category. For numerical data, we can fill the missing data with 0, or some value that makes sense, and add an indicator variable for missing data.

```

## Replace missing data
import numpy as np

# Add an indicator column based on missing Marriage_Len
>>> df['missing_ML'] = df['Marriage_Len'].isna()

# Fill in all missing data with 0
>>> df['Marriage_Len'] = df['Marriage_Len'].fillna(0)

# Change all other NaNs to missing
>>> df = df.fillna('missing')

# Change Not Applicable row to NaNs
>>> df = df.replace('Not Applicable', np.nan)

# Drop rows with NaNs
>>> df = df.dropna()

>>> df
      Name  Age     name        DOB Marital_Status
0    John Doe  30    john  01/01/2010      Divorcee
1    Jane Doe  29    jane  12/02/1990      Divorced
2  Jill smith  40  missing  03/04/1980       married
3  Jill smith  40     jill  03/04/1980       married
4  jack smith 100     jack   4/4/1980      marrieed
5 Jenny Smith   5  missing  05/05/2015       missing
7      Rover   2  missing  05/05/2018       missing

      Height  Weight  Marriage_Len     Spouse missing_ML
0     72.0     175          5  missing      False
1      5.5     125          5  John Doe      False
2     64.0     120         10  Jack Smith      False
3     64.0     120          0  jack smith     True
4      1.8     220         10  jill smith      False
5    105.0      40          0  missing      True
7     36.0      50          0  missing      True

```

## Nonnumerical Values Misencoded as Numbers

Missing data should always be stored in a form that cannot accidentally be incorporated into the model. This is typically done by storing missing values as NaN. Some algorithms will not run on data with NaN values, in which case you may choose to fill missing data with a string '`'missing'`'. Many datasets have recorded missing values with a 0 or some other number. You should verify that this does not occur in your dataset.

Categorical data are also often encoded as numerical values. These values should not be left as numbers that can be computed with. For example, postal codes are shorthand for locations, and there is no numerical meaning to the code. It makes no sense to add, subtract, or multiply postal codes, so it is important to not do so. It is good practice to convert this kind of non-numeric data into strings or other data types that cannot be computed with.

## Ordinal Data

Ordinal data is data that has a meaningful order but the differences between the values aren't consistent or meaningful. For example, a survey question might ask about your level of education, with 1 being high-school graduate, 2 bachelor's degree, 3 master's degree, and 4 doctoral degree. These values are called ordinal data because it is meaningful to talk about an answer of 1 being less than an answer of 2, but the difference between 1 and 2 is not necessarily the same as the difference between 3 and 4. Subtracting, taking the average, and other algebraic methods do not make a lot of sense with ordinal data.

**Problem 2.** `imdb.csv` contains a small set of information about 99 movies. Valid movies for this dataset should be longer than 30 minutes long, should have a positive `imdb_score`, and have a `title_year` after 2000.

Clean the data set by doing the following in order:

1. Remove duplicate rows by dropping the first or last. Print the shape of the dataframe after removing the rows.
2. Drop all rows that contain missing data. Print the shape of the dataframe after removing the rows.
3. Remove rows that have data outside of the valid data ranges described above and explain briefly how you determined your ranges for each column.
4. Identify and drop columns with three or fewer different values. Print a tuple with the names of the columns dropped.
5. Convert the titles to all lower case.

Print the first five rows of your dataframe.

## Feature Engineering

Constructing new features is called *feature engineering*. Once new features are created, we can analyze how much a model depends on each feature. Features with low importance probably do not contribute much and could potentially be removed.

Discrete Fourier transforms and wavelet decomposition often reveal important properties of data collected over time (*called time-series*), like sound, video, economic indicators, etc. In many such settings it is useful to engineer new features from a wavelet decomposition, the DFT, or some other function of the data.

### Engineering for Categorical Variables

Categorical features are those that take only a finite number of values, and usually no categorical value has a numerical meaning, even if it happens to be number. For example in an election dataset, the names of the candidates in the race are categorical, and there is no numerical meaning (neither ordering nor size) to numbers assigned to candidates based solely on their names.

Consider the following election data.

Ballot number	For Governor	For President
001	Herbert	Romney
002	Cooke	Romney
003	Cooke	Obama
004	Herbert	Romney
005	Herbert	Romney
006	Cooke	Stein

A common mistake occurs when someone assigns a number to each categorical entry (say 1 for Cooke, 2 for Herbert, 3 for Romney, etc.). While this assignment is not, in itself, inherently incorrect, it is incorrect to use the value of this number in a statistical model. Whenever you encounter categorical data that is encoded numerically like this, immediately change it to non-numerical form (“Cooke,” “Herbert,” “Romney,” . . . ) or apply *one-hot encoding* or *dummy variable encoding*.<sup>1</sup> To do this construct a new feature for every possible value of the categorical variable, and assign the value 1 to that feature if the variable takes that value and zero otherwise. Pandas makes one-hot encoding simple:

```
# one-hot encoding
df = pd.get_dummies(df, columns=['For President'])
```

The previous dataset, when the presidential race is one-hot encoded, becomes

Ballot number	Governor	Romney	Obama	Stein
001	Herbert	1	0	0
002	Cooke	1	0	0
003	Cooke	0	1	0
004	Herbert	1	0	0
005	Herbert	1	0	0
006	Cooke	0	0	1

<sup>1</sup>Yes, these are silly names, but they are the most common names for it. Unfortunately, it is probably too late to change these now.

Note that the sum of the terms of the one-hot encoding in each row is 1, corresponding to the fact that every ballot had exactly one presidential candidate.

When the gubernatorial race is also one-hot encoded, this becomes

Ballot number	Cooke	Herbert	Romney	Obama	Stein
001	0	1	1	0	0
002	1	0	1	0	0
003	1	0	0	1	0
004	0	1	1	0	0
005	0	1	1	0	0
006	1	0	0	0	1

Now the sum of the terms of the one-hot encodings in each row is 2, corresponding to the fact that every ballot had two names—one gubernatorial candidate and one presidential candidate.

Summing the columns of the one-hot-encoded data gives the total number of votes for the candidate of that column. So the numerical values in the one-hot encodings are actually numerically meaningful, and summing the entries gives meaningful information. One-hot encoding also avoids the pitfalls of incorrectly using numerical proxies for categorical data.

The main disadvantage of one-hot encoding is that it is an inefficient representation of the data. If there are  $C$  categories and  $n$  datapoints, a one-hot encoding takes an  $n \times 1$ -dimensional feature and turns it into an  $n \times C$  sparse matrix. But there are ways to store these data efficiently and still maintain the benefits of the one-hot encoding.

### ACHTUNG!

When performing linear regression, it is good practice to add a constant column to your dataset and to remove one column of the one-hot encoding of each categorical variable. This is done because the constant column is a linear combination of the one-hot encoded columns causing the matrix to fail to be invertible and can cause identifiability problems.

The standard way to deal with this is to remove one column of the one-hot embedding for each categorical variable. For example, with the elections dataset above, we could remove the Cooke (governor variable) and Romney (presidential variable) columns. Doing that means that in the new dataset a row sum of 0 corresponds to a ballot with a vote for Cooke and a vote for Romney, while a 1 in any column indicates how the ballot differed from the base choice of Cooke and Romney.

When using pandas, you can drop the first column of a one-hot encoding by passing in `drop_first=True`.

**Problem 3.** Load `housing.csv` into a dataframe with `index_col=0`. Descriptions of the features are in `housing_data_description.txt` for your convenience. The goal is to construct a regression model that predicts `SalePrice` using the other features of the dataset. Do this as follows:

- Identify and handle the missing data. Hint: Dropping every row with some missing data is not a good choice because it gives you an empty dataframe. What can you do instead?

2. Create two new features:
  - (a) **Remodeled**: Whether or not a house has been remodeled with a Y if it has been remodeled, or a N if it has not.
  - (b) **TotalPorch**: Using the 5 different porch/deck columns, create a new column that provides the total square footage of all the decks and porches for each house.
3. Identify the variable with nonnumerical values that are misencoded as numbers. One-hot encode it. (Hint: don't forget to remove one of the encoded columns to prevent collinearity with the constant column).
4. Add a constant column to the dataframe.
5. Save a copy of the dataframe.
6. Choose four categorical features that seem very important in predicting SalePrice. One-hot encode these features, and remove all other categorical features.
7. Run an OLS (Ordinary Least Squares) regression on your model.

Print a list of the ten features that have the highest coefficients in your model. Print the summary for the dataset as well.

To run an OLS model in python, use the following code.

```
import statsmodels.api as sm

# In our case, y is SalesPrice, and X is the rest of the dataset.
>>> results = sm.OLS(y, X).fit()

# Print the summary
>>> results.summary()

# Convert the summary table to a dataframe
>>> results_as_html = results.summary().tables[1].as_html()
>>> result_df = pd.read_html(results_as_html, header=0, index_col=0)[0]
```

**Problem 4.** Using the copy of the dataframe you created in Problem 3, one-hot encode all the categorical variables. Print the shape of your database, and Run OLS.

Print the ten features that have the highest coefficient in your model and the summary. Write a couple of sentences discussing which model is better and why.

# 11

# Introduction to Parallel Computing

**Lab Objective:** *Many modern problems involve so many computations that running them on a single processor is impractical or even impossible. There has been a consistent push in the past few decades to solve such problems with parallel computing, meaning computations are distributed to multiple processors. In this lab, we explore the basic principles of parallel computing by introducing the cluster setup, standard parallel commands, and code designs that fully utilize available resources.*

## Parallel Architectures

Imagine that you are in charge of constructing a very large building. You could, in theory, do all of the work yourself, but that would take so long that it simply would be impractical. Instead, you hire workers, who collectively can work on many parts of the building at once. Managing who does what task takes some effort, but the overall effect is that the building will be constructed many times faster than if only one person was working on it. This is the essential idea behind parallel computing.

A *serial* program is executed one line at a time in a single process. This is analogous to a single person creating a building. Since modern computers have multiple processor cores, serial programs only use a fraction of the computer's available resources. This is beneficial for smooth multitasking on a personal computer because multiple programs can run at once without interrupting each other.

For smaller computations, running serially is fine. However, some tasks are large enough that running serially could take days, months, or in some cases years. In these cases it is beneficial to devote all of a computer's resources (or the resources of many computers) to a single program by running it in *parallel*. Each processor can run part of the program on some of the inputs, and the results can be combined together afterwards. In theory, using  $N$  processors at once can allow the computation to run  $N$  times faster. Even though communication and coordination overhead prevents the improvement from being quite that good, the difference is still substantial.

A *computer cluster* or *supercomputer* is essentially a group of regular computers that share their processors and memory. There are several common architectures that are used for parallel computing, and each architecture has a different protocol for sharing memory, processors, and tasks between *computing nodes*, the different simultaneous processing areas. Each architecture offers unique advantages and disadvantages, but the general commands used with each are very similar.

In this lab, we will explore the usage and capabilities of parallel computing using Python's iPyParallel package. iPyParallel can be installed using pip:

```
$ pip install ipyparallel==8.6.1
```

Use version 8.6.1.

## The iPyParallel Architecture

There are three main parts of the iPyParallel architecture:

- *Client*: The main program that is being run.
- *Controller*: Receives directions from the client and distributes instructions and data to the computing nodes. Consists of a *hub* to manage communications and *schedulers* to assign processes to the engines.
- *Engines*: The individual processors. Each engine is like a separate Python terminal, each with its own namespace and computing resources.

Essentially, a Python program using iPyParallel creates a `Client` object connected to the cluster that allows it to send tasks to the cluster and retrieve their results. The engines run the tasks, and the controller manages which engines run which tasks.

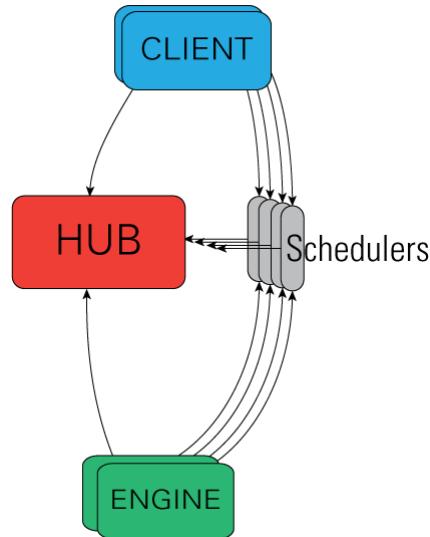


Figure 11.1: An outline of the iPyParallel architecture.

## Setting up an iPyParallel Cluster

Before being able to use iPyParallel in a script or interpreter, it is necessary to start an iPyParallel cluster. We demonstrate here how to use a single machine with multiple processor cores as a cluster. Establishing a cluster on multiple machines requires additional setup, which is detailed in the Additional Material section. The following commands initialize parts or all of a cluster when run in a terminal window:

Command	Description
<code>ipcontroller start</code>	Initialize a controller process.
<code>ipengine start</code>	Initialize an engine process.
<code>ipcluster start</code>	Initialize a controller process and several engines simultaneously.

Each of these processes can be stopped with a keyboard interrupt (`Ctrl+C`). By default, the controller uses JSON files in `UserDirectory/.ipython/profile-default/security/` to determine its settings. Once a controller is running, it acts like a server, listening connections from clients and engines. Engines will connect automatically to the controller when they start running. There is no limit to the number of engines that can be started in their own terminal windows and connected to the controller, but it is recommended to only use as many engines as there are cores to maximize efficiency.

#### ACHTUNG!

The directory that the controller and engines are started from matters. To facilitate connections, navigate to the same folder as your source code before using `ipcontroller`, `ipengine`, or `ipcluster`. Otherwise, the engines may not connect to the controller or may not be able to find auxiliary code as directed by the client.

Starting a controller and engines in individual terminal windows with `ipcontroller` and `ipengine` is a little inconvenient, but having separate terminal windows for the engines allows the user to see individual errors in detail. It is also actually more convenient when starting a cluster of multiple computers. For now, we use `ipcluster` to get the entire cluster started quickly.

```
$ ipcluster start          # Assign an engine to each processor core.
$ ipcluster start --n 4    # Or, start a cluster with 4 engines.
```

#### NOTE

Jupyter notebooks also have a **Clusters** tab in which clusters can be initialized using an interactive GUI. To enable the tab, run the following command. This operation may require root permissions.

```
$ ipcluster nbextension enable
```

## The iPyParallel Interface

Once a controller and its engines have been started and are connected, a cluster has successfully been established. The controller will then be able to distribute messages to each of the engines, which will compute with their own processor and memory space and return their results to the controller. The client uses the `ipyparallel` module to send instructions to the controller via a `Client` object.

```
>>> from ipyparallel import Client
```

```
>>> client = Client()      # Only works if a cluster is running.
>>> client.ids
[0, 1, 2, 3]              # Indicates that there are four engines running.
```

Once the client object has been created, it can be used to create one of two classes: a `DirectView` or a `LoadBalancedView`. These views allow for messages to be sent to collections of engines simultaneously. A `DirectView` allows for total control of task distribution while a `LoadBalancedView` automatically tries to spread out the tasks equally on all engines. The remainder of the lab will be focused on the `DirectView` class.

```
>>> dview = client[:]    # Group all engines into a DirectView.
>>> dview2 = client[:2] # Group engines 0,1, and 2 into a DirectView.
>>> dview2.targets      # See which engines are connected.
[0, 1, 2]
```

Since each engine has its own namespace, modules must be imported in every engine. There is more than one way to do this, but the easiest way is to use the `DirectView` object's `execute()` method, which accepts a string of code and executes it in each engine.

```
# Import NumPy in each engine.
>>> dview.execute("import numpy as np")
```

```
# Make sure to include client.close() after each function or else the test ←
     driver will time out
client.close()
```

Before continuing, set the `DirectView` you are using to use blocking:

```
>>> dview.block = True
```

This affects the way that functions called using the `DirectView` return their values. Using blocking makes the process simpler, so we will use it initially. What blocking is will be explained later.

**Problem 1.** Write a function that initializes a `Client` object, creates a `DirectView` (with blocking) with all available engines, and imports `scipy.sparse` as `sparse` on all engines. Return the `DirectView`. Note: Make sure to include `client.close()` after EVERY function or else the test driver will time out.

## Managing Engine Namespaces

We now discuss how to handle namespaces within each engine, beginning with setting and retrieving variables.

## Push and Pull

The `push()` and `pull()` methods of a `DirectView` object manage variable values in the engines. Use `push()` to set variable values and `pull()` to get variables. Each method also has a shortcut via indexing.

```
# Initialize the variables 'a' and 'b' on each engine.
>>> dview.push({'a': 10, 'b': 5})           # OR dview['a'] = 10; dview['b'] = 5
[None, None, None, None]                   # Output from each engine

# Check the value of 'a' on each engine.
>>> dview.pull('a')                      # OR dview['a']
[10, 10, 10, 10]

# Put a new variable 'c' only on engines 0 and 2.
>>> dview.push({'c': 12}, targets=[0, 2])
[None, None]
```

## Scatter and Gather

Parallelization almost always involves splitting up collections and sending different pieces to each engine for processing. The process is called *scattering* and is usually used for dividing up arrays or lists. The inverse process of pasting a collection back together is called *gathering* and is usually used on the results of processing. This method of distributing a dataset and collecting the results is common for processing large data sets using parallelization.

```
>>> import numpy as np

# Send parts of an array of 8 elements to each of the 4 engines.
>>> x = np.arange(1, 9)
>>> dview.scatter("nums", x)
>>> dview["nums"]
[array([1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]

# Scatter the array to only the first two engines.
>>> dview.scatter("nums_big", x, targets=[0,1])
>>> dview.pull("nums_big", targets=[0,1])
[array([1, 2, 3, 4]), array([5, 6, 7, 8])]

# Gather the array again.
>>> dview.gather("nums")
array([1, 2, 3, 4, 5, 6, 7, 8])

>>> dview.gather("nums_big", targets=[0,1])
array([1, 2, 3, 4, 5, 6, 7, 8])
```

## Executing Code on Engines

### Execute

The `execute()` method is the simplest way to run commands on parallel engines. It accepts a string of code (with exact syntax) to be executed. Though simple, this method works well for small tasks.

```
# "nums" is the scattered version of np.arange(1, 9).
>>> dview.execute("c = np.sum(nums)")    # Sum each scattered component.
<AsyncResult: execute:finished>
>>> dview['c']
[3, 7, 11, 15]
```

### Apply

The `apply()` method accepts a function and arguments to plug into it, and distributes them to the engines. Unlike `execute()`, `apply()` returns the output from the engines directly.

```
>>> dview.apply(lambda x: x**2, 3)
[9, 9, 9, 9]
>>> dview.apply(lambda x, y: 2*x + 3*y, 5, 2)
[16, 16, 16, 16]
```

Note that the engines can access their local variables in either of the execution methods.

### Map

The built-in `map()` function applies a function to each element of an iterable. The iPyParallel equivalent, the `map()` method of the `DirectView` class, combines `apply()` with `scatter()` and `gather()`. Simply put, it accepts a dataset, splits it between the engines, executes a function on the given elements, returns the results, and combines them into one object.

```
>>> num_list = [1, 2, 3, 4, 5, 6, 7, 8]
>>> def triple(x):                      # Map a function with a single input.
...     return 3*x
...
>>> dview.map(triple, num_list)
[3, 6, 9, 12, 15, 18, 21, 24]

>>> def add_three(x, y, z):            # Map a function with multiple inputs.
...     return x+y+z
...
>>> x_list = [1, 2, 3, 4]
>>> y_list = [2, 3, 4, 5]
>>> z_list = [3, 4, 5, 6]
>>> dview.map(add_three, x_list, y_list, z_list)
[6, 9, 12, 15]
```

## Blocking vs. Non-Blocking

Parallel commands can be implemented two ways. The difference is subtle but extremely important.

- *Blocking*: The main program sends tasks to the controller, and then waits for all of the engines to finish their tasks before continuing (the controller "blocks" the program's execution). This mode is usually best for problems in which each node is performing the same task.
- *Non-Blocking*: The main program sends tasks to the controller, and then continues without waiting for responses. Instead of the results, functions return an `AsyncResult` object that can be used to check the execution status and eventually retrieve the actual result.

Whether a function uses blocking is determined by default by the `block` attribute of the `DirectView`. The execution methods `execute()`, `apply()`, and `map()`, as well as `push()`, `pull()`, `scatter()`, and `gather()`, each have a keyword argument `block` that can instead be used to specify whether or not to use blocking. Alternatively, the methods `apply_sync()` and `map_sync()` always use blocking, and `apply_async()` and `map_async()` always use non-blocking.

```
>>> f = lambda n: np.sum(np.random.random(n))

# Evaluate f(n) for n=0,1,...,999 with blocking.
>>> %time block_results = [dview.apply_sync(f, n) for n in range(1000)]
CPU times: user 9.64 s, sys: 879 ms, total: 10.5 s
Wall time: 13.9 s

# Evaluate f(n) for n=0,1,...,999 with non-blocking.
>>> %time responses = [dview.apply_async(f, n) for n in range(1000)]
CPU times: user 4.19 s, sys: 294 ms, total: 4.48 s
Wall time: 7.08 s

# The non-blocking method is faster, but we still need to get its results.
# Both methods produced a list, although the contents are different
>>> block_results[10] # This list holds actual result values from each engine.
[3.833061790352166,
 4.8943956129713335,
 4.268791758626886,
 4.73533677711277]

>>> responses[10]           # This list holds AsyncResult objects.
<AsyncResult: <lambda>:finished>
# We can get the actual results by using the get() method of each AsyncResult
>>> %time nonblock_results = [r.get() for r in responses]
CPU times: user 3.52 ms, sys: 11 mms, total: 3.53 ms
Wall time: 3.54 ms          # Getting the responses takes little time.

>>> nonblock_results[10]     # This list also holds actual result values
[5.652608204341693,
 4.984164642641558,
 4.686288406810953,
 5.275735658763963]
```

When non-blocking is used, commands can be continuously sent to engines before they have finished their previous task. This allows them to begin their next task without waiting to send their calculated answer and receive a new command. However, this requires a design that incorporates checkpoints to retrieve answers and enough memory to store response objects.

Class Method	Description
<code>wait(timeout)</code>	Wait until the result is available or until <code>timeout</code> seconds pass.
<code>ready()</code>	Return whether the call has completed.
<code>successful()</code>	Return whether the call completed without raising an exception.
<code>get(timeout)</code>	Will raise <code>AssertionError</code> if the result is not ready. Return the result when it arrives. If <code>timeout</code> is not <code>None</code> and the result does not arrive within <code>timeout</code> seconds then <code>TimeoutError</code> is raised.

Table 11.1: All information from <https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult>.

Table 11.1 details the methods of the `AsyncResult` object.

There are additional magic methods supplied by `iPyParallel` that make some of these operations easier. These methods are explained in the Additional Material section. More information on `iPyParallel` architecture, interface, and methods can also be found at <https://ipyparallel.readthedocs.io/en/latest/index.html>.

**Problem 2.** Write a function that accepts an integer  $n$ . Instruct each engine to make  $n$  draws from the standard normal distribution, then hand back the mean, minimum, and maximum draws to the client. Return the results in three lists.

If you have four engines running, your results should resemble the following:

```
>>> means, mins, maxs = problem3(1000000)
>>> means
[0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
>>> mins
[-4.1508589, -4.3848019, -4.1313324, -4.2826519]
>>> maxs
[4.0388107, 4.3664958, 4.2060184, 4.3391623]
```

**Problem 3.** Use your function from Problem 2 to compare serial and parallel execution times. For  $n = 1000000, 5000000, 10000000, 15000000$ ,

1. Time how long it takes to run your function.
2. Time how long it takes to do the same process serially. Make  $n$  draws and then calculate and record the statistics, but use a `for` loop with  $N$  iterations, where  $N$  is the number of engines running.

Plot the execution times against  $n$ . You should notice an increase in efficiency in the parallel version as the problem size increases.

## Applications

Parallel computing, when used correctly, is one of the best ways to speed up the run time of an algorithm. As a result, it is very commonly used today and has many applications, such as the following:

- Graphic rendering
- Facial recognition with large databases
- Numerical integration
- Calculating discrete Fourier transforms
- Simulation of various natural processes (weather, genetics, etc.)
- Natural language processing

In fact, there are many problems that are only feasible to solve through parallel computing because solving them serially would take too long. With some of these problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack simple encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in such a way that running them on multiple processors results in a significant run time improvement. Manipulating an algorithm to be run with parallel computing is called *parallelizing* the algorithm. When a problem only requires very minor manipulations to parallelize, it is often called *embarrassingly parallel*. Typically, an algorithm is embarrassingly parallel when there is little to no dependency between results. Algorithms that do not meet this criteria can still be parallelized, but there is not always a significant enough improvement in run time to make it worthwhile. For example, calculating the Fibonacci sequence using the usual formula,  $F(n) = F(n - 1) + F(n - 2)$ , is poorly suited to parallel computing because each element of the sequence is dependent on the previous two elements.

**Problem 4.** The *trapezoid rule* is a simple technique for numerical integration:

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{k=1}^{N-1} (f(x_k) + f(x_{k+1})),$$

where  $a = x_1 < x_2 < \dots < x_N = b$  and  $h = x_{n+1} - x_n$  for each  $n$ . See Figure 11.2.

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered embarrassingly parallel.

Write a function that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Parallelize the trapezoid rule in order to estimate the integral of  $f$ . That is, divide the points among all available processors and run the trapezoid rule on each portion simultaneously. Be sure that the intervals for each processor share endpoints so that the subintervals are not skipped, resulting in rather large amounts of error. The sum of the results of all the processors will be the estimation of the integral over the entire interval of integration. Return this sum (consider coding the problem in serial to test your function).

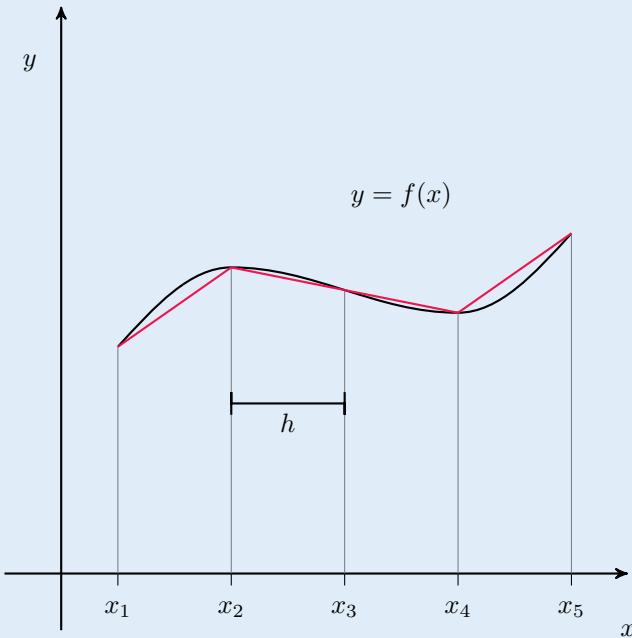


Figure 11.2: A depiction of the trapezoid rule with uniform partitioning.

## Intercommunication

The phrase *parallel computing* refers to designing an architecture and code that makes the best use of computing resources for a problem. Occasionally, this will require nodes to be interdependent on each other for previous results. This contributes to a slower result because it requires a great deal of communication latency, but is sometimes the only method to parallelize a function. Although important, the ability to effectively communicate between engines has not been added to iPyParallel. It is, however, possible in an MPI framework and will be covered in the MPI lab.

## Additional Material

### Clusters of Multiple Machines

Though setting up a computing cluster with `iPyParallel` on multiple machines is similar to a cluster on a single computer, there are a couple of extra considerations to make. The majority of these considerations have to do with the network setup of your machines, which is unique to each situation. However, some basic steps have been taken from <https://ipyparallel.readthedocs.io/en/latest/process.html> and are outlined below.

#### SSH Connection

When using engines and controllers that are on separate machines, their communication will most likely be using an SSH tunnel. This *Secure Shell* allows messages to be passed over the network.

In order to enable this, an SSH user and IP address must be established when starting the controller. An example of this follows.

```
$ ipcontroller --ip=<controller IP> --user=<user of controller> --enginessh=<←
    user of controller>@<controller IP>
```

Engines started on remote machines then follow a similar format.

```
$ ipengine --location=<controller IP> --ssh=<user of controller>@<controller IP←
    >
```

Another way of affecting this is to alter the configuration file in `UserDirectory/.ipython/profile-default/security/ipcontroller-engine.json`. This can be modified to contain the controller IP address and SSH information.

All of this is dependent on the network feasibility of SSH connections. If there are a great deal of remote engines, this method will also require the SSH password to be entered many times. In order to avoid this, the use of SSH Keys from computer to computer is recommended.

### Magic Methods & Decorators

To be more easily usable, the `iPyParallel` module has incorporated a few magic methods and decorators for use in an interactive iPython or Python terminal.

#### Magic Methods

The `iPyParallel` module has a few magic methods that are very useful for quick commands in iPython or in a Jupyter Notebook. The most important are as follows. Additional methods are found at <https://ipyparallel.readthedocs.io/en/latest/magics.html>.

**%px** - This magic method runs the corresponding Python command on the engines specified in `dview.targets`.

**%autopx** - This magic method enables a boolean that runs any code run on every engine until `%autopx` is run again.

Examples of these magic methods with a client and four engines are as follows.

```
# %px
In [4]: with dview.sync_imports():
....:     import numpy
....:
importing numpy on engine(s)
In [5]: \%px a = numpy.random(2)

In [6]: dview['a']
Out[6]:
[array([ 0.30390162,  0.14667075]),
 array([ 0.95797678,  0.59487915]),
 array([ 0.20123566,  0.57919846]),
 array([ 0.87991814,  0.31579495])]

# %autopx
In [7]: %autopx
%autopx enabled
In [8]: max_draw = numpy.max(a)

In [9]: print("Max_Draw: {}".format(max_draw))
[stdout:0] Max_Draw: 0.30390161663280246
[stdout:1] Max_Draw: 0.957976784975849
[stdout:2] Max_Draw: 0.5791984571339429
[stdout:3] Max_Draw: 0.8799181411958089

In [10]: %autopx
%autopx disabled
```

## Decorators

The `iPyParallel` module also has a few decorators that are very useful for quick commands. The two most important are as follows:

`@remote` - This decorator creates methods on the remote engines.

`@parallel` - This decorator creates methods on remote engines that break up element wise operations and recombine results.

Examples of these decorators are as follows.

```
# Remote decorator
>>> @dview.remote(block=True)
>>> def plusone():
...     return a+1
>>> dview['a'] = 5
>>> plusone()
[6, 6, 6, 6,]
```

```
# Parallel decorator
>>> import numpy as np

>>> @dview.parallel(block=True)
>>> def combine(A, B):
...     return A+B
>>> ex1 = np.random.random((3, 3))
>>> ex2 = np.random.random((3, 3))
>>> print(ex1+ex2)
[[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
>>> print(combine(ex1, ex2))
[[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
```



# 12

# Linear Regression

**Lab Objective:** This section will introduce the basics of Linear Regression, feature selection methods, and regularization.

## Introduction to Linear Regression

One of the first skills taught in basic algebra is to effectively plot the line  $y = mx + b$  which can be done with two points. But what if we want to find the line that best fits a set of points?

In this case, we can use the simplest form of linear regression: *Ordinary Least Squares (OLS)*. Given data as a set of points  $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$  we wish to find the line that best fits the data. The line is given by  $y = mx + b$  where  $m$  and  $b$  are unknown constants and  $x$  and  $y$  are the independent and dependent variables respectively. Using OLS, let

$$y_i = mx_i + b + \varepsilon_i$$

describe the  $i$ th point in  $D$  for each  $i \in \{1, \dots, n\}$ . Note that  $\varepsilon_i$  is the vertical distance from the  $i$ th point to the line given by  $y = mx + b$  and is often called the *residual* or the *error*.

The  $n$  equations for each point in  $D$  can be written in vector notation. Let the  $x$  and  $y$  coordinates of  $D$  be represented by column vectors  $\mathbf{x}$  and  $\mathbf{y}$  respectively. In statistical science, the intercept ( $b$ ) and slope ( $m$ ) are denoted as  $\beta_0$  and  $\beta_1$  respectively and

$$\begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \boldsymbol{\beta}.$$

Additionally, the residuals are represented by a column vector  $\boldsymbol{\varepsilon}$  and  $\mathbf{1}$  is a column vector of ones. So we have

$$\mathbf{y} = m\mathbf{x} + \mathbf{1}b + \boldsymbol{\varepsilon} = [\mathbf{1}, \mathbf{x}] \cdot \begin{bmatrix} b \\ m \end{bmatrix} + \boldsymbol{\varepsilon}.$$

Denoting  $X = [\mathbf{1}, \mathbf{x}]$ , we have our final equation given as

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon}.$$

This notation may seem excessive, but suppose we wanted to fit a model of the form  $y = ax^3 + bx^2 + cx + d$ . A little work can show that  $X = [\mathbf{1}, \mathbf{x}, \mathbf{x}^2, \mathbf{x}^3]$  and  $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, \beta_3]^\top$ , which is very easy to work with. Thus, this notation is actually the ideal way to generalize linear regression, especially when working with higher degree polynomials.

The solution to OLS is straight forward with some important assumptions. Sparing you the algebraic details and assuming that  $\mathbf{y} \sim \mathcal{N}(X\beta, \sigma^2 \mathbf{I})$  and  $\varepsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$  and  $\mathbf{I}$  is the identity matrix, the least squares estimator for  $\beta$  is given as

$$\hat{\beta} = (X^\top X)^{-1} X^\top \mathbf{y}. \quad (12.1)$$

**Problem 1.** Write a function that takes as input  $X$  and  $y$ . In your function, add a column of ones to  $X$  to account for  $\beta_0$ . Call this function `ols`. This function should return the least squares estimator for  $\beta$  as a numpy array.

Hint: Try rearranging equation (12.1) and use `np.linalg.solve()` to avoid inverting  $(X^\top X)$ .

**Problem 2.** Use the following code to generate random data.

```
n = 100 # Number of points to generate
X = np.arange(100) # The input X for the function ols
eps = np.random.uniform(-10, 10, size=(100,)) # Noise to generate random y←
    coordinates
y = .3*X + 3 + eps # The input y for the function ols
```

Find the least squares estimator for  $\beta$  using this random data. Produce a plot showing the random data and the line of best fit determined by the least squares estimator for  $\beta$ . Your plot should include a title, axis labels, and a legend.

## Rank-Deficient Models

Notice that in order to find the least squares estimator  $\hat{\beta}$ , we need  $X^\top X$  to be invertible. However, when  $X$  does not have full rank, the product  $X^\top X$  is singular and not invertible. We can no longer use the previous solution for the least squares estimator, but we can use the SVD and still compute a solution.

Recall that if  $X \in M_{n \times d}$  has rank  $r$ , then the compact form of the SVD of  $X$  is

$$X = U\Sigma V^H$$

where  $U \in M_{n \times r}$  and  $V \in M_{r \times d}$  have orthonormal columns and  $\Sigma \in M_{r \times r}$  is diagonal. In addition, if  $X$  is real, then the factors  $U$ ,  $\Sigma$ , and  $V^H$  are also real. In this lab we assume  $X$  is real. As described in Volume 1, there is a unique solution for the least squares estimator given by

$$\hat{\beta} = V\Sigma^{-1}U^\top \mathbf{y}. \quad (12.2)$$

**Problem 3.** Write a function that finds the least squares estimator for rank-deficient models using the SVD. The function should still take  $X$  and  $y$  as inputs. In your function, add a column of ones to  $X$  to account for  $\beta_0$ . Call the function `svd_ols` and return the least squares estimator for  $\beta$  as a numpy array.

Hint: Use `np.linalg.svd` to factor  $X$  and use the argument `full_matrices=False`. Consider solving for  $\Sigma^{-1}$  using the command, `np.diag(1/s)`, where  $s$  is the second thing being returned in `np.linalg.svd`.

**Problem 4.** Use the following code to generate random data:

```
x = np.linspace(-4, 2, 100)
y = x**3 + 3*x**2 - x - 3.5
eps = np.random.normal(0, 3, len(y)) # Create noise
y += eps # Add noise to randomize data
```

Now use your function `svd_ols` to find the least squares estimator for a cubic polynomial. This can be done by passing in  $X = [x, x^2, x^3]$  into `svd_ols`. Create a plot that shows a scatter plot of the data and a curve using the least squares estimator. Your plot should include a title, axis labels, a legend, and should look similar to Figure 12.1.

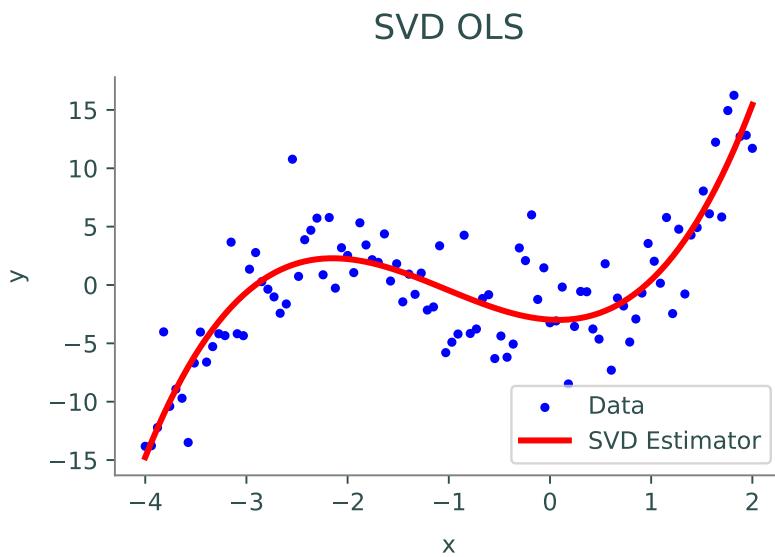


Figure 12.1: This is the SVD estimator for a cubic polynomial.

## Model Accuracy

### Residual Sum of Squares

The *Residual Sum of Squares* (*RSS*) is a common choice of measure for the quality of a model. The formula for *RSS* is given by

$$RSS = \|\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\|_2^2.$$

Notice that the *RSS* measures the variance in the error of the model. So relative to other models, a smaller *RSS* value indicates a more accurate model.

## Coefficient of Determination

Another method of model accuracy is the *Coefficient of Determination*, denoted  $R^2$ . In the case of linear regression,

$$R^2 = 1 - \frac{RSS}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

and  $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$  is the sample mean of  $y$ . The intuition of  $R^2$  is that the ratio of the average residual and biased sample variance of  $y$  is approximately the total variance explained by the model. A larger  $R^2$  corresponds to a model that fits better. However,  $R^2$  comes with flaws such as being able to take negative values, rewarding overfitting, and punishing under-fit models. Because of this, we typically want to use other methods for model accuracy.

## Python Example

There are various python packages that can be used to calculate  $R^2$ , but we will use `statsmodels` in this lab. Below is an example of how to build a model and extract  $R^2$  using `statsmodels`.

```
import statsmodels.api as sm
data = pd.read_csv("/filepath") # Read in data as pandas dataframe
y = data["dependent_variable"] # Extract dependent variable
temp_X = data[["var_1", ..., "var_n"]] # Extract independent variables
X = sm.add_constant(temp_X) # Add column of 1's
model = sm.OLS(y, X).fit() # Fit the linear regression model
print(model.rsquared) # Print the R squared value
```

**Problem 5.** The file `realestate.csv` contains transaction data from 2012-2013. It has columns for transaction data, house age, distance to nearest MRT station, number of convenience stores, latitude, longitude, and house price of unit area.<sup>a</sup> Each row in the array is a separate measurement. As independent variables, use `house age`, `distance to the nearest MRT station`, `number of convenience stores`, `latitude`, and `longitude`.

Find the combination of independent variables that builds the model with the best  $R^2$  value when predicting `house price of unit area`, the dependent variable. Use `statsmodels` to build each model and calculate  $R^2$ . Using the same combination of variables, time the methods `ols`, `svd_ols`, and `statsmodels`. Return a list with the first element being a tuple of times for each method and the second element being the best  $R^2$  value from the first part of the problem.

Note that  $R^2$  cannot get worse by adding more columns and also rewards overfitting, so solving for the  $R^2$  value isn't the greatest in practice. The purpose of this problem is to explore the issues of using  $R^2$ .

Hint: The `combinations` method from the `itertools` package will be very helpful for finding all feature combinations.

<sup>a</sup>See <https://www.kaggle.com/datasets/quantbruce/real-estate-price-prediction?resource=download>.

## Feature Selection

Every regression model consists of features or variables used to predict a dependent variable or result. An important question to ask when building regression models is, which features are the most important in predicting the dependent variable? In addition to being used for model accuracy,  $R^2$  can also be used in feature selection, as it was in Problem 5. It still has the same pitfalls of rewarding overfitting and punishing under-fit models, but it can be a useful tool used in conjunction with the following tools for feature selection. While there are other methods for implementing feature selection, most incorporate the p-value and are not included in this lab.

### Akaike's Information Criterion (AIC)

A simple motivation for AIC is based on balancing goodness of fit and prescribing a penalty for model complexity. A more rigorous motivation for AIC is given in Volume 3 using the *Kullback-Leibler* (KL) divergence. Given two models,  $f$  and  $g$ , the KL divergence is given by

$$KL(f, g) = \int f(z) \log \left( \frac{f(z)}{g(z)} \right) dz$$

and it measures the amount of information lost when  $g$  is used to model  $f$ . Thus, a lower AIC value indicates a better model. Additionally, AIC penalizes the size of the parameter space with a coefficient of 2 which allows for slightly more complex models.

### Bayesian Information Criterion (BIC)

Instead of estimating the KL-divergence between the model in question and the true model, BIC has the property of being minimized precisely when the posterior probability of a model, given the data, is maximized. The equations for AIC and BIC only differ with one term: the coefficient weighting the size of the parameter space. The coefficient for BIC is  $\log(n)$  which is generally much larger than 2. As a result, BIC penalizes complex models more than AIC. The difference in AIC and BIC values will grow from having more data points.

When using AIC or BIC for feature selection, you need to consider how you want to penalize features in your model. If you want to exclude irrelevant features, then use BIC. If you want to keep all features that are relevant, then use AIC. In other words, BIC is more likely to choose too small a model, and AIC is more likely to choose too large a model.

### Python Example

There are multiple ways to calculate AIC and BIC with various python packages. We will use the package `statsmodels` for the following problem. When constructing  $X$  for `statsmodels`, do not add the column of 1's manually because `statsmodels` has a method that will do this for us.

```
import statsmodels.api as sm
data = pd.read_csv("/filepath") # Read in data as pandas dataframe
y = data["dependent_variable"] # Extract dependent variable
temp_X = data[["var_1", ..., "var_n"]] # Extract independent variables
X = sm.add_constant(temp_X) # Add column of 1's
model = sm.OLS(y, X).fit() # Fit the linear regression model
print(model.aic) # or print(model.bic)
```

**Problem 6.** Use the file `realestate.csv` and the Python Example above as a template for constructing  $\mathbf{y}$  and  $\mathbf{X}$  and calculating model AIC and BIC. For the dependent variable, use `house price of unit area`. For the independent variables, use `house age`, `distance to the nearest MRT station`, `number of convenience stores`, `latitude`, and `longitude`. Loop through all of the combinations of these variables and create OLS models for each of these combinations. Solve for the AIC and BIC for each of these models.

Find the model that has the lowest AIC and the model that has the lowest BIC. Print the features of the model with the lowest AIC and then the features of the model with the lowest BIC as separate lists.

Hint: The `combinations` method from the `itertools` package will be very helpful for finding all feature combinations.

## Regularization

Up to this point, we have been solving the problem

$$\min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2.$$

However, we have also assumed independence among the features used to predict the dependent variable. The pitfall of multicollinearity arises when the features of  $\mathbf{X}$  have dependence and  $\mathbf{X}$  becomes nearly singular. As a result, the least squares estimator is susceptible to random noise or error. Multicollinearity typically occurs when data is collected with poor experimental design. It is important to have good experimental design, but regularization can be used to mitigate poor design. Another issue OLS faces is feature selection. While there are feature selection methods available, regularization can be used to minimize non-zero coefficients.

### Ridge Regularization Regression

The problem posed by *Ridge Regularization* is

$$\min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_2^2$$

where  $\alpha \geq 0$ . This essentially penalizes the size of the coefficients. The larger  $\alpha$  is, the more the model resists multicollinearity.

### Lasso Regularization Regression

The problem posed by *Lasso Regularization* is

$$\min_{\boldsymbol{\beta}} \frac{1}{n} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_1.$$

Note that  $\alpha$  provides the same functionality here as it does in Ridge Regularization. However, the use of the 1-norm often results in sparse solutions. As a result, Lasso Regularization can be used for feature selection since it only includes the most important features.

## Python Example

Since  $\alpha$  is not a fixed value in Ridge and Lasso Regularization, it is best practice to perform a Grid-Search to find the best parameter value. The example below goes over the syntax for implementing Ridge Regularization. Note that the syntax for Lasso Regularization is similar.

```
>>> from sklearn import linear_model
>>> y = # dependent variable data
>>> X = # independent variable data with no column of ones
>>> reg = linear_model.RidgeCV(alphas=np.logspace(-6, 6, 13)) # Range for grid ←
    search
>>> reg.fit(X, y) # Fit the model
>>> reg.alpha_ # Best parameter value
```

**Problem 7.** Use Ridge and Lasso Regression to model house price of unit area from the file `realestate.csv`. Use the same columns for your independent variables as you did in Problem 6. First, do a grid search for the model parameter for both the Ridge and the Lasso models, separately. Note that the objects `RidgeCV` and `LassoCV` have built in cross-validation. Be sure to pass in `alphas=np.logspace(-6, 6, 13)` as grid parameters for each of the models. Then use the grid search result to fit each model. Once you have fit the model, you can use the `score` method to get  $R^2$ . Print  $R^2$  for each model as a tuple.



# 13

## Logistic Regression

**Lab Objective:** *Understand the basic principles of Logistic Regression and binary classifiers. Apply this to a dataset.*

Linear regression is unsuitable for predicting probabilities, because the resulting model may take values in any fixed interval in  $\mathbb{R}$ , but a probability-predicting model can only take values in the interval  $[0, 1]$ . *Logistic regression* is a form of regression that always takes its values in the interval  $[0, 1]$  and as such, is a popular method for predicting probabilities and for constructing *classifiers*. As in linear regression, in a classification problem we have a random variable  $Y$ , conditioned on an input  $X \in \mathbb{R}^d$ . However, in *binary classification* problems the random variable  $Y$  is binary, that is,  $Y \in \{0, 1\}$ . A *binary classifier* is any function  $f$  taking values in  $\{0, 1\}$ . For example,  $\mathbf{x} \in \mathbb{R}^d$  could be the pixel intensities of an image, and the classifier  $f$  gives 1 if the image is a picture of a duck and 0 otherwise. The goal of a classification problem is to choose a classifier  $\hat{f}$  so that  $(X, \hat{f}(X))$  is a good approximation for  $(X, Y)$ .

### Logistic Regression

Logistic regression relies heavily on the *logistic function*, also known as the *sigmoid function*,  $\text{sigm} : \mathbb{R} \rightarrow (0, 1)$  given by

$$\text{sigm}(x) = \frac{1}{1 + e^{-x}}. \quad (13.1)$$

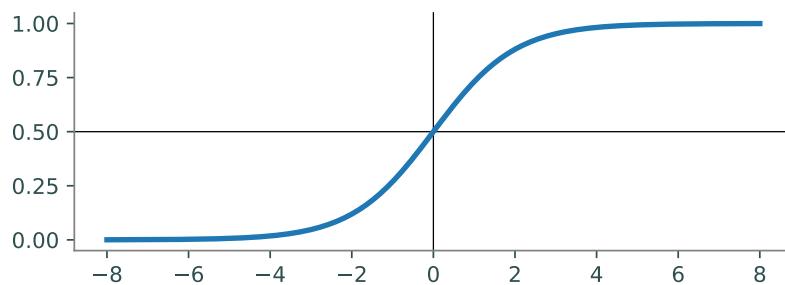


Figure 13.1: Sigmoid Function

This function works well for classifying objects based on probabilities, because it has some key properties that translate well into probability theory. Of particular note, the graph can be translated by adding a constant, giving the form  $\text{sigm}(\beta_1 t + \beta_0)$ . A larger value of  $\beta_1$  makes the ramp up from 0 to 1 steeper, while a smaller value of  $\beta_1$  makes it less steep. The trick behind logistic regression is to find the values of  $\beta_i$  such that the resulting sigmoid function best classifies the data.

In logistic regression models we have a random variable  $Y$  with support  $\{0, 1\}$ , where  $Y$  is conditioned on another random variable  $X$ , with support in  $\mathbb{R}^d$ . The distribution of  $Y$ , given  $X$ , is assumed to be Bernoulli

$$Y | X \sim \text{Bernoulli}(\text{sigm}(X^\top \boldsymbol{\beta})),$$

so that

$$P(Y | X) = \text{sigm}(X^\top \boldsymbol{\beta}) = \frac{1}{1 + \exp(-X^\top \boldsymbol{\beta})}.$$

As in the case of linear regression, we usually add a constant feature  $X_0 = \mathbf{1}$  to  $X$  and a corresponding coefficient  $\beta_0$  to  $\boldsymbol{\beta}$ , so that  $X^\top \boldsymbol{\beta} = \beta_0 + \beta_1 X_1 + \dots + \beta_d X_d$ . Given a draw of length  $n$  of the form  $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$  we wish to estimate  $\boldsymbol{\beta}$ . The maximum likelihood estimator is a good choice. To find this estimator, first observe that the likelihood of  $\boldsymbol{\beta}$ , given the data, is

$$\begin{aligned} L(\boldsymbol{\beta} | D) &= \prod_{i=1}^n P(Y = y_i, X = \mathbf{x}_i | \boldsymbol{\beta}) \\ &= \prod_{i=1}^n P(Y = y_i | X = \mathbf{x}_i, \boldsymbol{\beta}) P(X_i). \end{aligned}$$

which is equivalent to maximizing

$$\prod_{i=1}^n P(Y = y_i | X = \mathbf{x}_i, \boldsymbol{\beta}) = \prod_{i=1}^n p_i^{y_i} (1 - p_i)^{1-y_i}.$$

where

$$p_i = P(Y = 1 | \mathbf{x}_i, \boldsymbol{\beta}) = \text{sigm}(\mathbf{x}_i^\top \boldsymbol{\beta}) = \frac{1}{1 + \exp(-\mathbf{x}_i^\top \boldsymbol{\beta})}.$$

Taking the negative logarithm turns this into a convex minimization problem, and a little math shows that

$$\ell(\boldsymbol{\beta} | D) = \sum_{i=1}^n (y_i \log(1 + \exp(-\mathbf{x}_i^\top \boldsymbol{\beta})) + (1 - y_i) \log(1 + \exp(\mathbf{x}_i^\top \boldsymbol{\beta}))). \quad (13.2)$$

The convexity of this problem implies there is a unique minimizer  $\hat{\boldsymbol{\beta}}$  of  $\ell(\boldsymbol{\beta} | D)$ .

**Problem 1.** Write a `fit()` method in the Python `LogiReg` classifier that accepts an  $(n \times 1)$  array `y` of binary labels (0's and 1's) as well as an  $(n \times d)$  array `X` of data points that uses equation 13.2 to find and save the optimal  $\hat{\boldsymbol{\beta}}$ . Save `X`, `y`, and  $\hat{\boldsymbol{\beta}}$  as attributes. Remember to add a column of ones to `X` before implementing the `fit` algorithm.

Once the maximum likelihood estimate  $\hat{\beta}$  is found, we have an estimate for the probability

$$P(Y = 1 \mid \mathbf{x}) \approx \text{sigm}(\mathbf{x}^T \hat{\beta}).$$

From this, we can construct a classifier  $\hat{f}$  by setting  $\hat{f}(x) = 1$  if  $P(Y = 1 \mid \mathbf{x}) \geq \frac{1}{2}$  and  $\hat{f}(x) = 0$  otherwise.

**Problem 2.** Write a method called `predict_prob()` for your classifier that accepts an  $(n \times d)$  array `x_test` and returns  $P(Y = 1 \mid \mathbf{x}_{\text{test}})$ . Also write a method called `predict()` that calls `predict_prob()` and returns an array of predicted labels (0's or 1's) for the given array `x_test`. Remember to add a column of ones to `x_test` in your `predict` function like you did in the previous problem with `X`.

**Problem 3.** To test your classifier, create training arrays `X` and `y` as well as a testing array `X_test`. The code to generate `X`, `y` and `X_test` is provided below. Both `X` and `X_test` have 100 random draws from a 2-dimensional multivariate normal distribution centered at  $(1, 2)$ , and another 100 draws from one centered at  $(4, 3)$ .

Train your classifier on `X` and `y`. Then generate a list of predicted labels using your trained classifier and `X_test`, and use it to plot `X_test` with a different color for each predicted label. Your plot should look similar to Figure 13.2. If you didn't add a constant column in Problem 1, go back and do that. This will allow the decision boundary between the two classes to not intersect the origin.

```
>>> import numpy as np

>>> data = np.column_stack(
    # draw from 2 2-dim. multivariate normal dists.
    np.concatenate((
        np.random.multivariate_normal(np.array([1, 2]), np.eye(2), 100),
        np.random.multivariate_normal(np.array([4, 3]), np.eye(2), 100)
    )),
    # labels corresponding to each distribution
    np.concatenate((np.zeros(100), np.ones(100))) )
>>> np.random.shuffle(data)
>>> # extract X and y from the shuffled data
>>> X = data[:, :2]
>>> y = data[:, 2].astype(int)

>>> X_test = np.concatenate(
    # draw from 2 identical 2-dim. multivariate normal dists.
    np.random.multivariate_normal(np.array([1, 2]), np.eye(2), 100),
    np.random.multivariate_normal(np.array([4, 3]), np.eye(2), 100)
)
>>> np.random.shuffle(X_test)
```

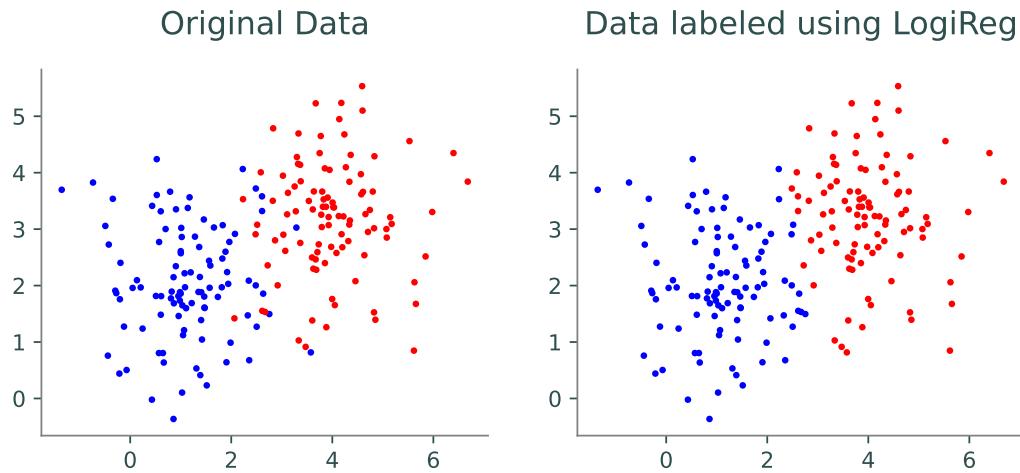


Figure 13.2: In reality, these two distributions overlap a little, but the logistic regression model makes a clean divide between the two.

## Statsmodels and Sklearn

The module `statsmodels` contains a package that includes a logistic regression class called `Logit`. A simple example of this class being implemented is as follows.

```
>>> import statsmodels.api as sm

>>> model = sm.Logit(y, X).fit(disp=0) # setting disp=0 turns off printed info
>>> probs = model.predict(X_test) # list of probabilities, not labels
```

`Logit` does *not* add a constant feature (column of 1's) to `X` by default, so in order to do so, you must apply the function `sm.add_constant()` to both `X` and `X_test`. In addition, the `.fit()` method does not regularize the problem by default, which may lead to some errors involving singular matrices. To fix this, you can use the `.fit_regularized()` method instead of `.fit()`.

The module `sklearn` also has a package for logistic regression called `LogisticRegression`, which can be implemented as follows.

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

>>> model = LogisticRegression(fit_intercept=True).fit(X, y) # X before y
>>> labels = model.predict(X_test) # predicted labels of X_test
```

`LogisticRegression` already regularizes the problem by default. The parameter `fit_intercept` (which defaults to `False`) indicates whether you want to add a constant feature (column of 1's) to `X` and `X_test`.

You can also use `sklearn` to score a logistic regression model. After fitting an `sklearn` model, you can call `<model>.score(X_test, y_test)` to find the percentage of accuracy of the model's prediction for `X_test`, given the true labels in `y_test`. Alternatively, you can use `sklearn.metrics.accuracy_score` to find the percentage of accuracy between a list of predicted labels and the list of true labels.

```
>>> from sklearn.metrics import accuracy_score

>>> true_labels = [0, 1, 2, 3, 4]
>>> pred_labels = [0, 2, 2, 2, 4] # predicted labels from logistic regression
>>> accuracy_score(true_labels, predicted_labels)
0.6
```

## Support Vector Machines

Support Vector Machines (SVM) are another type of classifier. It uses what is called the 'kernel trick' to handle nonlinear input spaces. It classifies data by finding an optimal hyperplane to split the data.

```
>>> from sklearn import svm

>>> svm = svm.SVC(kernel = 'linear').fit(X, y)
>>> labels = svm.predict(X_test) # predicted labels of X_test
```

**Problem 4.** The code to generate arrays `X`, `y`, `X_test`, and `y_test` is provided below. `X` and `X_test` are each composed of 200 draws from two 20-dimensional multivariate normal distributions, one centered at `0`, and the other centered at `2`.

Using each of `LogiReg`, `statsmodels`, `sklearn.LogisticRegression`, and `sklearn.svm`, train a logistic regression classifier on `X` and `y` to generate a list of predicted labels for `X_test`. Then, using `y_test`, print the accuracy scores for each trained model. Compare the accuracies and training/testing time for all four classifiers. Be sure to add a constant feature with each model.

```
>>> # redefine the true beta
>>> beta = np.random.normal(0, 7, 20)

>>> # X is generated from 2 20-dim. multivariate normal dists.
>>> X = np.concatenate((
    np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
    np.random.multivariate_normal(np.ones(20)*2, np.eye(20), 100)
))
>>> np.random.shuffle(X)
>>> # create y based on the true beta
>>> pred = 1. / (1. + np.exp(-X @ beta))
>>> y = np.array([1 if pred[i] >= 1/2 else 0
```

```

    for i in range(pred.shape[0]): )

>>> # X_test and y_test are generated similar to X and y
>>> X_test = np.concatenate((
    np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
    np.random.multivariate_normal(np.ones(20), np.eye(20), 100)
))
>>> np.random.shuffle(X_test)
>>> pred = 1. / (1. + np.exp(-X_test @ beta))
>>> y_test = np.array([1 if pred[i] >= 1/2 else 0
    for i in range(pred.shape[0])])

```

Hint: Consider using the command `fit_regularized(disp = 0)` for the `statsmodels` case.

## Multiclass Logistic Regression

Sometimes we may want to classify data into more than two categories, but so far we've only used logistic regression as a binary classifier. The good news is that we can extend logistic regression to classify more than just two categories.

The more popular method for doing this is to generalize the logistic regression model to a multiclass setting. This method is called *multinomial logistic regression* or sometimes *softmax regression*. While standard logistic regression was based on the sigmoid function, multinomial logistic regression is based on the *softmax function*  $\mathcal{S} : \mathbb{R}^k \rightarrow (0, 1)^k$ , which is a multivariate version of the sigmoid function, given by

$$\mathcal{S}(t_1, \dots, t_k) = \left( \frac{e^{t_1}}{\sum_{j=1}^k e^{t_j}}, \dots, \frac{e^{t_k}}{\sum_{j=1}^k e^{t_j}} \right). \quad (13.3)$$

We will assume that  $Y | X$  is categorically distributed as

$$\text{Cat}(p_1(X), \dots, p_k(X)) = \text{Cat}(\mathcal{S}(X^\top \boldsymbol{\beta}_1, \dots, X^\top \boldsymbol{\beta}_k))$$

for some choice of vectors  $\boldsymbol{\beta}_1, \dots, \boldsymbol{\beta}_k$ , which we will estimate from the data. Here

$$p_i(X) = P(Y = i | X) = \frac{e^{X^\top \boldsymbol{\beta}_i}}{\sum_{j=1}^k e^{X^\top \boldsymbol{\beta}_j}} = \frac{\text{sigm}(X^\top \boldsymbol{\beta}_i)}{\sum_{j=1}^k \text{sigm}(X^\top \boldsymbol{\beta}_j)}.$$

Given a draw of length  $n$  of the form  $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ , we wish to compute  $\boldsymbol{\theta} = (\boldsymbol{\beta}_1, \dots, \boldsymbol{\beta}_k)$  where, without loss of generality, we may assume  $\boldsymbol{\beta}_k = \mathbf{0}$ . The maximum likelihood estimate of  $\boldsymbol{\theta}$  is computed in a manner similar to the way it was for standard logistic regression. A bit of math shows that

$$\begin{aligned} \ell(\boldsymbol{\theta} | D) &= - \sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log(p_j(\mathbf{x}_i)) \\ &= - \sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log \left( \frac{e^{\mathbf{x}_i^\top \boldsymbol{\beta}_j}}{\sum_{m=1}^k e^{\mathbf{x}_i^\top \boldsymbol{\beta}_m}} \right) \end{aligned}$$

where

$$\delta_{c_j}(y_i) = \begin{cases} 1 & \text{if } y_i = c_j, \text{ the jth class} \\ 0 & \text{otherwise.} \end{cases}$$

This is a convex minimization problem with unique minimizer  $\hat{\theta}$ . Once  $\hat{\theta} = (\hat{\beta}_1, \dots, \hat{\beta}_k)$  is found, we have an estimate for the probability

$$P(Y = y \mid \mathbf{x}) \approx \frac{e^{\mathbf{x}^\top \hat{\beta}_y}}{\sum_{j=1}^k e^{\mathbf{x}^\top \hat{\beta}_j}}.$$

From this, we can construct a classifier  $\hat{f}$  by setting  $\hat{f}(\mathbf{x}) = \operatorname{argmax}_j P(Y = c_j \mid \mathbf{x})$ .

Conveniently, `sklearn` has a very simple implementation of multinomial logistic regression that simply requires the argument `multi_class='multinomial'` when initiating a `LogisticRegression` model.

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

>>> model = LogisticRegression(
        multi_class='multinomial',
        fit_intercept=True).fit(X, y) # add constant feature
```

**Problem 5.** The Iris Dataset contains information taken from 150 samples of 3 different types of iris flowers (Setosa, Versicolor, and Virginica). The columns contain measurements for sepal length, sepal width, pedal length, and pedal width. Import the Iris Dataset and perform a train-test split on only the first two columns of the data with `test_size=0.4`. Train a multinomial logistic regression model using the training data with an added constant feature, and generate prediction labels for the test data. Plot the test data by color using your prediction labels. Also, print the model score. Reference code below for accessing the Iris Dataset and for using `train_test_split`.

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split

>>> iris = datasets.load_iris()

>>> X = iris.data[:, :2] # we only take the first two features.
>>> y = iris.target

>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

Your plot should reflect Figure 13.3

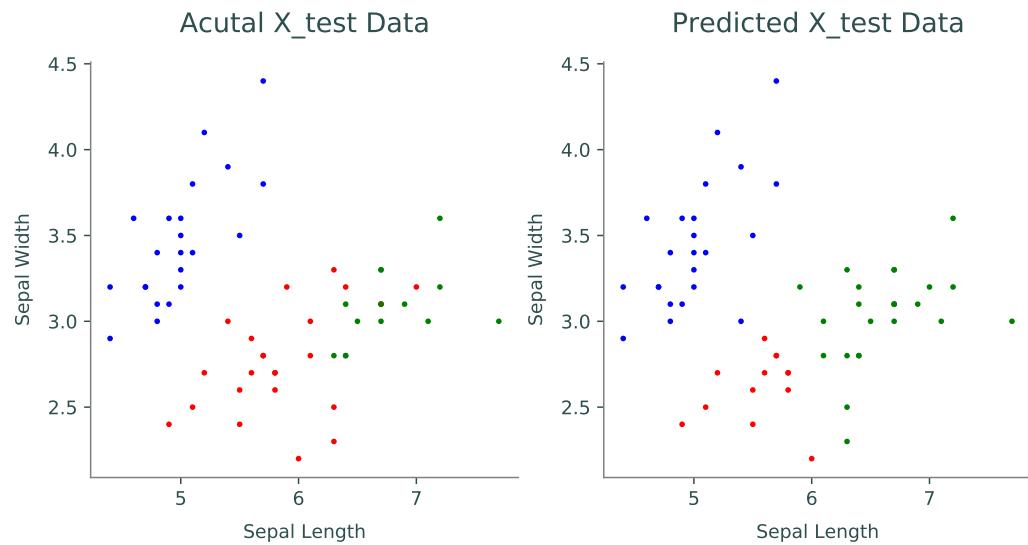


Figure 13.3: Multinomial logistic regression attempt to categorize the Iris Dataset.

# 14

## Naive Bayes

**Lab Objective:** *In this lab, we will create a Naïve Bayes Classifier and use it to make an SMS spam filter.*

### Naïve Bayes Classifiers

Naïve Bayes classifiers are a family of machine learning classification methods that use Bayes' theorem to probabilistically categorize data. They are called naïve because they assume independence between the features. The main idea is to use Bayes' theorem to determine the probability that a certain data point belongs in a certain class, given the features of that data. Despite what the name may suggest, the naïve Bayes classifier is not a Bayesian method, as it is based on likelihood rather than Bayesian inference.

While naïve Bayes classifiers are most easily seen as applicable in cases where the features have, ostensibly, well defined probability distributions (such as classifying sex given physical characteristics), they are applicable in many other cases. In this lab, we will apply them to the problem of spam filtering. While it is generally a bad idea to assume independence, naïve Bayes classifiers can still be very effective, even when we can be confident that features are not independent.

Given the feature vector of a piece of data we want to classify, we want to know which of all the classes is most likely. Essentially, we want to answer the following question

$$\operatorname{argmax}_{k \in K} P(C = k | \mathbf{x}), \quad (14.1)$$

where  $C$  is the random variable representing the class of the data. Using Bayes' Theorem, we can reformulate this problem into something that is actually computable. For any  $k \in K$ ,

$$P(C = k | \mathbf{x}) = \frac{P(C = k)P(\mathbf{x} | C = k)}{P(\mathbf{x})}.$$

Now we will examine each feature individually and use the chain rule to expand the new conditional probability:

$$\begin{aligned} P(\mathbf{x} | C = k) &= P(x_1, \dots, x_n | C = k) \\ &= P(x_1 | x_2, \dots, x_n, C = k)P(x_2, \dots, x_n | C = k) \\ &= \dots \\ &= P(x_1 | x_2, \dots, x_n, C = k)P(x_2 | x_3, \dots, x_n, C = k) \cdots P(x_n | C = k). \end{aligned}$$

By applying the assumption that each feature is independent we can drastically simplify this expression to the following:

$$P(x_1 | x_2, \dots, x_n, C = k) \cdots P(x_n | C = k) = \prod_{i=1}^n P(x_i | C = k).$$

Therefore we have that

$$P(C = k | \mathbf{x}) = \frac{P(C = k)}{P(\mathbf{x})} \prod_{i=1}^n P(x_i | C = k),$$

which reforms Equation 14.1 as

$$\operatorname{argmax}_{k \in K} P(C = k | \mathbf{x}) = \operatorname{argmax}_{k \in K} P(C = k) \prod_{i=1}^n P(x_i | C = k). \quad (14.2)$$

We can drop the  $P(\mathbf{x})$  in the denominator since it does not depend on  $k$ . In this form, the problem is computationally tractable, since we can use the training data to find approximations of  $P(C = k)$  and  $P(x_i | C = k)$  for each  $i$  and  $k$ . Something to note here is that we are actually maximizing  $P(C = k | \mathbf{x})$  by computing and maximizing  $P(C = k, \mathbf{x})$ . This means that naïve Bayes is a generative classifier, and not a discriminative classifier.

## Spam Filters

A spam filter is just a special case of a classifier with two classes: spam and not spam (often called ham). Spam filtering is a situation where naive Bayes classifiers perform relatively well. Throughout the lab, we will use the SMS spam dataset contained in `sms_spam_collection.csv`. The messages in this dataset have already been cleaned by converting to lowercase and removing all punctuation. To load the dataset, use the following code:

```
import pandas as pd
df = pd.read_csv("sms_spam_collection.csv")

# separate the data into the messages and labels
X = df.Message
y = df.Label
```

Before we can construct a naive Bayes classifier, we need to choose a probability distribution for the  $x_i$ . Two common choices are categorical distributions and Poisson distributions. We will first create a classifier using a categorical distribution. In this case, each feature  $x_i$  represents the  $i$ -th word of a message. The probability  $P(x_i | C = k)$  then represents the probability that a specific word in the message is the word that we observed, given that the category is  $k$ . For simplicity, we assume that these values do not change with respect to  $i$ , so the probability of observing a specific word is the same for every position in a message.

Suppose we have a labeled training dataset. To train the model, we need to just find values for  $P(C = k)$  and  $P(x_i | C = k)$ . In this case, a reasonably good choice is the maximum likelihood estimator, which in this case is just

$$P(C = k) = \frac{N_{\text{samples in } k}}{N_{\text{samples}}}$$

$$P(x_i | C = k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k}}{N_{\text{words in class } k}}$$

However, this choice leads to some issues. For example, if a certain word  $x_j$  occurs only in spam messages and never in ham messages in our training dataset, then our classifier will predict  $P(x_i | C = \text{ham}) = 0$  for any message that contains  $x_j$ . This is not desirable, but to make matters worse, the same situation could happen for both classes within a single message, leading our model to predict  $P(x_i | C = k) = 0$  for all classes. This makes our classifier unable to classify such samples. To circumvent this issue, we will use *Laplace add-one smoothing*, which consists of adding 1 to the numerator of  $P(x_i | C = k)$  and 2 to its denominator. So, the probabilities we will use are the following:

$$P(C = \text{spam}) = \frac{N_{\text{messages in spam}}}{N_{\text{samples}}}, \quad (14.3)$$

$$P(C = \text{ham}) = \frac{N_{\text{messages ham}}}{N_{\text{samples}}}, \quad (14.4)$$

$$P(x_i | C = k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k} + 1}{N_{\text{total words in class } k} + 2}. \quad (14.5)$$

The result of Laplace add-one smoothing is equivalent to the maximum likelihood estimators if a certain Bayesian prior is used for the probabilities. We don't use this for the  $P(C = k)$ , since it is not really needed for those probabilities and does not lead to any particular benefit. Lastly, note that the denominator in Equation 14.5 is not the number of unique words in class  $k$ , but the total number of occurrences of any word in class  $k$ .

**Problem 1.** Create a class `NaiveBayesFilter`. Implement a method `fit()` that accepts the training data `X` and the training labels `y`. In this method, compute the probabilities  $P(C = \text{spam})$  and  $P(C = \text{ham})$  as in Equations (14.3) and (14.4). Also compute the probabilities  $P(x_i | C = k)$  for each word in both the spam and ham classes, thereby training the model. Store these computed probabilities in dictionaries called `self.spam_probs` and `self.ham_probs`, where the key is the word and the value is the associated probability. Make sure to include an entry in each dictionary for every word that shows up in either class, even if there are no occurrences.

For example, `self.ham_probs["out"]` will give the computed value for  $P(x_i = \text{"out"} | C = \text{ham})$  value:

```
# Example model trained on the first 300 data points
>>> nb = NaiveBayesFilter()
>>> nb.fit(X[:300], y[:300])

# Check spam and ham probabilities of "out"
>>> nb.ham_probs["out"]
0.003147128245476003
>>> nb.spam_probs["out"]
0.004166666666666667
```

Hint: be sure you count the number of occurrences of a word, and not simply of a string. For example, when searching the string "find it in there" for the word "in", make sure you get 1 and not 2 (because of the "in" in "find"). The methods `pd.Series.str.split()` and `count()` may be helpful. When using `split()`, call it without any arguments, as otherwise you may accidentally add empty strings to your data.

## Predictions

Now that we have trained our model, we can predict the class of a message by calculating

$$P(C = k) \prod_{i=1}^n P(x_i | C = k)$$

for each class  $k$ , then choosing the class that maximizes this probability. As discussed above, this is equivalent to maximizing the probability  $P(C = k | \mathbf{x})$ ; however, be aware that we are not actually computing those. The probabilities we compute here will not sum to 1, since they are actually the values  $P(C = k, \mathbf{x})$ .

However, directly computing this probability as a product can lead to an issue: underflow. If  $\mathbf{x}$  is a particularly long message, then, since we are multiplying lots of numbers between 0 and 1, it is possible for the computed probability to *underflow*, or become too small to be machine representable with ordinary floating-point numbers. In this case the computed probability becomes 0. This is particularly problematic because if underflow happens for a sample for one class, it will likely also happen for all of the other classes, making such samples impossible to classify. To avoid this issue, we will work with the logarithm of the probability:

$$\ln P(\mathbf{x}, C = k) = \ln(P(C = k)) + \sum_{i=1}^n \ln(P(x_i | C = k)). \quad (14.6)$$

This has the same maximizer as before, so our predictions are unaffected, while also avoiding any issues with underflow.

**Problem 2.** Implement the `predict_proba()` method in your naïve Bayes classifier. It should take as an argument  $\mathbf{X}$ , the data that needs to be classified, and it will compute the log probabilities as given in Equation 14.6 for each message in  $\mathbf{X}$ . In the case we have some word  $x_u$  that is not found in the training set, use the value  $P(x_u | C = k) = \frac{1}{2}$  for both classes.

The method should return an  $(N, 2)$  array, where  $N$  is the length of  $\mathbf{X}$ , whose entries are the log probabilities of each message  $\mathbf{x}$  in  $\mathbf{X}$  belonging to each category. The first column corresponds to  $\ln P(\mathbf{x}, C = \text{ham})$ , and the second to  $\ln P(\mathbf{x}, C = \text{spam})$ .

Your code should produce the following output with the example from Problem 1:

```
>>> nb.predict_proba(X[800:805])
array([[ -30.8951931 , -35.42406687] ,
       [-108.85464069, -91.70332157] ,
      [-74.65014875, -88.71184709] ,
     [-164.94297917, -133.8497405 ] ,
    [-127.17743715, -101.32098062]])
```

Hint: The dataframe `X`'s index might not be in order or consecutive integers, so accessing its rows as `X[i]` may lead to errors later. Using `for row in X` or similar to iterate will work better.

**Problem 3.** Implement the method `predict()`. Accept as an argument `X`, the data to be classified, and return an array with the same shape holding the predicted class for each sample in `X`. The entries of this array should be strings "`spam`" and "`ham`". Use your method `predict_proba()` to compute the log probabilities of each class. In case of a tie, predict "`ham`".

Your code should produce the following output with the example from Problem 1:

```
>>> nb.predict(X[800:805])
array(['ham', 'spam', 'ham', 'spam', 'spam'], dtype=object)
```

**Problem 4.** We now test our spam filter. Use the `sklearn`'s `train_test_split` function with the default parameters to split the data into training and test sets. Train a `NaiveBayesFilter` on the train set, and have it predict the labels of each message in the test set. Compute the answer to the following questions:

- What proportion of the spam messages in the test set were correctly identified by the classifier?
- What proportion of the ham messages were incorrectly identified?

Return the answers to these questions as a tuple.

## The Poisson Model

Now that we've examined one way to constructing a naïve Bayes classifier, let us look at one more method. In the Poisson model we assume that each word in the vocabulary is Poisson random variable, occurring with potentially different frequencies among spam and ham messages. Because each of the messages is a different length, we can reparameterize the Poisson PMF to the following

$$P(n_i = x) = \frac{(rn)^x e^{-rn}}{x!} \quad (14.7)$$

where  $n_i$  is the number of times word  $i$  occurs in a message,  $n$  is the length of the message, and  $\lambda = rn$  is the classical Poisson rate. In this case  $r$  represents the number of events per unit time/space/etc. We will again use maximum likelihood estimation to find the values of  $r$  for each word and class.

## Training the Model

Similar to the other classifier that we made, training the model amounts to using the training data to determine how  $P(x_i | C = k)$  is computed, as well as computing  $P(C = k)$ . For the Poisson model we must find a value for  $r$  for each word that appears in the training set. To do this we will use maximum likelihood estimation. We will label the chosen value of  $r$  for the  $i$ -th word and class  $k$  as  $r_{i,k}$ . In this case, since we are using a Poisson distribution (14.7) for each word, we will solve the following problem for both the spam class and the ham classes

$$r_{i,k} = \operatorname{argmax}_{r \in [0,1]} \frac{(rN_k)^{N_{i,k}} e^{-rN_k}}{N_{i,k}!}, \quad (14.8)$$

where  $N_k = N_{\text{total words in class } k}$  is the total number of words in class  $k$  and  $N_{i,k} = N_{\text{occurrences of word } i \text{ in class } k}$  is the number of times the  $i$ -th word occurs in class  $k$ . We have  $r \in [0, 1]$  because a word cannot occur more than once per word in the message. It can then be shown that the maximizing value is

$$r_{i,k} = N_{\text{occurrences of word } i \text{ in class } k} / N_{\text{total words in class } k}.$$

However, observe that in Equation (14.7), if we have  $r = 0$  then  $P(n_i = x) = 0$  whenever  $x > 0$ . This leads to the exact same issue we saw with the categorical approach, and will lead to predicted probabilities being 0. To resolve this issue, we will again use Laplace add-one smoothing and instead use the values

$$r_{i,k} = \frac{N_{\text{occurrences of word } i \text{ in class } k} + 1}{N_{\text{total words in class } k} + 2}. \quad (14.9)$$

As before, this can be interpreted as the maximum likelihood estimator if we start with a certain Bayesian prior for  $r$ .<sup>1</sup>

Making predictions with this model is exactly the same as we did earlier, albeit with slightly different equations. Specifically, if we write Equation 14.6 with the Poisson probability, our prediction is given by

$$\operatorname{argmax}_{k \in K} \ln(P(C = k)) + \sum_{i \in \text{Vocab}} \ln \left( \frac{(r_{i,k}n)^{n_i} e^{-r_{i,k}n}}{n_i!} \right), \quad (14.10)$$

with  $n_i$  the number of times the  $i^{\text{th}}$  word occurs in the message,  $n$  the total number of words in the message, and  $r_{i,k}$  the Poisson rate of the  $i^{\text{th}}$  word in class  $k$ .

**Problem 5.** Create a new class called `PoissonBayesFilter`, with three methods called `fit()`, `predict_proba()`, and `predict()`, analogous to those in the `NaiveBayesFilter` class.

Implement `fit()` by finding the MLE found in Equation 14.9 to predict  $r$  for each word in both the spam and ham classes, thereby training the model. Store these computed rates in dictionaries called `self.spam_rates` and `self.ham_rates`, where the key is the word and the value is the associated  $r$ .

For example, `self.ham_rates["in"]` will give the computed  $r$  value for the word "in" found in ham messages.

```
# Example model trained on the first 300 data points
>>> pb = PoissonBayesFilter()
>>> pb.fit(X[:300], y[:300])

# Check spam and ham rate of 'in'
```

<sup>1</sup>Note that these rates are exactly the same as the probabilities we computed for the categorical model.

```
>>> pb.ham_rates["in"]
0.012588512981904013
>>> pb.spam_rates["in"]
0.0041666666666666667
```

Implement the `predict_proba()` and `predict()` methods using equation 14.10. These methods will take the same arguments and return the same object types as the methods of the same name in the `NaiveBayesFilter` class. If a word  $u$  not in the training set is in one of the messages, use the value  $r_{u,k} = 1/(N_k + 2)$ . In case of a tie in the probabilities of two classes, predict "ham". Your code should produce the following output with the example above:

```
>>> pb.predict_proba(X[800:805])
array([[ -37.14113097,  -38.2193235 ],
       [-112.61977379,  -83.54702923],
       [-55.70966168,  -63.83602125],
       [-130.02471282,  -90.15687624],
       [-102.36539804,  -69.55261684]])

>>> pb.predict(X[800:805])
array(['ham', 'spam', 'ham', 'spam', 'spam'], dtype=object)
```

Hint: Most of your code will be very similar to your `NaiveBayesFilter` class. The function `np.unique` with the argument `return_counts=True` and the function `scipy.stats.poisson.logpmf` will be useful for `predict_proba()`.

**Problem 6.** In the function `prob6()`, use the `sklearn.model_selection.train_test_split` function to split the data into training and test sets. Train a `PoissonBayesClassifier` on the train set, and have it predict the labels of each message in the test set. Compute the answer to the following questions:

- What proportion of the spam messages in the test set were correctly identified by the classifier?
- What proportion of the ham messages were incorrectly identified?

Return the answers to these two questions as a tuple.

How do the performances of the categorical and Poisson models compare?

## Naive Bayes with Sklearn

Now that we've explored a few ways to implement our own naïve Bayes classifier, we can examine some robust tools from the `sklearn` library that will accomplish all the things we've coded up in a very simple manner.

The first thing we need to do is create a dictionary and transform the training data, which is what our first `fit()` method did. We instantiate a `CountVectorizer` model from `sklearn.feature_extraction.text`, and then use the `fit_transform()` method to create the dictionary and transform the training data.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
train_counts = vectorizer.fit_transform(X_train)
```

Now we can use the transformed training data to fit a `MultinomialNB` model from `sklearn.naive_bayes`

```
.
```

```
from sklearn.naive_bayes import MultinomialNB

clf = MultinomialNB()
clf = clf.fit(train_counts, y_train)
```

Testing data we want to classify must first be transformed by our vectorizer with the `transform()` method (not the `fit_transform()` method). We can then classify the data using the `predict()` method of the `MultinomialNB` model.

```
test_counts = vectorizer.transform(X_test)
labels = clf.predict(test_counts)
```

This naïve Bayes model uses the multinomial distribution where we have used the categorical and poisson distributions. Multinomial is very similar to the categorical implementation, as the multinomial distribution models the outcome of  $n$  categorical trials (in the same way that the binomial distribution models  $n$  Bernoulli trials).

**Problem 7.** Write a function that will classify messages using the SkLearn naive Bayes implementation. It will take as arguments training data `X_train` and `y_train`, and test data `X_test`. In this function use the `CountVectorizer` and `MultinomialNB` from SkLearn and return the predicted classification of the model.

# 15 Apache Spark

**Lab Objective:** *Dealing with massive amounts of data often requires parallelization and cluster computing; Apache Spark is an industry standard for doing just that. In this lab we introduce the basics of PySpark, Spark's Python API, including data structures, syntax, and use cases. Finally, we conclude with a brief introduction to the Spark Machine Learning Package.*

## Apache Spark

Apache Spark is an open-source, general-purpose distributed computing system used for big data analytics. Spark is able to complete jobs substantially faster than previous big data tools (i.e. Apache Hadoop) because of its in-memory caching, and optimized query execution. Spark provides development APIs in Python, Java, Scala, and R. On top of the main computing framework, Spark provides machine learning, SQL, graph analysis, and streaming libraries.

Spark's Python API can be accessed through the PySpark package. You must install Spark, along with the supporting tools like Java, on your local machine for PySpark to work. This will include ensuring that both Java and Spark are included in the environment variable PATH.<sup>1</sup> Installation of PySpark for local execution or remote connection to an existing cluster can be accomplished by running

```
$ pip install pyspark <= 3.4.1
```

To install Java on WSL or another Linux system, run the following commands in the terminal:

```
$ sudo apt-get install openjdk-8-jdk
```

For Mac M1 users, installing Java will be a little trickier.<sup>2</sup> With Homebrew installed, you'll need to install Maven by running

```
$ brew install maven
```

You will then navigate to [this link](#) to download the JDK 8 .dmg file. Enter the following details if the link doesn't automatically populate them for you:

<sup>1</sup>See the Apache Spark configuration instructions for detailed installation instructions

<sup>2</sup>Full instructions can be found at <https://dev.to/shane/configure-m1-mac-to-use-jdk8-with-maven-4b4g>

```
Java Version - Java 8 (LTS)
Operating System - macOS
Architecture - ARM 64-bit
Java Package - JDK
```

Then install the .dmg file. Now, add the `JAVA_HOME` environment variable to the `~/.zshrc` file (with vim or some other file editor) by adding the following line to the file:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/jre
```

Restart your terminal, and you should be all set. If this process takes longer than a few minutes, you should stop and finish this lab on one of the lab machines.

## PySpark

One major benefit of using PySpark is the ability to run it in an interactive environment. One such option is the interactive Spark shell that comes prepackaged with PySpark. To use the shell, simply run `pyspark` in the terminal. In the Spark shell you can run code one line at a time without the need to have a fully written program. This is a great way to get a feel for Spark. To get help with a function use `help(function)`; to exit the shell simply run `quit()`.

In the interactive shell, the `SparkSession` object - the main entrypoint to all Spark functionality - is available by default as `spark`. When running Spark in a standard Python script (or in IPython) you need to define this object explicitly. The code box below outlines how to do this. It is standard practice to name your `SparkSession` object `spark`.

### ACHTUNG!

It is important that when you are finished with a `SparkSession` you should end it by calling `spark.stop()`. For this lab, in each problem you will need to instantiate a new `SparkSession` in each problem and then end the session at the end of the function using `spark.stop()`.

### NOTE

While the interactive shell is very robust, it may be easier to learn Spark in an environment that you are more familiar with (like IPython). To do so, just use the code given below. Help can be accessed in the usual way for your environment. Just remember to `stop()` the `SparkSession`!

```
>>> from pyspark.sql import SparkSession

# instantiate your SparkSession object
>>> spark = SparkSession\
...     .builder\
...     .appName("app_name")\
...     .getOrCreate()
```

```
# stop your SparkSession
>>> spark.stop()
```

### NOTE

The syntax

```
>>> spark = SparkSession\
...     .builder\
...     .appName("app_name")\
...     .getOrCreate()
```

is somewhat unusual. While this code can be written on a single line, it is often more readable to break it up when dealing with many chained operations; this is standard styling for Spark. Note that you *cannot* write a comment after a line continuation character '`\`'.

## Resilient Distributed Datasets

The most fundamental data structure used in Apache Spark is the Resilient Distributed Dataset (RDD). RDDs are immutable distributed collections of objects. They are *resilient* because performing an operation on one RDD produces a *new* RDD without altering the original; if something goes wrong, you can always go back to your original RDD and restart. They are *distributed* because the data resides in logical partitions across multiple machines. While RDDs can be difficult to work with, they offer the most granular control of all the Spark data structures.

There are two main ways of creating RDDs. The first is reading a file directly into Spark and the second is parallelizing an existing collection (list, numpy array, pandas dataframe, etc.). We will use the Titanic dataset<sup>3</sup> in most of the examples throughout this lab. The example below shows various ways to load the Titanic dataset as an RDD.

```
# initialize your SparkSession object
>>> spark = SparkSession\
...     .builder\
...     .appName("app_name")\
...     .getOrCreate()

# load the data directly into an RDD
>>> titanic = spark.sparkContext.textFile("titanic.csv")

# the file is of the format
# Pclass,Survived,Name,Sex,Age,Sibsp,Parch,Ticket,Fare
# Survived | Class | Name | Sex | Age | Siblings/Spouses Aboard | Parents/←
# Children Aboard | Fare
```

<sup>3</sup><https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html>

```
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# note that each element is a single string - not particularly useful
# one option is to first load the data into a numpy array
>>> np_titanic = np.loadtxt("titanic.csv", delimiter=',', dtype=list)

# use sparkContext to parallelize the data into 4 partitions
>>> titanic_parallelize = spark.sparkContext.parallelize(np_titanic, 4)

>>> titanic_parallelize.take(2)
[array(['0', '3', ..., 'male', '22', '1', '0', '7.25'], dtype=object),
 array(['1', '1', ..., 'female', '38', '1', '0', '71.283'], dtype=object)]

# end SparkSession
>>> spark.stop()
```

### ACHTUNG!

Because Apache Spark partitions and distributes data, calling for the first  $n$  objects using the same code (such as `take(n)`) may yield different results on different computers (or even each time you run it on one computer). This is not something you should worry about; it is the result of variation in partitioning and will not affect data analysis.

## RDD Operations

### Transformations

There are two types of operations you can perform on RDDs: *transformations* and *actions*. Transformations are functions that produce new RDDs from existing ones. Transformations are also lazy; they are not executed until an *action* is performed. This allows Spark to boost performance by optimizing *how* a sequence of transformations is executed at runtime.

One of the most commonly used transformations is the `map(func)`, which creates a new RDD by applying `func` to each element of the current RDD. This function, `func`, can be any callable python function, though it is often implemented as a `lambda` function. Similarly, `flatMap(func)` creates an RDD with the flattened results of `map(func)`.

```
# initialize your SparkSession object
>>> spark = SparkSession\
...         .builder\
...         .appName("app_name")\
...         .getOrCreate()

# use map() to format the data
>>> titanic = spark.sparkContext.textFile("titanic.csv")
>>> titanic.take(2)
```

```
[ '0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
  '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# apply split(',') to each element of the RDD with map()
>>> titanic.map(lambda row: row.split(','))\
...     .take(2)
[['0', '3', 'Mr. Owen Harris Braund', 'male', '22', '1', '0', '7.25'],
 ['1', '1', ..., 'female', '38', '1', '0', '71.283']]

# compare to flatMap(), which flattens the results of each row
>>> titanic.flatMap(lambda row: row.split(','))\
...     .take(2)
['0', '3']
```

The `filter(func)` transformation returns a new RDD containing only the elements that satisfy `func`. In this case, `func` should be a callable python function that returns a Boolean. The elements of the RDD that evaluate to `True` are included in the new RDD while those that evaluate to `False` are excluded.

```
# create a new RDD containing only the female passengers
>>> titanic = titanic.map(lambda row: row.split(','))
>>> titanic_f = titanic.filter(lambda row: row[3] == "female")
>>> titanic_f.take(3)
[['1', '1', ..., 'female', '38', '1', '0', '71.283'],
 ['1', '3', ..., 'female', '26', '0', '0', '7.925'],
 ['1', '1', ..., 'female', '35', '1', '0', '53.1']]
```

### NOTE

A great transformation to help validate or explore your dataset is `distinct()`. This will return a new RDD containing only the distinct elements of the original. In the case of the Titanic dataset, if you did not know how many classes there were, you could do the following:

```
>>> titanic.map(lambda row: row[1])\
...     .distinct()\
...     .collect()
['1', '3', '2']
```

Spark Command	Transformation
<code>map(f)</code>	Returns a new RDD by applying <code>f</code> to each element of this RDD
<code>flatMap(f)</code>	Same as <code>map(f)</code> , except the results are flattened
<code>filter(f)</code>	Returns a new RDD containing only the elements that satisfy <code>f</code>
<code>distinct()</code>	Returns a new RDD containing the distinct elements of the original
<code>reduceByKey(f)</code>	Takes an RDD of <code>(key, val)</code> pairs and merges the values for each <code>key</code> using an associative and commutative reduce function <code>f</code>
<code>sortBy(f)</code>	Sorts this RDD by the given function <code>f</code>
<code>sortByKey(f)</code>	Sorts an RDD assumed to consist of <code>(key, val)</code> pairs by the given function <code>f</code>
<code>groupByKey(f)</code>	Returns a new RDD of groups of items based on <code>f</code>
<code>groupByKey()</code>	Takes an RDD of <code>(key, val)</code> pairs and returns a new RDD with <code>(key, (val1, val2, ...))</code> pairs

```
# the following counts the number of passengers in each class
# note that this isn't necessarily the best way to do this

# create a new RDD of (pclass, 1) elements to count occurrences
>>> pclass = titanic.map(lambda row: (row[1], 1))
>>> pclass.take(5)
[('3', 1), ('1', 1), ('3', 1), ('1', 1), ('3', 1)]

# count the members of each class
>>> pclass = pclass.reduceByKey(lambda x, y: x + y)
>>> pclass.collect()
[('3', 487), ('1', 216), ('2', 184)]

# sort by number of passengers in each class, ascending order
>>> pclass.sortBy(lambda row: row[1]).collect()
[('2', 184), ('1', 216), ('3', 487)]

# end SparkSession
>>> spark.stop()
```

### ACHTUNG!

Note that you must use `.collect()` to extract data from an RDD. Using `.collect()` will return an array.

**Problem 1.** Write a function that accepts the name of a text file with default `filename=huck_finn.txt`.<sup>a</sup> Load the file as a PySpark RDD, and count the number of occurrences of each word. Sort the words by count, in descending order, and return a list of the `(word, count)` pairs for the 20 most used words. The data does not need to be cleaned.

Hint: to ensure that your function doesn't consider an empty string `" "` to be a word, make sure to split each line with `.split()` instead of on a space with `.split(' ')`.

<sup>a</sup><https://www.gutenberg.org/files/76/76-0.txt>

## Actions

Actions are operations that return non-RDD objects. Two of the most common actions, `take(n)` and `collect()`, have already been seen above. The key difference between the two is that `take(n)` returns the first `n` elements from one (or more) partition(s) while `collect()` returns the contents of the entire RDD. When working with small datasets this may not be an issue, but for larger datasets running `collect()` can be very expensive.

Another important action is `reduce(func)`. Generally, `reduce()` combines (reduces) the data in each row of the RDD using `func` to produce some useful output. Note that `func` *must* be an associative and commutative binary operation; otherwise the results will vary depending on partitioning.

```
# create an RDD with the first million integers in 4 partitions
>>> ints = spark.sparkContext.parallelize(range(1, 1000001), 4)
# [1, 2, 3, 4, 5, ..., 1000000]
# sum the first one million integers
>>> ints.reduce(lambda x, y: x + y)
500000500000

# create a new RDD containing only survival data
>>> survived = titanic.map(lambda row: int(row[0]))
>>> survived.take(5)
[0, 1, 1, 1, 0]

# find total number of survivors
>>> survived.reduce(lambda x, y: x + y)
500
```

Spark Command	Action
<code>take(n)</code>	returns the first $n$ elements of an RDD
<code>collect()</code>	returns the entire contents of an RDD
<code>reduce(f)</code>	merges the values of an RDD using an associative and commutative operator $f$
<code>count()</code>	returns the number of elements in the RDD
<code>min(); max(); mean()</code>	returns the minimum, maximum, or mean of the RDD, respectively
<code>sum()</code>	adds the elements in the RDD and returns the result
<code>saveAsTextFile(path)</code>	saves the RDD as a collection of text files (one for each partition) in the directory specified
<code>foreach(f)</code>	immediately applies $f$ to each element of the RDD; not to be confused with <code>map()</code> , <code>foreach()</code> is useful for saving data somewhere not natively supported by PySpark

**Problem 2.** Since the area of a circle of radius  $r$  is  $A = \pi r^2$ , one way to estimate  $\pi$  is to estimate the area of the unit circle. A Monte Carlo approach to this problem is to uniformly sample points in the square  $[-1, 1] \times [-1, 1]$  and then count the percentage of points that land within the unit circle. The percentage of points within the circle approximates the percentage of the area occupied by the circle. Multiplying this percentage by 4 (the area of the square  $[-1, 1] \times [-1, 1]$ ) gives an estimate for the area of the circle.<sup>a</sup>

Write a function that uses Monte Carlo methods to estimate the value of  $\pi$ . Your function should accept two keyword arguments: `n=10**5` and `parts=6`. Use `n*parts` sample points and partition your RDD with `parts` partitions. Return your estimate.

<sup>a</sup>See Example 7.1.1 in the Volume 2 textbook

## DataFrames

While RDDs offer granular control, they can be slower than their Scala and Java counterparts when implemented in Python. The solution to this was the creation of a new data structure: Spark DataFrames. Just like RDDs, DataFrames are immutable distributed collections of objects; however, unlike RDDs, DataFrames are organized into named (and typed) columns. In this way they are conceptually similar to a relational database (or a pandas DataFrame).

The most important difference between a relational database and Spark DataFrames is in the execution of transformations and actions. When working with DataFrames, Spark's Catalyst Optimizer creates and optimizes a logical execution plan before sending any instructions to the drivers. After the logical plan has been formed, an optimal physical plan is created and executed. This provides significant performance boosts, especially when working with massive amounts of data. Since the Catalyst Optimizer functions the same across all language APIs, DataFrames bring performance parity to all of Spark's APIs.

## Spark SQL and DataFrames

Creating a DataFrame from an existing text, csv, or JSON file is generally easier than creating an RDD. The DataFrame API also has arguments to deal with file headers or to automatically infer the schema.

```
# note that you should initialize your spark object first
# load the titanic dataset using default settings
>>> titanic = spark.read.csv("titanic.csv")
>>> titanic.show(2)
+---+---+-----+---+---+---+---+
|_c0|_c1|      _c2| _c3|_c4|_c5|_c6|      _c7|
+---+---+-----+---+---+---+---+-----+
|  0|  3|Mr. Owen Harris B...| male| 22|   1|   0|    7.25|
|  1|  1|Mrs. John Bradley...|female| 38|   1|   0| 71.2833|
+---+---+-----+---+---+---+---+-----+
only showing top 2 rows

# spark.read.csv("titanic.csv", inferSchema=True) will try to infer
# data types for each column

# load the titanic dataset specifying the schema
>>> schema = ("survived INT, pclass INT, name STRING, sex STRING, "
...             "age FLOAT, sibsp INT, parch INT, fare FLOAT"
...         )
>>> titanic = spark.read.csv("titanic.csv", schema=schema)
>>> titanic.show(2)
+-----+-----+-----+-----+-----+-----+
|survived|pclass|          name| sex|age|sibsp|parch|  fare|
+-----+-----+-----+-----+-----+-----+-----+
|       0|     3|Mr. Owen Harris B...| male| 22|   1|   0|    7.25|
|       1|     1|Mrs. John Bradley...|female| 38|   1|   0| 71.2833|
+-----+-----+-----+-----+-----+-----+
only showing top 2 rows

# for files with headers, the following is convenient
>>> spark.read.csv("my_file.csv", header=True, inferSchema=True)
```

### NOTE

To convert a DataFrame to an RDD use `my_df.rdd`; to convert an RDD to a DataFrame use `spark.createDataFrame(my_rdd)`. You can also use `spark.createDataFrame()` on numpy arrays and pandas DataFrames.

DataFrames can be easily updated, queried, and analyzed using SQL operations. Spark allows you to run queries directly on DataFrames similar to how you perform transformations on RDDs. Additionally, the `pyspark.sql.functions` module contains many additional functions to further analysis. Below are many examples of basic DataFrame operations; further examples involving the `pyspark.sql.functions` module can be found in the additional materials section. Full documentation can be found at <https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html>.

```
# select data from the survived column
>>> titanic.select(titanic.survived).show(3) # or titanic.select("survived")
+-----+
|survived|
+-----+
|     0|
|     1|
|     1|
+-----+
only showing top 3 rows

# find all distinct ages of passengers (great for data exploration)
>>> titanic.select("age")\
...      .distinct()\
...      .show(3)
+---+
| age|
+---+
|18.0|
|64.0|
|0.42|
+---+
only showing top 3 rows

# filter the DataFrame for passengers between 20-30 years old (inclusive)
>>> titanic.filter(titanic.age.between(20, 30)).show(3)
+-----+-----+-----+-----+-----+-----+
|survived|pclass|          name| sex| age|sibsp|parch| fare|
+-----+-----+-----+-----+-----+-----+
|     0|     3|Mr. Owen Harris B...| male|22.0|   1|  0| 7.25|
|     1|     3|Miss. Laina Heikk...|female|26.0|   0|  0| 7.925|
|     0|     3| Mr. James Moran| male|27.0|   0|  0|8.4583|
+-----+-----+-----+-----+-----+-----+
only showing top 3 rows

# find total fare by pclass (or use .avg("fare") for an average)
>>> titanic.groupBy("pclass")\
...      .sum("fare")\
...      .show()
+-----+-----+
|pclass|sum(fare)|
+-----+-----+
```

```

|      1| 18177.41|
|      3|  6675.65|
|      2|  3801.84|
+----+-----+


# group and count by age and survival; order age/survival descending
>>> titanic.groupBy("age", "survived").count()\
...         .sort("age", "survived", ascending=False) \
...         .show(2)
+---+-----+-----+
|age|survived|count|
+---+-----+-----+
| 80|        1|     1|
| 74|        0|     1|
+---+-----+
only showing top 2 rows

# join two DataFrames on a specified column (or list of columns)
>>> titanic_cabins.show(3)
+-----+-----+
|           name|  cabin|
+-----+-----+
|Miss. Elisabeth W...|      B5|
|Master. Hudson Tr...|C22 C26|
|Miss. Helen Lorai...|C22 C26|
+-----+-----+
only showing top 3 rows

>>> titanic.join(titanic_cabins, on="name").show(3)
+-----+-----+-----+-----+-----+-----+-----+
|           name|survived|pclass|   sex|  age|sibsp|parch|  fare|  cabin|
+-----+-----+-----+-----+-----+-----+-----+
|Miss. Elisabeth W...|      0|     3| male|22.0|     1|     0|  7.25|      B5|
|Master. Hudson Tr...|      1|     3|female|26.0|     0|     0| 7.925|C22 C26|
|Miss. Helen Lorai...|      0|     3| male|27.0|     0|     0| 8.4583|C22 C26|
+-----+-----+-----+-----+-----+-----+-----+
only showing top 3 rows

```

### NOTE

If you prefer to use traditional SQL syntax you can use `spark.sql("SQL QUERY")`. Note that this requires you to first create a temporary view of the DataFrame.

```

# create the temporary view so we can access the table through SQL
>>> titanic.createOrReplaceTempView("titanic")

```

```
# query using SQL syntax
>>> spark.sql("SELECT age, COUNT(*) AS count\
...           FROM titanic\
...           GROUP BY age\
...           ORDER BY age DESC").show(3)
+-----+
|age|count|
+---+---+
| 80|    1|
| 74|    1|
| 71|    2|
+---+---+
only showing top 3 rows
```

Spark SQL Command	SQLite Command
select(*cols)	SELECT
groupBy(*cols)	GROUP BY
sort(*cols, **kwargs)	ORDER BY
filter(condition)	WHERE
when(condition, value)	WHEN
between(lowerBound, upperBound)	BETWEEN
drop(*cols)	DROP
join(other, on=None, how=None)	JOIN (join type specified by how)
count()	COUNT()
sum(*cols)	SUM()
avg(*cols) or mean(*cols)	AVG()
collect()	fetchall()

**Problem 3.** Write a function with keyword argument `filename="titanic.csv"`. Load the file into a PySpark DataFrame and find (1) the number of women on-board, (2) the number of men on-board, (3) the survival rate of women, and (4) the survival rate of men. Return these four values in the order given as a tuple of floats.

**Problem 4.** In this problem, you will be using the `london_income_by_borough.csv` and the `london_crime_by_lsoa.csv` files to visualize the relationship between income and the frequency of crime.<sup>a</sup> The former contains estimated mean and median income data for each London borough, averaged over 2008-2016; the first line of the file is a header with columns `borough`, `mean-08-16`, and `median-08-16`. The latter contains over 13 million lines of crime data, organized by borough and LSOA (Lower Super Output Area) code, for London between 2008 and 2016; the first line of the file is a header, containing the following seven columns:

`lsoa_code`: LSOA code (think area code) where the crime was committed  
`borough`: London borough where the crime was committed  
`major_category`: major or general category of the crime  
`minor_category`: minor or specific category of the crime  
`value`: number of occurrences of this crime in the given `lsoa_code`, `month`, and `year`  
`year`: year the crime was committed  
`month`: month the crime was committed

Write a function that accepts three keyword arguments:

`crimefile="london_crime_by_lsoa.csv", incomefile="london_income_by_borough.csv"`, and `major_cat="Robbery"`. Load the two files as PySpark DataFrames. Use them to create a new DataFrame. The new DataFrame will contain a row for each borough and have columns for borough, total number of crimes for the given major category (`major_cat`), and median income. Order the DataFrame by the total number of crimes for `major_cat`, descending. The final DataFrame should have three columns: `borough`, `major_cat_total_crime`, and `median -08-16` (column names may be different).

Convert the DataFrame to a numpy array using `np.array(df.collect())`, and create a scatter plot of the number of `major_cat=` crimes by the median income for each borough. Return the numpy array.

<sup>a</sup>data.london.gov.uk

## Machine Learning with Apache Spark

Apache Spark includes a vast and expanding ecosystem to perform machine learning. PySpark's primary machine learning API, `pyspark.ml`, is DataFrame-based.

Here we give a start to finish example using Spark ML to tackle the classic Titanic classification problem.

```
# prepare data
# convert the "sex" column to binary categorical variable
>>> from pyspark.ml.feature import StringIndexer, OneHotEncoder
>>> sex_binary = StringIndexer(inputCol="sex", outputCol="sex_binary")

# one-hot-encode pclass (Spark automatically drops a column)
>>> onehot = OneHotEncoder(inputCols=["pclass"],
...                           outputCols=["pclass_onehot"])

# create single features column
>>> from pyspark.ml.feature import VectorAssembler
>>> features = ["sex_binary", "pclass_onehot", "age", "sibsp", "parch", "fare"]
>>> features_col = VectorAssembler(inputCols=features, outputCol="features")

# now we create a transformation pipeline to apply the operations above
# this is very similar to the pipeline ecosystem in sklearn
>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=[sex_binary, onehot, features_col])
>>> titanic = pipeline.fit(titanic).transform(titanic)
```

```

# drop unnecessary columns for cleaner display (note the new columns)
>>> titanic = titanic.drop("pclass", "name", "sex")
>>> titanic.show(2)
+-----+-----+-----+-----+-----+-----+
|survived| age|sibsp|parch|fare|sex_binary|pclass_onehot|  features|
+-----+-----+-----+-----+-----+-----+-----+
|      0|22.0|     1|    0|7.25|      0.0| (3, [], [])|(8,[4,5...|
|      1|38.0|     1|    0|71.3|      1.0| (3,[1],...|[0.0,1....|
+-----+-----+-----+-----+-----+-----+-----+

# split into train/test sets (75/25)
>>> train, test = titanic.randomSplit([0.75, 0.25], seed=11)

# initialize logistic regression
>>> from pyspark.ml.classification import LogisticRegression
>>> lr = LogisticRegression(labelCol="survived", featuresCol="features")

# run a train-validation-split to fit best elastic net param
# ParamGridBuilder constructs a grid of parameters to search over.
>>> from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator as MCE
>>> paramGrid = ParamGridBuilder() \
...           .addGrid(lr.elasticNetParam, [0, 0.5, 1]).build()
# TrainValidationSplit will try all combinations and determine best model using
# the evaluator (see also CrossValidator)
>>> tvs = TrainValidationSplit(estimator=lr,
...                             estimatorParamMaps=paramGrid,
...                             evaluator=MCE(labelCol="survived"),
...                             trainRatio=0.75,
...                             seed=11)

# we train the classifier by fitting our tvs object to the training data
>>> clf = tvs.fit(train)

# use the best fit model to evaluate the test data
>>> results = clf.bestModel.evaluate(test)
>>> results.predictions.select(["survived", "prediction"]).show(5)
+-----+-----+
|survived|prediction|
+-----+-----+
|      0|      1.0|
|      0|      1.0|
|      0|      1.0|
|      0|      1.0|
|      0|      0.0|
+-----+-----+

# performance information is stored in various attributes of "results"

```

```

>>> results.accuracy
0.7527272727272727

>>> results.weightedRecall
0.7527272727272727

>>> results.weightedPrecision
0.751035147726004

# many classifiers do not have this object-oriented interface (yet)
# it isn't much more effort to generate the same statistics for a ←
    DecisionTreeClassifier, for example
>>> dt_clf = dt_tvs.fit(train) # same process, except for a different paramGrid

# generate predictions - this returns a new DataFrame
>>> preds = clf.bestModel.transform(test)
>>> preds.select("survived", "probability", "prediction").show(5)
+-----+-----+-----+
|survived|probability|prediction|
+-----+-----+-----+
|      0|   [1.0,0.0]|      0.0|
|      0|   [1.0,0.0]|      0.0|
|      0|   [1.0,0.0]|      0.0|
|      0|   [0.0,1.0]|      1.0|
+-----+-----+-----+

# initialize evaluator object
>>> dt_eval = MCE(labelCol="survived")
>>> dt_eval.evaluate(preds, {dt_eval.metricName: "accuracy"})
0.8433179723502304

```

Below is a broad overview of the `pyspark.ml` ecosystem. It should help give you a starting point when looking for a specific functionality.

PySpark ML Module	Module Purpose
<code>pyspark.ml.feature</code>	provides functions to transform data into feature vectors
<code>pyspark.ml.tuning</code>	grid search, cross validation, and train/validation split functions
<code>pyspark.ml.evaluation</code>	tools to compute prediction metrics (accuracy, f1, etc.)
<code>pyspark.ml.classification</code>	classification models (logistic regression, SVM, etc.)
<code>pyspark.ml.clustering</code>	clustering models (k-means, Gaussian mixture, etc.)
<code>pyspark.ml.regression</code>	regression models (linear regression, decision tree regressor, etc.)

**Problem 5.** Write a function with keyword argument `filename="titanic.csv"`. Load the file into a PySpark DataFrame, and use the `pyspark.ml` package to train a classifier that outperforms the logistic regression each of the three metrics from the example above (`accuracy`, `weightedRecall`, `weightedPrecision`).

Some of Spark's available classifiers are listed below. For complete documentation, visit <https://spark.apache.org/docs/latest/api/python/reference/pyspark.ml.html>.

```
# from pyspark.ml.classification import LinearSVC
#                                         DecisionTreeClassifier
#                                         GBTClassifier
#                                         MultilayerPerceptronClassifier
#                                         NaiveBayes
#                                         RandomForestClassifier
```

Use `randomSplit([0.75, 0.25], seed=11)` to split your data into train and test sets before fitting the model. Return the `accuracy`, `weightedRecall`, and `weightedPrecision` for your model, in the given order as a tuple.

Hint: to calculate the accuracy of a classifier in PySpark, use `results.accuracy` where `results = (classifier name).bestModel.evaluate(test)`.

## Additional Material

### Further DataFrame Operations

There are a few other functions built directly on top of DataFrames to further analysis. Additionally, the `pyspark.sql.functions` module expands the available functions significantly.<sup>4</sup>

```
# some immediately accessible functions
# covariance between pclass and fare
>>> titanic.cov("pclass", "fare")
-22.86289824115662

# summary of statistics for selected columns
>>> titanic.select("pclass", "age", "fare")\
...     .summary().show()
+-----+-----+-----+
|summary|      pclass|        age|       fare|
+-----+-----+-----+
|  count|      887|      887|      887|
|  mean| 2.305524239007892|29.471443066501564|32.305420253026846|
| stddev|0.8366620036697728|14.121908405492908| 49.78204096767521|
|   min|          1|      0.42|      0.0|
|  25%|          2|      20.0|      7.925|
|  50%|          3|      28.0|      14.4542|
|  75%|          3|      38.0|      31.275|
|   max|          3|      80.0|      512.3292|
+-----+-----+-----+

# additional functions from the functions module
>>> from pyspark.sql import functions as sqlf

# finding the mean of a column without grouping requires sqlf.avg()
# alias(new_name) allows us to rename the column on the fly
>>> titanic.select(sqlf.avg("age").alias("Average Age")).show()
+-----+
|    Average Age|
+-----+
|29.471443066516347|
+-----+

# use .agg([dict]) on GroupedData to specify [multiple] aggregate
# functions, including those from pyspark.sql.functions
>>> titanic.groupBy("pclass")\
...     .agg({"fare": "var_samp", "age": "stddev"})\
...     .show(3)
+-----+-----+
|pclass| var_samp(fare)|    stddev(age)|
+-----+-----+
```

---

<sup>4</sup><https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html>

```
| 1| 6143.483042924841|14.183632587264817|
| 3|139.64879027298073|12.095083834183779|
| 2|180.02658999396826|13.756191206499766|
+-----+-----+
# perform multiple aggregate actions on the same column
>>> titanic.groupBy("pclass")\
...     .agg(sqlf.sum("fare"), sqlf.stddev("fare"))\
...     .show()
+-----+-----+
|pclass|      sum(fare)| stddev_samp(fare) |
+-----+-----+
| 1|18177.412506103516| 78.38037409278448|
| 3| 6675.653553009033| 11.81730892686574|
| 2|3801.8417053222656|13.417398778972332|
+-----+-----+
```

pyspark.sql.functions	Operation
ceil(col)	computes the ceiling of each element in col
floor(col)	computes the floor of each element in col
min(col), max(col)	returns the minimum/maximum value of col
mean(col)	returns the average of the values of col
stddev(col)	returns the unbiased sample standard deviation of col
var_samp(col)	returns the unbiased variance of the values in col
rand(seed=None)	generates a random column with i.i.d. samples from [0, 1]
randn(seed=None)	generates a random column with i.i.d. samples from the standard normal distribution
exp(col)	computes the exponential of col
log(arg1, arg2=None)	returns arg1-based logarithm of arg2; if there is only one argument, then it returns the natural logarithm
cos(col), sin(col), etc.	computes the given trigonometric or inverse trigonometric (asin(col), etc.) function of col

# 16

# Parallel Programming with MPI

**Lab Objective:** *In the world of parallel computing, MPI is the most widespread and standardized message passing library. As such, it is used in the majority of parallel computing programs. In this lab, we explore and practice the basic principles and commands of MPI to further recognize when and how parallelization can occur.*

## MPI: the Message Passing Interface

At its most basic, the Message Passing Interface (MPI) provides functions for sending and receiving messages between different processes. MPI was developed to provide a standard framework for parallel computing in any language. It specifies a library of functions — the syntax and semantics of message passing routines — that can be called from programming languages such as Fortran and C.

MPI can be thought of as “the assembly language of parallel computing,” because of this generality.<sup>1</sup> MPI is important because it was the first portable and universally available standard for programming parallel systems and continues to be the de facto standard today.

### NOTE

Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are “parallel” programs and are typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed for any general architecture.

## Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial because it requires managing multiple processors and their interactions. Python, however, is an excellent language for simplifying algorithm design because it allows for problem solving without too much detail. Unfortunately, Python is not designed for high performance computing and is a notably slower scripted language. It is best practice to prototype in Python and then to write production code in fast compiled languages such as C or Fortran.

<sup>1</sup> *Parallel Programming with MPI*, by Peter S. Pacheco, pg. 7.

In this lab, we will explore the Python library `mpi4py` which retains most of the functionality of C implementations of MPI and is a good learning tool. There are three main differences to keep in mind between `mpi4py` and MPI in C:

- Python is array-based while C is not.
- `mpi4py` is object oriented but MPI in C is not.
- `mpi4py` supports two methods of communication to implement each of the basic MPI commands. They are the upper and lower case commands (e.g. `Bcast(...)` and `bcast(...)`). The uppercase implementations use traditional MPI datatypes while the lower case use Python's pickling method. Pickling offers extra convenience to using `mpi4py`, but the traditional method is faster. In these labs, we will only use the uppercase functions.

To install `mpi4py` on a Linux (or WSL) machine, you will need to execute the following commands:

```
$ sudo apt-get install libopenmpi-dev
$ pip install mpi4py
```

And to install `mpi4py` on a Mac, you will need to execute the following commands:

```
$ brew install openmpi
$ brew install mpi4py
```

## Using MPI

We will start with a Hello World program.

```
1 #hello.py
2 from mpi4py import MPI
3
4 COMM = MPI.COMM_WORLD
5 RANK = COMM.Get_rank()
6
7 print(f"Hello world! I'm process number {RANK}.")
```

examplecode/hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpiexec -n 5 python hello.py
```

### ACHTUNG!

Note that some systems may require `python3`, `py`, or a specific version like `python3.10` instead of `python` as shown above. Check the `python` environment that your IDE uses to ensure your code runs correctly.

**ACHTUNG!**

Some systems may not have enough slots to run the desired number of processes. In this case, the `--oversubscribe` flag must be used to ignore the number of available slots when running multiple processes. Therefore, the command will look like

```
$ mpiexec --oversubscribe -n 5 python hello.py
```

The program should output something like this:

```
Hello world! I'm process number 3.  
Hello world! I'm process number 2.  
Hello world! I'm process number 0.  
Hello world! I'm process number 4.  
Hello world! I'm process number 1.
```

Notice that when you try this on your own, the lines will not necessarily print in order. This is because there will be five separate processes running autonomously, and we cannot know beforehand which one will execute its `print()` statement first.

**ACHTUNG!**

It is usually bad practice to perform I/O (e.g., call `print()`) from any process besides the root process (rank 0), though it can be a useful tool for debugging.

How does this program work? First, the `mpiexec` program is launched. This is the program which starts MPI, a wrapper around whatever program you pass into it. The `-n 5` option specifies the desired number of processes. In our case, 5 processes are run, with each one being an instance of the program “python”. To each of the 5 instances of python, we pass the argument `hello.py` which is the name of our program’s text file, located in the current directory. Each of the five instances of python then opens the `hello.py` file and runs the same program. The difference in each process’s execution environment is that the processes are given different ranks in the communicator. Because of this, each process prints a different number when it executes.

MPI and Python combine to make succinct source code. In the above program, the line `from mpi4py import MPI` loads the MPI module from the `mpi4py` package. The line `COMM = MPI.COMM_WORLD` accesses a static communicator object, which represents a group of processes which can communicate with each other via MPI commands. The next line, `RANK = COMM.Get_rank()`, accesses the processes *rank* number. A rank is the process’s unique ID within a communicator, and they are essential to learning about other processes. When the program `mpiexec` is first executed, it creates a global communicator and stores it in the variable `MPI.COMM_WORLD`. One of the main purposes of this communicator is to give each of the five processes a unique identifier, or rank. When each process calls `COMM.Get_rank()`, the communicator returns the rank of that process. `RANK` points to a local variable, which is unique for every calling process because each process has its own separate copy of local variables. This gives us a way to distinguish different processes while writing all of the source code for the five processes in a single file.

Here is the syntax for `Get_size()` and `Get_rank()`, where `Comm` is a communicator object:

**Comm.Get\_size()** Returns the number of processes in the communicator. It will return the same number to every process. Parameters:

**Return value** - the number of processes in the communicator

**Return type** - integer

Example:

```

1 #Get_size_example.py
2 from mpi4py import MPI
3 SIZE = MPI.COMM_WORLD.Get_size()
4 print(f"The number of processes is {SIZE}.")
```

examplecode/Get\_size\_example.py

**Comm.Get\_rank()** Determines the rank of the calling process in the communicator. Parameters:

**Return value** - rank of the calling process in the communicator

**Return type** - integer

Example:

```

1 #Get_rank_example.py
2 from mpi4py import MPI
3 RANK = MPI.COMM_WORLD.Get_rank()
4 print(f"My rank is {RANK}.")
```

examplecode/Get\_rank\_example.py

## The Communicator

A communicator is a logical unit that defines which processes are allowed to send and receive messages. In most of our programs we will only deal with the `MPI.COMM_WORLD` communicator, which contains all of the running processes. In more advanced MPI programs, you can create custom communicators to group only a small subset of the processes together. This allows processes to be part of multiple communicators at any given time. By organizing processes this way, MPI can physically rearrange which processes are assigned to which CPUs and optimize your program for speed. Note that within two different communicators, the same process will most likely have a different rank.

Note that one of the main differences between `mpi4py` and MPI in C or Fortran, besides being array-based, is that `mpi4py` is largely object oriented. Because of this, there are some minor changes between the `mpi4py` implementation of MPI and the official MPI specification.

For instance, the MPI Communicator in `mpi4py` is a Python class and MPI functions like `Get_size()` or `Get_rank()` are instance methods of the communicator class. Throughout these MPI labs, you will see functions like `Get_rank()` presented as `Comm.Get_rank()` where it is implied that `Comm` is a communicator object.

## Separate Codes in One File

When an MPI program is run, each process receives the same code. However, each process is assigned a different rank, allowing us to specify separate behaviors for each process. In the following code, the three processes perform different operations on the same pair of numbers.

```

1 #separateCode.py
2 from mpi4py import MPI
3 RANK = MPI.COMM_WORLD.Get_rank()
4
5 a = 2
6 b = 3
7 if RANK == 0:
8     print(a + b)
9 elif RANK == 1:
10    print(a * b)
11 elif RANK == 2:
12    print(max(a, b))

```

examplecode/separateCode.py

**Problem 1.** Write a program which determines the rank  $n$  of the calling process and prints “Hello from process  $n$ ” if  $n$  is even and “Goodbye from process  $n$ ” if  $n$  is odd.

## Message Passing between Processes

Let us begin by demonstrating a program designed for two processes. One will draw a random number and then send it to the other. We will do this using the routines `Comm.Send()` and `Comm.Recv()`.

```

1 #passValue.py
2 import numpy as np
3 from mpi4py import MPI
4
5 COMM = MPI.COMM_WORLD
6 RANK = COMM.Get_rank()
7
8 if RANK == 1: # This process chooses and sends a random value
9     num_buffer = np.random.rand(1)
10    print(f"Process 1: Sending: {num_buffer} to process 0.")
11    COMM.Send(num_buffer, dest=0)
12    print("Process 1: Message sent.")
13 if RANK == 0: # This process receives a value from process 1
14    num_buffer = np.zeros(1)
15    print(f"Process 0: Waiting for the message... current {num_buffer=}.")
16    COMM.Recv(num_buffer, source=1)
17    print(f"Process 0: Message received! {num_buffer=}.")

```

examplecode/passValue.py

To illustrate simple message passing, we have one process choose a random number and then pass it to the other. Inside the receiving process, we have it print out the value of the variable `num_buffer` before it calls `Recv()` to prove that it really is receiving the variable through the message passing interface.

Here is the syntax for `Send()` and `Recv()`, where `Comm` is a communicator object:

**Comm.Send(buf, dest=0, tag=0)** Performs a basic send from one process to another. Parameters:

- buf (array-like)** : data to send
- dest (integer)** : rank of destination
- tag (integer)** : message tag

The `buf` object is not as simple as it appears. It must contain a pointer to a Numpy array. For example, a string must be packaged inside an array before it can be passed. The `tag` object can help distinguish between data if multiple pieces of data are being sent/received by the same processes.

**Comm.Recv(buf, source=0, tag=0, Status status=None)** Basic point-to-point receive of data.

Parameters:

- buf (array-like)** : initial address of receive buffer (choose receipt location)
- source (integer)** : rank of source
- tag (integer)** : message tag
- status (Status)** : status of object

Example:

```

1 #Send_example.py
2 from mpi4py import MPI
3 import numpy as np
4
5 RANK = MPI.COMM_WORLD.Get_rank()
6
7 a = np.zeros(1, dtype=int) # This must be an array.
8 if RANK == 0:
9     a[0] = 10110100
10    MPI.COMM_WORLD.Send(a, dest=1)
11 elif RANK == 1:
12    MPI.COMM_WORLD.Recv(a, source=0)
13    print(a[0])

```

examplecode/Send\_example.py

**Problem 2.** Write a script that runs on two (and only two!) processes and passes a random numpy array of length `n` from the root process to process 1. Write it so that the user passes in the value of `n` as a command-line argument. The following code demonstrates how to access command-line arguments.

```
from sys import argv

# The first command line argument is saved as n.
n = int(argv[1])
```

In process 1, instantiate a zero array of length `n`, then print this array clearly labeled under process 1. Then have the root process generate a random array, and print this array clearly labeled under the root process. Then send the randomly generated array in the root process to process 1, and print the array clearly labeled under process 1. The output should reflect the following for `n = 4` (make sure to follow the output formatting exactly).

```
$ mpiexec -n 2 python problem2.py 4

Process 1: Before checking mailbox: vec=[ 0.  0.  0.  0.]
Process 0: Sent: vec=[ 0.03162613  0.38340242  0.27480538  0.56390755]
Process 1: Received: vec=[ 0.03162613  0.38340242  0.27480538  0.56390755]
```

Hint: if the number of processes is not 2, you can abort the program with `COMM.Abort()`.

#### NOTE

`Send()` and `Recv()` are referred to as *blocking* functions. That is, if a process calls `Recv()`, it will sit idle until it has received a message from a corresponding `Send()` before it will proceed. (However, in Python the process that calls `Comm.Send` will *not* necessarily block until the message is received, though in C, `MPI_Send` does block) There are corresponding *non-blocking* functions `Irecv()` and `Irecv()` (The *I* stands for immediate). In essence, `Irecv()` will return immediately. If a process calls `Irecv()` and doesn't find a message ready to be picked up, it will indicate to the system that it is expecting a message, proceed beyond the `Irecv()` to do other useful work, and then check back later to see if the message has arrived. This can be used to dramatically improve performance.

#### NOTE

When calling `Comm.Recv`, you can allow the calling process to accept a message from any process that happened to be sending to the receiving process. This is done by setting source to a predefined MPI constant, `source=ANY_SOURCE` (note that you would first need to import this with `from mpi4py.MPI import ANY_SOURCE` or use the syntax `source=MPI.ANY_SOURCE`).

**Problem 3.** Write a script in which the process with rank  $i$  sends a random value to the process with rank  $i + 1$  in the global communicator. The process with the highest rank will send its random value to the root process. Notice that we are communicating in a ring. For communication, only use `Send()` and `Recv()`. The script should work with any number of processes. Does the order in which `Send()` and `Recv()` are called matter?

Generate an initial random value in each process. Print each initial value clearly labeled under its process (match the formatting given below). Then send the values to the next process as explained above, and print the value each process ends with clearly labeled. The output should reflect the following (note that both the values and order will vary).

```
$ mpiexec -n 2 python problem3.py

Process 1 started with [ 0.79711384]
Process 1 received [ 0.54029085]
Process 0 started with [ 0.54029085]
Process 0 received [ 0.79711384]

$ mpiexec -n 3 python problem3.py

Process 2 started with [ 0.99893055]
Process 0 started with [ 0.6304739]
Process 1 started with [ 0.28834079]
Process 1 received [ 0.6304739]
Process 2 received [ 0.28834079]
Process 0 received [ 0.99893055]
```

## Application: Monte Carlo Integration

Monte Carlo integration uses random sampling to approximate volumes (whereas most numerical integration methods employ some sort of regular grid). It is a useful technique, especially when working with higher-dimensional integrals. It is also well-suited to parallelization because it involves a large number of independent operations. In fact, Monte Carlo algorithms can be made “embarrassingly parallel” — the processes don’t need to communicate with one another during execution, simply reporting results to the root process upon completion.

In a simple example, the following code calculates the value of  $\pi$  by sampling random points inside the square  $[-1, 1] \times [-1, 1]$ . Since the volume of the unit circle is  $\pi$  and the volume of the square is 4, the probability of a given point landing inside the unit circle is  $\pi/4$ , so the proportion of samples that fall within the unit circle should also be  $\pi/4$ . The program samples  $N = 2000$  points, determines which samples are within the unit circle (say  $M$  are), and estimates  $\pi \approx 4M/N$ .

```
1 # pi.py
2 import numpy as np
3 from scipy import linalg as la
4
5 # Get 2000 random points in the 2-D domain [-1,1]x[-1,1].
```

```

7 points = np.random.uniform(-1, 1, (2, 2000))

9 # Determine how many points are within the unit circle.
lengths = la.norm(points, axis=0)
11 num_within = np.count_nonzero(lengths < 1)

13 # Estimate the circle's area.
print(4 * (num_within / 2000))

```

examplecode/pi.py

```
$ python pi.py
3.166
```

**Problem 4.** The  $n$ -dimensional *open unit ball* is the set  $U_n = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 < 1\}$ . Write a script that accepts integers  $n$  and  $N$  on the command line. Estimate the volume of  $U_n$  by drawing  $N$  points over the  $n$ -dimensional domain  $[-1, 1] \times [-1, 1] \times \cdots \times [-1, 1]$  on each available process except the root process (for a total of  $(r - 1)N$  draws, where  $r$  is the number of processes). The root process should finally print the volume estimate given by the entire set of  $(r - 1)N$  points, with the dimension of the unit ball clearly labeled.

Hint: the volume of  $[-1, 1] \times [-1, 1] \times \cdots \times [-1, 1]$  is  $2^n$ .

When  $n = 2$ , this is the same experiment outlined above so your function should return an approximation of  $\pi$ . The volume of  $U_3$  is  $\frac{4}{3}\pi \approx 4.18879$ , and the volume of  $U_4$  is  $\frac{\pi^2}{2} \approx 4.9348$ . Try increasing the number of sample points  $N$  or processes  $r$  to see if your estimates improve. The output of a 4 process estimate of  $U_2$  with 2000 draws should reflect the following.

```
$ mpiexec -n 4 python problem4.py 2 2000
```

```
Volume of 2-D unit ball: 3.132666666667
```

### NOTE

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages requires a level of synchronization and causes some processes to pause as they wait to receive or send messages, negating the advantages of parallelism. It is also important to divide work evenly between simultaneous processes, as a program can only be as fast as its slowest process. This is called load balancing, and can be difficult in more complex algorithms.



# 17

# Web Scraping

**Lab Objective:** *Web Scraping is the process of gathering data from websites on the internet. Since almost everything rendered by an internet browser as a web page uses HTML, the first step in web scraping is being able to extract information from HTML. In this lab, we introduce the requests library for scraping web pages, BeautifulSoup: Python's canonical tool for efficiently and cleanly navigating and parsing HTML, and how to use Pandas to extract data from HTML tables.*

## HTTP and Requests

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP, which we used to build a server in the Web Technologies lab, but uses TCP protocols to manage connections and provide network capabilities. The HTTP protocol is centered around a request and response paradigm, in which a client makes a request to a server and the server replies with a response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. A GET request asks for information from the server, a POST request modifies the state of the server, and a PUT request adds new pieces of data to the server.

The standard way to get the source code of a website using Python is via the `requests` library.<sup>1</sup> Calling `requests.get()` sends an HTTP GET request to a specified website. The website returns a response code, which indicates whether or not the request was received, understood, and accepted. If the response code is good, typically 200<sup>2</sup>, then the response will also include the website source code as an HTML file.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.byu.edu")
>>> print(response.status_code, response.ok, response.reason)
200 True OK
```

<sup>1</sup>Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See <http://docs.python-requests.org/>.

<sup>2</sup>See [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for explanation of specific response codes.

```
# The HTML of the website is stored in the "text" attribute.
>>> print(response.text)
<!DOCTYPE html>
<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/←
    content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ ←
    og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema:←
        http://schema.org/ sioc: http://rdfs.org/sioc/ns# sioct: http://rdfs.org←
        /sioc/types# skos: http://www.w3.org/2004/02/skos/core# xsd: http://www.←
        w3.org/2001/XMLSchema# " class=" ">

<head>
    <meta charset="utf-8" />
# ...
```

Attribute	Description
<code>text</code>	Content of the response, in unicode
<code>content</code>	Content of the response, in bytes
<code>encoding</code>	Encoding used to decode response.content
<code>raw</code>	Raw socket response, which allows applications to send and obtain packets of information
<code>status_code</code>	Numeric code that shows how the action was successful
<code>ok</code>	Boolean that is <code>True</code> if the <code>status_code</code> is less than 400, or <code>False</code> if not
<code>reason</code>	Text corresponding to the status code

Table 17.1: Different content attributes of the request class.

Note that some websites aren't built to handle large amounts of traffic or many repeated requests. Most are built to identify web scrapers or crawlers that initiate many consecutive GET requests without pauses, and retaliate or block them. When web scraping, always make sure to store the data that you receive in a file and include error checks to prevent retrieving the same data unnecessarily. We won't spend much time on that in this lab, but it's especially important in larger applications.

**Problem 1.** Use the `requests` library to get the HTML source for the website `http://www.example.com`. Save the source as a file called "`example.html`". If the file already exists, make sure not to scrape the website or overwrite the file. You will use this file later in the lab.

Hint: When writing responses to file you need to use the `open()` function with the appropriate file write mode ((either `mode='w'` for plain write mode, or `mode="wb"` for binary write mode). Python interpreters will write the example file the same way whether you use `response.text` or `response.content`.

ACHTUNG!

Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.<sup>a</sup>

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`.<sup>b</sup>

We will cover this more in the next lab.

<sup>a</sup>See [www.robotstxt.org/orig.html](http://www.robotstxt.org/orig.html) and [en.wikipedia.org/wiki/Robots\\_exclusion\\_standard](https://en.wikipedia.org/wiki/Robots_exclusion_standard).

<sup>b</sup>Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see <https://docs.python.org/3/library/urllib.robotparser.html>.

## HTML

*Hyper Text Markup Language*, or *HTML*, is the standard *markup language*—a language designed for the processing, definition, and presentation of text—for creating webpages. It structures a document using pairs of *tags* that surround and define content. Opening tags have a tag name surrounded by angle brackets (`<tag-name>`). The companion closing tag looks the same, but with a forward slash before the tag name (`</tag-name>`). A list of all current HTML tags can be found at <http://htmldog.com/reference/htmltags>.

Most tags can be combined with *attributes* to include more data about the content, help identify individual tags, and make navigating the document much simpler. In the following example, the `<a>` tag has `id` and `href` attributes.

```
<html>                                <!-- Opening tags -->
  <body>
    <p>
      Click <a id='info' href='http://www.example.com'>here</a>
      for more information.
    </p>                               <!-- Closing tags -->
  </body>
</html>
```

In HTML, `href` stands for *hypertext reference*, a link to another website. Thus the above example would be rendered by a browser as a single line of text, with `here` being a clickable link to <http://www.example.com>:

Click here for more information.

Unlike Python, HTML does not enforce indentation (or any whitespace rules), though indentation generally makes HTML more readable. The previous example can be written in a single line.

```
<html><body><p>Click <a id='info' href='http://www.example.com/info'>here</a>
  for more information.</p></body></html>
```

Special tags, which don't contain any text or other tags, are written without a closing tag and in a single pair of brackets. A forward slash is included between the name and the closing bracket. Examples of these include `<hr/>`, which describes a horizontal line, and `<img/>`, the tag for representing an image.

#### NOTE

You can open .html files using a text editor or any web browser. In a browser, you can inspect the source code associated with specific elements. Right click the element and select **Inspect**. If you are using Safari, you may first need to enable "Show Develop menu" in "Preferences" under the "Advanced" tab.

## BeautifulSoup

BeautifulSoup (`bs4`) is a package<sup>3</sup> that makes it simple to navigate and extract data from HTML documents. See <http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html> for the full documentation.

The `bs4.BeautifulSoup` class accepts two parameters to its constructor: a string of HTML code and an HTML parser to use under the hood. The HTML parser is technically a keyword argument, but the constructor prints a warning if one is not specified. The standard choice for the parser is "`html.parser`", which means the object uses the standard library's `html.parser` module as the engine behind the scenes.

#### NOTE

Depending on project demands, a parser other than "`html.parser`" may be useful. A couple of other options are "`lxml`", an extremely fast parser written in C, and "`html5lib`", a slower parser that treats HTML in much the same way a web browser does, allowing for irregularities. Both must be installed independently; see <https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser> for more information.

A BeautifulSoup object represents an HTML document as a tree. In the tree, each tag is a *node* with nested tags and strings as its *children*. The `prettyify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format that reflects the tree structure.

```
>>> from bs4 import BeautifulSoup

>>> small_example_html = """
<html><body><p>
    Click <a id='info' href='http://www.example.com'>here</a>
    for more information.
</p></body></html>
"""
```

<sup>3</sup>BeautifulSoup is not part of the standard library; install it with `pip install beautifulsoup4`.

```
>>> small_soup = BeautifulSoup(small_example_html, "html.parser")
>>> print(small_soup.prettify())
<html>
  <body>
    <p>
      Click
      <a href="http://www.example.com" id="info">
        here
      </a>
      for more information.
    </p>
  </body>
</html>
```

Each tag in a `BeautifulSoup` object's HTML code is stored as a `bs4.element.Tag` object, with actual text stored as a `bs4.element.NavigableString` object. Tags are accessible directly through the `BeautifulSoup` object.

```
# Get the <p> tag (and everything inside of it).
>>> small_soup.p
<p>
  Click <a href="http://www.example.com" id="info">here</a>
  for more information.
</p>

# Get the <a> sub-tag of the <p> tag.
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='\n')
<a href="http://www.example.com" id="info">here</a>
<class 'bs4.element.Tag'>

# Get just the name, attributes, and text of the <a> tag.
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a
{'id': 'info', 'href': 'http://www.example.com'}
here
```

Attribute	Description
<code>name</code>	The name of the tag
<code>attrs</code>	A dictionary of the attributes
<code>string</code>	The single string contained in the tag
<code>strings</code>	Generator for strings of children tags
<code>stripped_strings</code>	Generator for strings of children tags, stripping whitespace
<code>text</code>	Concatenation of strings from all children tags

Table 17.2: Data attributes of the `bs4.element.Tag` class.

**Problem 2.** The `BeautifulSoup` class has a `find_all()` method that, when called with `True` as the only argument, returns a list of all tags in the HTML source code.

Write a function that accepts a string of HTML code as an argument. Use `BeautifulSoup` to return a list of the **names** of the tags in the code.

## Navigating the Tree Structure

Not all tags are easily accessible from a `BeautifulSoup` object. Consider the following example.

```
>>> pig_html = """
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title"><b>The Three Little Pigs</b></p>
<p class="story">Once upon a time, there were three little pigs named
<a href="http://example.com/larry" class="pig" id="link1">Larry,</a>
<a href="http://example.com/mo" class="pig" id="link2">Mo</a>, and
<a href="http://example.com/curly" class="pig" id="link3">Curly.</a>
<p>The three pigs had an odd fascination with experimental construction.</p>
<p>...</p>
</body></html>
"""

>>> pig_soup = BeautifulSoup(pig_html, "html.parser")
>>> pig_soup.p
<p class="title"><b>The Three Little Pigs</b></p>

>>> pig_soup.a
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>
```

Since the HTML in this example has several `<p>` and `<a>` tags, only the **first** tag of each name is accessible directly from `pig_soup`. The other tags can be accessed by manually navigating through the HTML tree.

Every HTML tag (except for the topmost tag, which is usually `<html>`) has a *parent* tag. Each tag also has zero or more *sibling* and *children* tags or text. Following a true tree structure, every `bs4.element.Tag` in a soup has multiple attributes for accessing or iterating through parent, sibling, or child tags.

Attribute	Description
<code>parent</code>	The parent tag
<code>parents</code>	Generator for the parent tags up to the top level
<code>next_sibling</code>	The tag immediately after to the current tag
<code>next_siblings</code>	Generator for sibling tags after the current tag
<code>previous_sibling</code>	The tag immediately before the current tag
<code>previous_siblings</code>	Generator for sibling tags before the current tag
<code>contents</code>	A list of the immediate children tags
<code>children</code>	Generator for immediate children tags
<code>descendants</code>	Generator for all children tags (recursively)

Table 17.3: Navigation attributes of the `bs4.element.Tag` class.

```
# Start at the first <a> tag in the soup.
>>> a_tag = pig_soup.a
>>> a_tag
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

# Get the names of all of <a>'s parent tags, traveling up to the top.
# The name "[document]" means it is the top of the HTML code.
>>> [par.name for par in a_tag.parents]      # <a>'s parent is <p>, whose
['p', 'body', 'html', '[document]']          # parent is <body>, and so on.

# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'                                         # The first sibling is just text.
>>> a_tag.next_sibling.next_sibling          # The second sibling is a tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

Note carefully that newline characters are considered to be children of a parent tag. Therefore iterating through children or siblings often requires checking which entries are tags and which are just text. In the next example, we use a tag's `attrs` attribute to access specific attributes within the tag (see Table 17.2).

```
# Get to the <p> tag that has class="story" using these commands.
>>> p_tag = pig_soup.body.p.next_sibling.next_sibling
>>> p_tag.attrs["class"]                      # Make sure it's the right tag.
['story']

# Iterate through the child tags of <p> and print hrefs whenever they exist.
>>> for child in p_tag.children:
...     # Skip the children that are not bs4.element.Tag objects
...     # These don't have the attribute "attrs"
...     if hasattr(child, "attrs") and "href" in child.attrs:
...         print(child.attrs["href"])
http://example.com/larry
http://example.com/mo
http://example.com/curly
```

Note that the `"class"` attribute of the `<p>` tag is a list. This is because the `"class"` attribute can take on several values at once; for example, the tag `<p class="story book">` is of class `"story"` and of class `"book"`.

The behavior of the `string` attribute of a `bs4.element.Tag` object depends on the structure of the corresponding HTML tag.

1. If the tag has a string of text and no other child elements, then `string` is just that text.
2. If the tag has exactly one child tag and the child tag has only a string of text, then the tag has the same `string` as its child tag.
3. If the tag has more than one child, then `string` is `None`. In this case, use `strings` to iterate through the child strings. Alternatively, the `get_text()` method returns all text belonging to a tag and to all of its descendants. In other words, it returns anything inside a tag that isn't another tag.

```
>>> pig_soup.head
<head><title>Three Little Pigs</title></head>

# Case 1: the <title> tag's only child is a string.
>>> pig_soup.head.title.string
'Three Little Pigs'

# Case 2: The <head> tag's only child is the <title> tag.
>>> pig_soup.head.string
'Three Little Pigs'

# Case 3: the <body> tag has several children.
>>> pig_soup.body.string is None
True
>>> print(pig_soup.body.get_text().strip())
The Three Little Pigs
Once upon a time, there were three little pigs named
Larry,
Mo, and
Curly.
The three pigs had an odd fascination with experimental construction.
...
```

**Problem 3.** Write a function that reads a file of the same format as the file generated from Problem 1 and load it into BeautifulSoup. Find the first `<a>` tag, and return its text along with a boolean value indicating whether or not it has a hyperlink (`href` attribute).

## Searching for Tags

Navigating the HTML tree manually can be helpful for gathering data out of lists or tables, but these kinds of structures are usually buried deep in the tree. The `find()` and `find_all()` methods of the `BeautifulSoup` class identify tags that have distinctive characteristics, making it much easier to jump straight to a desired location in the HTML code. The `find()` method only returns the **first** tag that matches a given criteria, while `find_all()` returns a list of all matching tags. Tags can be matched by name, attributes, and/or text.

```
# Find the first <b> tag in the soup.
>>> pig_soup.find(name='b')
<b>The Three Little Pigs</b>

# Find all tags with a class attribute of "pig".
# Since "class" is a Python keyword, use "class_" as the argument.
>>> pig_soup.find_all(class_="pig")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the first tag that matches several attributes.
>>> pig_soup.find(attrs={"class": "pig", "href": "http://example.com/mo"})
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

# Find the first tag whose text is "Mo".
>>> pig_soup.find(string="Mo")
'Mo'                                         # The result is the actual string,
>>> pig_soup.find(string="Mo").parent      # so go up one level to get the tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

**Problem 4.** The file `san_diego_weather.html` contains the HTML source for an old page from Weather Underground.<sup>a</sup> Write a function that reads the file and loads it into BeautifulSoup.

Return a list of the following tags:

1. The tag containing the date “Thursday, January 1, 2015”.
2. The tags which contain the **links** “Previous Day” and “Next Day”.

Hint: the tags that contain the links do not explicitly have “Previous Day” or “Next Day,” so you cannot find the tags based on a string. Instead, open the HTML file manually and look at the attributes of the tags that contain the links. Then inspect them to see if there is a better attribute with which to conduct a search.

3. The tag which contains the number associated with the Max Actual Temperature.

Hint: Manually inspect the HTML document in both a browser and in a text editor. Look for a good attribute shared by the Actual Temperatures with which you can conduct a search. Note that the Max Actual Temperature tag will not be the first tag found. Make sure you return the correct tag.

<sup>a</sup>See [http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req\\_city=San+Diego&req\\_state=CA&req\\_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1](http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1)

## Advanced Search Techniques: Regular Expressions

Consider the problem of finding the tag that is a link to the URL <http://example.com/curly>.

```
>>> pig_soup.find(href="http://example.com/curly")
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>
```

This approach works, but it requires entering in the entire URL. To perform generalized searches, the `find()` and `find_all()` method also accept compiled regular expressions from the `re` module. This way, the methods locate tags whose name, attributes, and/or string matches a pattern.

```
>>> import re

# Find the first tag with an href attribute containing "curly".
>>> pig_soup.find(href=re.compile(r"curly"))
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find the first tag with a string that starts with "Cu".
>>> pig_soup.find(string=re.compile(r"^Cu")).parent
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find all tags with text containing "Three".
>>> [tag.parent for tag in pig_soup.find_all(string=re.compile(r"Three"))]
[<title>Three Little Pigs</title>, <b>The Three Little Pigs</b>]
```

Finally, to find a tag that has a particular attribute, regardless of the actual value of the attribute, use `True` in place of search values.

```
# Find all tags with an "id" attribute.
>>> pig_soup.find_all(id=True)
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the names all tags WITHOUT an "id" attribute.
>>> [tag.name for tag in pig_soup.find_all(id=False)]
['html', 'head', 'title', 'body', 'p', 'b', 'p', 'p', 'p']
```

It is important to note that using Regular Expressions to locate tags will only find tags whose name, attributes, and/or string match exactly. If searching through an HTML file with a Regular expression to find all the tags with an "`id`" attribute with labels that you specify, it will only return the tags found with those labels, not all instances of the labels. Sometimes that is very useful in simplifying Regular Expression searches.

## Advanced Search Techniques: CSS Selectors

`BeautifulSoup` also supports the use of CSS selectors. CSS (Cascading Style Sheet) describes the style and layout of a webpage, and CSS selectors provide a useful way to navigate HTML code. Use the method `soup.select()` to find all elements matching an argument. The general format for an argument is `tag-name[attribute-name = 'attribute value']`. The table below lists symbols you can use to more precisely locate various elements.

Symbol	Meaning
=	Matches an attribute value exactly
*=	Partially matches an attribute value
^=	Matches the beginning of an attribute value
\$=	Matches the end of an attribute value
+	Next sibling of matching element
>	Search an element's children

Table 17.4: CSS symbols for use with Selenium

You can do many other useful things with CSS selectors. A helpful guide can be found at [https://www.w3schools.com/cssref/css\\_selectors.asp](https://www.w3schools.com/cssref/css_selectors.asp). The code below gives an example using arguments described above.

```
# Find all <a> tags with id="link1"
>>> pig_soup.select("[id='link1']")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>]

# Find all tags with an href attribute containing "curly".
>>> pig_soup.select("[href*='curly']")
[<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <a> tags with an href attribute
>>> pig_soup.select("a[href]")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <b> tags within a <p> tag with class="title"
>>> pig_soup.select("p[class='title'] b")
[<b>The Three Little Pigs</b>]

# Use a comma to find elements matching one of two arguments
>>> pig_soup.select("a[href$='mo'],[id='link3']")
[<a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
```

```
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

**Problem 5.** The file `large_banks_index.html` is an index of data about large banks, as recorded by the Federal Reserve.<sup>a</sup> Write a function that reads the file and loads the source into BeautifulSoup. Return a list of the `<a>` tags containing the links to bank data from September 30, 2003 to December 31, 2014, where the dates are in reverse chronological order.

<sup>a</sup>See <https://www.federalreserve.gov/releases/lbr/>.

## Incorporating Pandas with HTML Tables

The Pandas `pd.read_html` method is a neat way to automatically read tables into DataFrames. It works similarly to BeautifulSoup, but is streamlined to only scrape all the tables from HTML pages.

```
>>> import pandas as pd
#Read in "file.html"
>>> tables = pd.read_html("file.html")
# Choose the correct table that pd.read_html found as a DataFrame
>>> df = tables[0]
# The DataFrame is now like any other Pandas DataFrame that needs a bit of data←
    cleaning
>>> df[["column_1"]]
0      0
1      2
2      .
3      6
4      8
...
15     30
16     32
17     .
18     .
19     38
```

Like any other DataFrame table, if there are non-numerical values in a column, the column type will default to strings, so make sure when sorting values by number, set the column data type to numeric with `pd.to_numeric`.

For more help with `pd.read_html` see [https://pandas.pydata.org/docs/reference/api/pandas.read\\_html.html](https://pandas.pydata.org/docs/reference/api/pandas.read_html.html).

**Problem 6.** The file `large_banks_data.html` is one of the pages from the index in Problem 5.<sup>a</sup> Read the specified file and load it into Pandas.

Create a single figure with two subplots:

1. A sorted bar chart of the seven banks with the most domestic branches.

2. A sorted bar chart of the seven banks with the most foreign branches.

<sup>a</sup>See <http://www.federalreserve.gov/releases/lbr/20030930/default.htm>.



# 18

## Metropolis Algorithm

**Lab Objective:** *Understand the basic principles of the Metropolis algorithm and apply these ideas to the Ising Model.*

### The Metropolis Algorithm

Sampling from a given probability distribution is an important part of many tasks throughout the sciences. When modeling real-world problems, these distributions are often very complicated, and direct sampling methods require computing high-dimensional integrals and are thus impractical. The Metropolis algorithm is an effective method to sample from many of these distributions. This algorithm only requires evaluating the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute any difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

Suppose that  $h : \mathbb{R}^n \rightarrow \mathbb{R}$  is the probability density function of a distribution that is difficult to evaluate (for example, a Bayesian posterior distribution), while some function  $f(\boldsymbol{\theta}) = c \cdot h(\boldsymbol{\theta})$  is easy to evaluate. The Metropolis algorithm is an MCMC sampling method which constructs a Markov chain  $Y$  whose invariant distribution is exactly the distribution associated with  $h$ . We can then use these samples as a sample from this distribution.

For the Metropolis algorithm, we need two ingredients: a *proposal function*, and an *acceptance function*. The proposal function is used to choose a potential next state. We denote this function as  $Q(\mathbf{x}, \mathbf{y}) : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ . For each  $\mathbf{y} \in \mathbb{R}^n$ ,  $Q(\cdot, \mathbf{y})$  is the probability density function for the proposed state. This distribution needs to be easy to sample from; typical choices are uniform or normal distributions. For simplicity we also require this function to be *symmetric*, so  $Q(\mathbf{x}, \mathbf{y}) = Q(\mathbf{y}, \mathbf{x})$ . In words, the probability density of moving from  $\mathbf{x}$  to  $\mathbf{y}$  is the same as moving from  $\mathbf{y}$  to  $\mathbf{x}$  for all  $\mathbf{x}$  and  $\mathbf{y}$ .

The acceptance function  $A : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  gives the probability that we actually transition from the current state  $\mathbf{y}$  to the proposed state  $\mathbf{x}$ .<sup>1</sup> This function is defined by

$$A(\mathbf{x}, \mathbf{y}) = \min \left( 1, \frac{f(\mathbf{x})}{f(\mathbf{y})} \right).$$

---

<sup>1</sup>In the Volume 3 textbook, the proposal function for continuous distributions and the acceptance function are denoted  $f_{X_{t+1}|X_t=\mathbf{y}}(\mathbf{x})$  and  $a_{\mathbf{x}, \mathbf{y}}$  respectively. In this lab we instead write them as  $Q(\mathbf{x}, \mathbf{y})$  and  $A(\mathbf{x}, \mathbf{y})$  to make it clearer that they are functions of  $\mathbf{x}$  and  $\mathbf{y}$ .

Following the proposals from  $Q$  causes us to wander around the space of allowed states. The acceptance function from  $A$  modifies this wandering so that we spend more time in more likely regions.

---

**Algorithm 1** Metropolis Algorithm

---

```

1: procedure METROPOLIS ALGORITHM
2:   Choose initial point  $\mathbf{y}_0$ .
3:   for  $t = 1, 2, \dots$  do
4:     Draw  $\mathbf{x} \sim Q(\cdot, \mathbf{y}_{t-1})$ 
5:     Draw  $a \sim \text{unif}(0, 1)$ 
6:     if  $a \leq A(\mathbf{x}, \mathbf{y}_{t-1})$  then
7:        $\mathbf{y}_t = \mathbf{x}$ 
8:     else
9:        $\mathbf{y}_t = \mathbf{y}_{t-1}$ 
10:    Return  $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \dots$ 
```

---

These functions form the basis for the Metropolis algorithm. At each step, given our current state  $\mathbf{y}_t$ , we propose a new state according to the distribution  $\mathbf{x} \sim Q(\cdot, \mathbf{y}_t)$ . We then accept the proposed state with probability  $A(\mathbf{x}, \mathbf{y}_t)$ . If we accept the proposal, we set  $\mathbf{y}_{t+1} = \mathbf{x}$ ; otherwise, we set  $\mathbf{y}_{t+1} = \mathbf{y}_t$ . Refer to Algorithm 1 for a write-up of this algorithm. Under certain conditions on  $Q$ , the Markov chain the samples  $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \dots$  are from will have a unique invariant distribution with density  $h$ , and any initial state will converge to this distribution.

We can consider each of the samples  $\mathbf{y}_i$  as draws from the distribution of  $h$ . Most of the time we don't just want samples from the distribution, but *independent* samples. However, the samples  $\mathbf{y}_t$  and  $\mathbf{y}_{t+1}$  are clearly not independent. We can get around this issue by only keeping some of the samples, for example every 10th or 100th sample. While  $\mathbf{y}_t$  and  $\mathbf{y}_{t+1}$  aren't independent,  $\mathbf{y}_t$  and  $\mathbf{y}_{t+100}$  will be closer to being independent.

Finally, for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

$$\log A(\mathbf{x}, \mathbf{y}) = \min(0, \log f(\mathbf{x}) - \log f(\mathbf{y})).$$

Let's apply the Metropolis algorithm to an example of Bayesian analysis. Consider the exam scores in `examscores.csv`, and suppose that these scores are distributed normally with (unknown) mean  $\mu$  and variance  $\sigma^2$ . We wish to compute the posterior distribution for  $\mu$  and  $\sigma^2$ . Denote the data as  $d_1, \dots, d_N$  and assume the prior distributions

$$\begin{aligned}\mu &\sim \mathcal{N}(m = 80, s^2 = 16) \\ \sigma^2 &\sim IG(\alpha = 3, \beta = 50).\end{aligned}$$

Note that  $IG$  is the inverse gamma distribution. In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 | d_1, \dots, d_N) = \frac{p(\mu)p(\sigma^2) \prod_{i=1}^N \mathcal{N}(d_i | \mu, \sigma^2)}{\int_{-\infty}^{\infty} \int_0^{\infty} p(\mu')p(\sigma'^2) \prod_{i=1}^N \mathcal{N}(d_i | \mu', \sigma'^2) d\sigma'^2 d\mu'}.$$

However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to  $\mu$  and  $\sigma^2$ , the numerator can serve as the function  $f$  in the Metropolis algorithm, and the denominator can serve as the constant  $c$ .

We choose our proposal function to be based on a bivariate Normal distribution:

$$Q(\mathbf{x}, \mathbf{y}) = \mathcal{N}(\mathbf{x} | \mathbf{y}, uI),$$

i.e. normally distributed with mean  $\mathbf{y}$  and variance  $uI$  where  $I$  is the  $2 \times 2$  identity matrix and  $u > 0$ .

```

def proposal(y, u):
    """Returns the proposal, i.e. a draw from Q(x|y,uI)."""
    return stats.multivariate_normal.rvs(mean=y, cov=u*np.eye(len(x)))

def propLogDensity(x, muprior, sig2prior, scores):
    """Calculate the log of the proportional density function f."""
    if x[1] <= 0:
        return -np.inf
    logprob = muprior.logpdf(x[0]) + sig2prior.logpdf(x[1])
    logprob += stats.norm.logpdf(scores, loc=x[0], scale=np.sqrt(x[1])).sum()
    return logprob

def acceptance(x, y, muprior, sig2prior, scores):
    """
    Returns the acceptance probability of moving from y to x.
    """
    return np.exp(min(0,
                      propLogDensity(x, muprior, sig2prior, scores)
                      - propLogDensity(y, muprior, sig2prior, scores)
                      ))

```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log probabilities  $\log f(\mathbf{y}_t)$  and the proportion of proposed samples that were accepted.

We will evaluate the quality of our results by plotting the log probabilities, the  $\mu$  samples, the  $\sigma^2$  samples, and kernel density estimators for the marginal posterior distributions of  $\mu$  and  $\sigma^2$ . The kernel density estimators approximate the continuous distribution of the marginal distributions. The kernel density estimator for  $\mu$  should be approximately normal, and the kernel density estimator for  $\sigma^2$  should be approximately an inverse gamma.

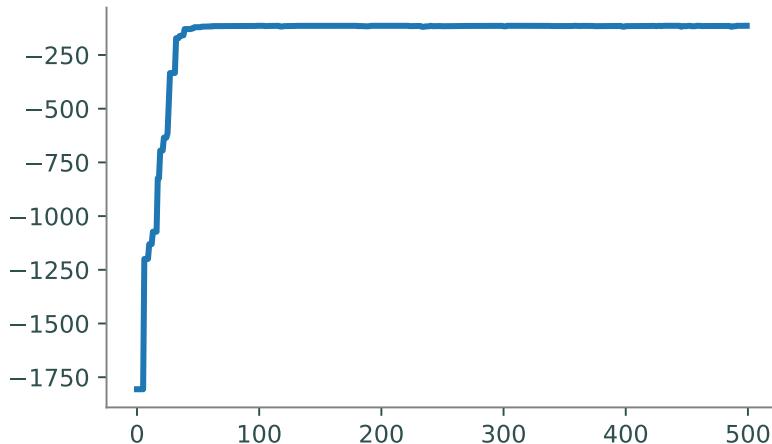


Figure 18.1: Log densities of the first 500 Metropolis samples.

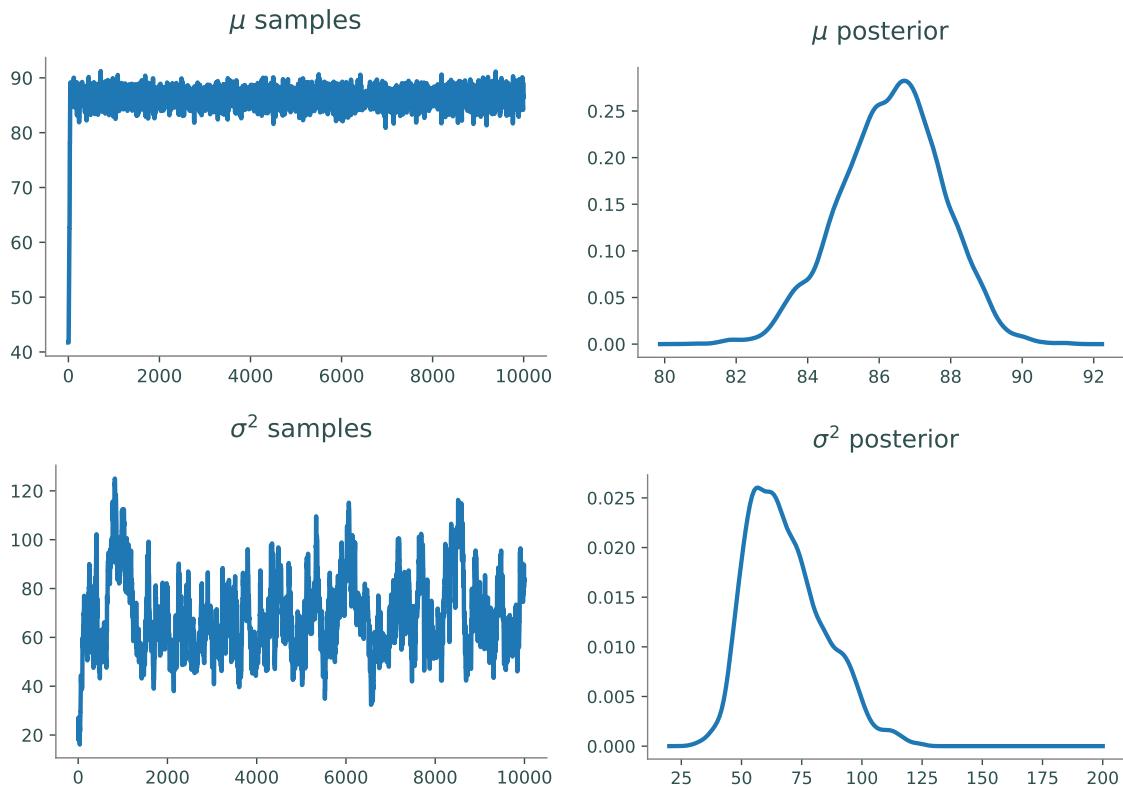


Figure 18.2: Metropolis samples and KDEs for the marginal posterior distribution of  $\mu$  (top row) and  $\sigma^2$  (bottom row).

**Problem 1.** Write a function that uses the Metropolis Hastings algorithm to draw from the posterior distribution over the mean  $\mu$  and variance  $\sigma^2$ . Use the given functions and Algorithm 1 to complete the problem.

Your function should return an array of draws, an array of the log probabilities, and an acceptance rate. Create plots resembling Figures 18.1 and 18.2:

- Plot the log probabilities of the first 500 samples.
- Plot the samples for  $\mu$  in the order they were drawn, and likewise for  $\sigma^2$ .
- Using `seaborn.kdeplot` plot the distribution of all samples for  $\mu$ , and likewise for  $\sigma^2$ .

Use  $u = 20$  for the parameter of the proposal function. Use the initial state  $\mathbf{y}_0 = (\mu_0, \sigma_0^2) = (40, 10)$ . Take 10,000 samples for both  $\mu$  and  $\sigma^2$ .

Use the following code to load the data and initialize the priors:

```
# Load in the data and initialize priors
>>> scores = np.load("examscores.npy")
```

```
# Prior sigma^2 ~ IG(alpha, beta)
>>> alpha = 3
>>> beta = 50
>>> muprior = stats.norm(loc=m, scale=sqrt(s**2))

#Prior mu ~ N(m, s)
>>> m = 80
>>> s = 4
>>> sig2prior = stats.invgamma(alpha, scale=beta)
```

## The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice  $\Lambda$  of sites. We say  $i \sim j$  if  $i$  and  $j$  are adjacent sites. Each site  $i$  in our lattice is assigned an associated *spin*  $\sigma_i \in \{\pm 1\}$ . A *state* in our Ising model is a particular spin configuration  $\sigma = (\sigma_k)_{k \in \Lambda}$ . If  $L = |\Lambda|$ , then there are  $2^L$  possible states in our model. If  $L$  is large, the state space becomes huge, which is why MCMC sampling methods (in particular the Metropolis algorithm) are so useful in calculating model estimations.

With any spin configuration  $\sigma$ , there is an associated energy

$$H(\sigma) = -J \sum_{i \sim j} \sigma_i \sigma_j$$

where  $J > 0$  for ferromagnetic materials, and  $J < 0$  for antiferromagnetic materials. Throughout this lab, we will assume  $J = 1$ , leaving the energy equation to be  $H(\sigma) = -\sum_{i \sim j} \sigma_i \sigma_j$  where the interaction from each pair is added only once.

We will consider a lattice that is a  $100 \times 100$  square grid. The adjacent sites for a given site are those directly above, below, to the left, and to the right of the site, so to speak. For sites on the edge of the grid, we assume it wraps around. In other words, a site at the farthest left side of the grid is adjacent to the corresponding site on the farthest right side. Thus, a single spin configuration can be represented as a  $100 \times 100$  array, with entries of  $\pm 1$ .

The following code will construct a random spin configuration of size  $n$ :

```
def random_lattice(n):
    """Constructs a random spin configuration for an nxn lattice."""
    random_spin = np.zeros((n, n))
    for k in range(n):
        random_spin[k, :] = 2*np.random.binomial(1, .5, n) - 1
    return random_spin
```

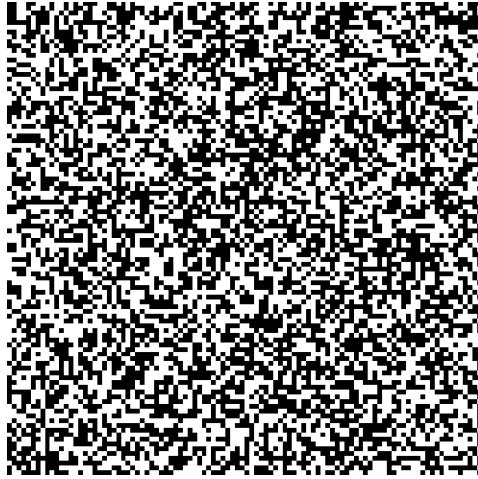


Figure 18.3: Spin configuration from random initialization.

**Problem 2.** Write a function that accepts a spin configuration  $\sigma$  for a lattice as a NumPy array. Compute the energy  $H(\sigma)$  of the spin configuration. Be careful to not double count site pair interactions!

(Hint: `np.roll()` may be helpful.)

Different spin configurations occur with different probabilities, depending on the energy of the spin configuration and  $\beta > 0$ , a quantity inversely proportional to the temperature. More specifically, for a given  $\beta$ , we have

$$\mathbb{P}_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta}$$

where  $Z_\beta = \sum_\sigma e^{-\beta H(\sigma)}$ . Because there are  $2^{100 \cdot 100} = 2^{10000}$  possible spin configurations for our particular lattice, computing this sum is infeasible. However, the numerator is quite simple, provided we can efficiently compute the energy  $H(\sigma)$  of a spin configuration. Thus the ratio of the probability densities of two spin configurations is simple:

$$\frac{\mathbb{P}_\beta(\sigma^*)}{\mathbb{P}_\beta(\sigma)} = \frac{e^{-\beta H(\sigma^*)}}{e^{-\beta H(\sigma)}} = e^{\beta(H(\sigma) - H(\sigma^*))}$$

The simplicity of this ratio should lead us to think that a Metropolis algorithm might be an appropriate way by which to sample from the spin configuration probability distribution, in which case the acceptance probability would be

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases} \quad (18.1)$$

By choosing our transition matrix  $Q$  cleverly, we can also make it easy to compute the energy for any proposed spin configuration. We restrict our possible proposals to only those spin configurations in which we have flipped the spin at exactly one lattice site, i.e. we choose a lattice site  $i$  and flip its spin. Thus, there are only  $L$  possible proposal spin configurations  $\sigma^*$  given  $\sigma$ , each being proposed with probability  $\frac{1}{L}$ , and such that  $\sigma_j^* = \sigma_j$  for all  $j \neq i$ , and  $\sigma_i^* = -\sigma_i$ . Note that we would never actually write out this matrix (it would be  $2^{10000} \times 2^{10000}$ ). Computing the proposed site's energy is simple: if the spin flip site is  $i$ , then we have

$$H(\sigma^*) = H(\sigma) + 2 \sum_{j:j \sim i} \sigma_i \sigma_j. \quad (18.2)$$

**Problem 3.** Write a function that accepts an integer  $n$  and chooses a pair of indices  $(i, j)$  where  $0 \leq i, j \leq n - 1$ . Each possible pair should have an equal probability  $\frac{1}{n^2}$  of being chosen.

**Problem 4.** Write a function that accepts a spin configuration  $\sigma$ , its energy  $H(\sigma)$ , and integer indices  $i$  and  $j$ . Use (18.2) to compute the energy of the new spin configuration  $\sigma^*$ , which is  $\sigma$  but with the spin flipped at the  $(i, j)$ th entry of the corresponding lattice. Do not explicitly construct the new lattice for  $\sigma^*$ .

**Problem 5.** Write a function that accepts a float  $\beta$  and spin configuration energies  $H(\sigma)$  and  $H(\sigma^*)$ . Using (18.1), calculate whether or not the new spin configuration  $\sigma^*$  should be accepted (return `True` or `False`). Consider doing the calculations in log space. (Hint: `np.random.binomial()` might be useful)

To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator  $Z_\beta$ , which is generally the reason we have to use a Metropolis algorithm to begin with. We can get away with examining only  $-\beta H(\sigma)$ . We should see this value increase as the algorithm proceeds, and it should converge once we are sampling from the correct distribution. Note that we don't expect these values to converge to a specific value, but rather to a restricted range of values.

**Problem 6.** Write a function that accepts a float  $\beta > 0$  and integers  $n$ , `n_samples`, and `burn_in`. Initialize an  $n \times n$  lattice for a spin configuration  $\sigma$  using Problem 2. Use the Metropolis algorithm to (potentially) update the lattice `burn_in` times.

1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration  $\sigma^*$ .
2. Use Problem 4 to calculate the energy  $H(\sigma^*)$  of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set  $\sigma = \sigma^*$  by flipping the spin at the indicated site.

4. Track  $-\beta H(\sigma)$  at each iteration (independent of acceptance).

After the burn-in period, continue the iteration `n_samples` times, also recording every 100th sample (to prevent memory failure). The acceptance rate is counted after the burn-in period. Return the samples, the sequence of weighted energies  $-\beta H(\sigma)$ , and the acceptance rate.

Test your sampler on a  $100 \times 100$  grid with 20,000 total iterations, with `n_samples` large enough so that you will keep 50 samples, for  $\beta = 0.2, 0.4, 1$ . Plot the proportional log probabilities across all iterations (including the burn-in), as well as a late sample from each test. How does the ferromagnetic material behave differently with differing temperatures? Recall that  $\beta$  is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figure 18.4.

To show the spin configuration, use `plt.imshow(L, cmap='gray')`.

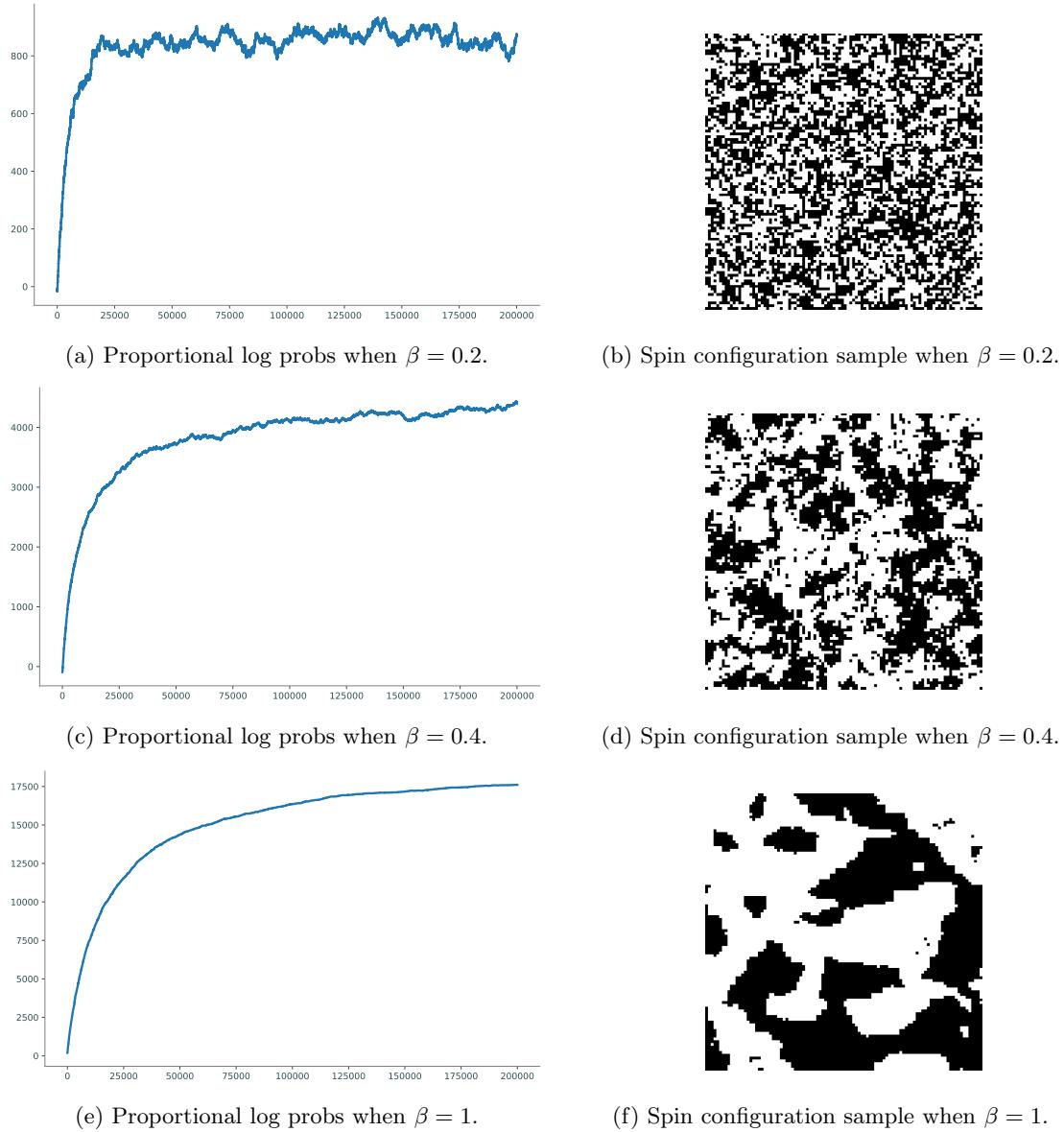


Figure 18.4



# 19

# Gibbs Sampling and LDA

**Lab Objective:** *Understand the basic principles of implementing a Gibbs sampler. Apply this to Latent Dirichlet Allocation.*

## Gibbs Sampling

Gibbs sampling is an MCMC sampling method in which we construct a Markov chain which is used to sample from a desired joint (conditional) distribution

$$\mathbb{P}(x_1, \dots, x_n \mid \mathbf{y}).$$

Often it is difficult to sample from this high-dimensional joint distribution, while it may be easy to sample from the one-dimensional conditional distributions

$$\mathbb{P}(x_i \mid \mathbf{x}_{-i}, \mathbf{y})$$

where  $\mathbf{x}_{-i} = x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ .

---

### Algorithm 1 Basic Gibbs Sampling Process.

---

```

1: procedure GIBBS SAMPLER
2:   Randomly initialize  $x_1, x_2, \dots, x_n$ .
3:   for  $k = 1, 2, 3, \dots$  do
4:     for  $i = 1, 2, \dots, n$  do
5:       Draw  $x \sim \mathbb{P}(x_i \mid \mathbf{x}_{-i}, \mathbf{y})$ 
6:       Fix  $x_i = x$ 
7:      $\mathbf{x}^{(k)} = (x_1, x_2, \dots, x_n)$ 

```

---

A Gibbs sampler proceeds according to Algorithm 1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of  $\mathbf{x}^{(k)}$  after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible  $\mathbf{x}$ . The unique invariant distribution for the chain is the desired joint distribution

$$\mathbb{P}(x_1, \dots, x_n \mid \mathbf{y}).$$

Thus, after a burn-in period, our samples  $\mathbf{x}^{(k)}$  are effectively samples from the desired distribution.

Consider the dataset of  $N$  scores from a calculus exam in the file `examscores.npy`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean  $\mu$  and variance  $\sigma^2$ . Because we are unsure of the true value of  $\mu$  and  $\sigma^2$ , we take a Bayesian approach and place priors on each parameter to quantify this uncertainty:

$$\begin{aligned}\mu &\sim N(\nu, \tau^2) && \text{(a normal distribution)} \\ \sigma^2 &\sim IG(\alpha, \beta) && \text{(an inverse gamma distribution)}\end{aligned}$$

Letting  $\mathbf{y} = (y_1, \dots, y_N)$  be the set of exam scores, we would like to update our beliefs of  $\mu$  and  $\sigma^2$  by sampling from the posterior distribution

$$\mathbb{P}(\mu, \sigma^2 | \mathbf{y}, \nu, \tau^2, \alpha, \beta).$$

Sampling directly can be difficult. However, we *can* easily sample from the following conditional distributions:

$$\begin{aligned}\mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta)\end{aligned}$$

The reason for this is that these conditional distributions are *conjugate* to the prior distributions, and hence are part of the same distributional families as the priors. In particular, we have

$$\begin{aligned}\mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) &\sim N(\nu^*, (\tau^*)^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) &\sim IG(\alpha^*, \beta^*),\end{aligned}$$

where

$$\begin{aligned}(\tau^*)^2 &= \left( \frac{1}{\tau^2} + \frac{N}{\sigma^2} \right)^{-1} \\ \nu^* &= (\tau^*)^2 \left( \frac{\nu}{\tau^2} + \frac{1}{\sigma^2} \sum_{i=1}^N y_i \right) \\ \alpha^* &= \alpha + \frac{N}{2} \\ \beta^* &= \beta + \frac{1}{2} \sum_{i=1}^N (y_i - \mu)^2\end{aligned}$$

Note that  $\nu^*$  and  $(\tau^*)^2$  are *not* samples and are not used to replace  $\mu$  and  $\sigma^2$  themselves; rather, they're parameters of the marginal distribution of  $\mu$  (which happens to also be distributed normally) as shown above.

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling  $\mu$  and sampling  $\sigma^2$  (so using a two dimensional version of Algorithm 1, we would have  $x_1 = \mu$  and  $x_2 = \sigma^2$ ). We can sample from a normal distribution and an inverse gamma distribution as follows:

```
import numpy as np
from scipy.stats import norm
from scipy.stats import invgamma

mu = 0 # the mean
```

```

sigma2 = 9 # the variance
normal_sample = norm.rvs(mu, scale=np.sqrt(sigma))
alpha = 2
beta = 15
invgamma_sample = invgamma.rvs(alpha, scale=beta)

```

Note that when sampling from the normal distribution, we need to set the `scale` parameter to the standard deviation, *not* the variance.

**Problem 1.** Write a function that accepts data  $\mathbf{y}$ , prior parameters  $\nu$ ,  $\tau^2$ ,  $\alpha$ , and  $\beta$ , and an integer  $n$ . Use Gibbs sampling to generate  $n$  samples of  $\mu$  and  $\sigma^2$  for the exam scores problem.

Test your sampler with priors  $\nu = 80$ ,  $\tau^2 = 16$ ,  $\alpha = 3$ , and  $\beta = 50$ , collecting 1000 samples. Plot your samples of  $\mu$  and your samples of  $\sigma^2$  versus the number of samples. They should both converge quickly, so that both plots look like “fuzzy caterpillars”.

We'd like to look at the posterior marginal distributions for  $\mu$  and  $\sigma^2$ . To plot these from the samples, use a kernel density estimator from `scipy.stats`. If our samples of  $\mu$  are called `mu_samples`, then we can do this with the following code.

```

import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import gaussian_kde

mu_kernel = gaussian_kde(mu_samples)
x = np.linspace(min(mu_samples) - 1, max(mu_samples) + 1, 200)
plt.plot(x, mu_kernel(x))
plt.show()

```

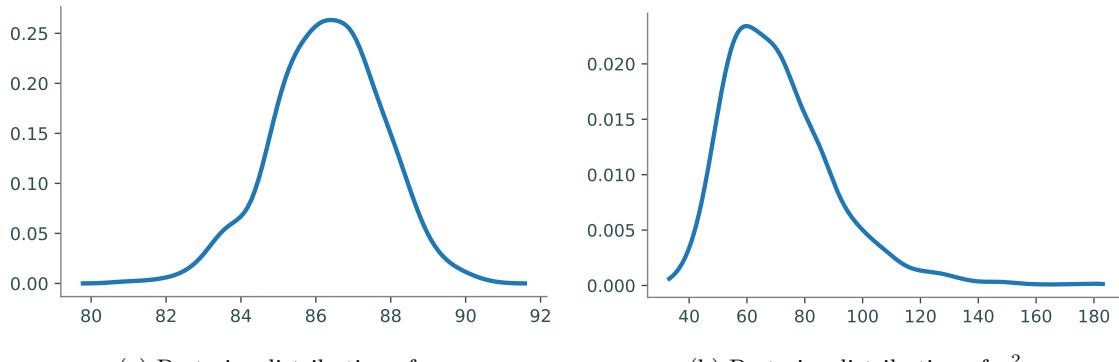


Figure 19.1: Posterior marginal probability densities for  $\mu$  and  $\sigma^2$ .

Keep in mind that the plots above are of the posterior distributions of the *parameters*, not of the scores. If we would like to compute the posterior distribution of a new exam score  $\tilde{y}$  given our data  $\mathbf{y}$  and prior parameters, we compute what is known as the *posterior predictive distribution*:

$$\mathbb{P}(\tilde{y} | \mathbf{y}, \lambda) = \int_{\Theta} \mathbb{P}(\tilde{y} | \Theta) \mathbb{P}(\Theta | \mathbf{y}, \lambda) d\Theta$$

where  $\Theta$  denotes our parameters (in our case  $\mu$  and  $\sigma^2$ ) and  $\lambda$  denotes our prior parameters (in our case  $\nu, \tau^2, \alpha$ , and  $\beta$ ).

Rather than actually computing this integral for each possible  $\tilde{y}$ , we can do this by sampling scores from our parameter samples. In other words, sample

$$\tilde{y}_{(t)} \sim N(\mu_{(t)}, \sigma_{(t)}^2)$$

for each sample pair  $\mu_{(t)}, \sigma_{(t)}^2$ . Now we have essentially drawn samples from our posterior predictive distribution, and we can use a kernel density estimator to plot this distribution from the samples.

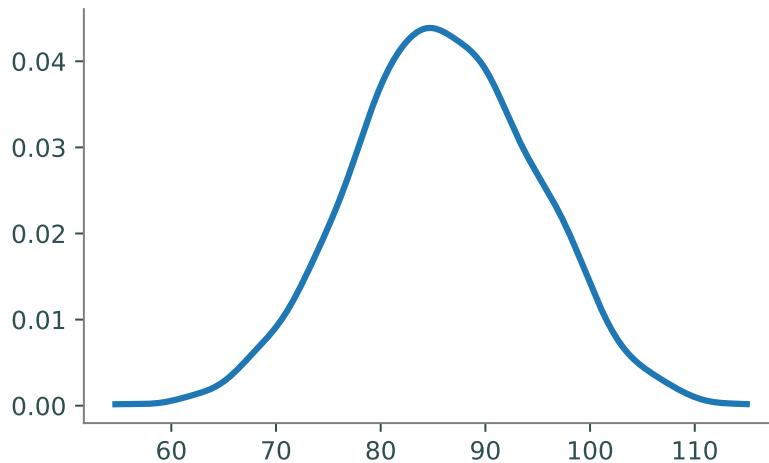


Figure 19.2: Predictive posterior distribution of exam scores.

**Problem 2.** Plot the kernel density estimators for the posterior distributions of  $\mu$  and  $\sigma^2$ . You should get plots similar to those in Figure 19.1.

Next, use your samples of  $\mu$  and  $\sigma^2$  to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. Compare your plot to Figure 19.2.

## Latent Dirichlet Allocation

Gibbs sampling can be applied to an interesting problem in natural language processing (NLP): determining which topics are prevalent in a document. *Latent Dirichlet Allocation* (LDA) is a generative model for a collection of text documents. It supposes that there is some fixed vocabulary (composed of  $V$  distinct terms) and  $K$  different topics, each represented as a probability distribution  $\phi_k$  over the vocabulary, each with a Dirichlet prior  $\beta$ . This means  $\phi_{k,v}$  is the probability that topic  $k$  is represented by vocabulary term  $v$ .

With the vocabulary and topics chosen, the LDA model assumes that we have a set of  $M$  documents (each “document” may be a paragraph or other section of the text, rather than a “full” document). The  $m$ -th document consists of  $N_m$  words, and a probability distribution  $\theta_m$  over the topics is drawn from a Dirichlet distribution with parameter  $\alpha$ . Thus  $\theta_{m,k}$  is the probability that document  $m$  is assigned label  $k$ . If  $\phi_{k,v}$  and  $\theta_{m,k}$  are viewed as matrices, their rows sum to one.

We will now iterate through each document in the same manner. Assume we are working on document  $m$ , which you will recall contains  $N_m$  words. For word  $n$ , we first draw a topic assignment  $z_{m,n}$  from the categorical distribution  $\theta_m$ , and then we draw a word  $v$  from the categorical distribution  $\phi_{z_{m,n}}$ . Throughout this implementation, we assume  $\alpha$  and  $\beta$  are scalars<sup>1</sup>. In summary, we have

1. Draw  $\phi_k \sim \text{Dir}(\beta)$  for  $1 \leq k \leq K$ .

2. For  $1 \leq m \leq M$ :

(a) Draw  $\theta_m \sim \text{Dir}(\alpha)$ .

(b) Draw  $z_{m,n} \sim \text{Cat}(\theta_m)$  for  $1 \leq n \leq N_m$ .

(c) Draw  $v \sim \text{Cat}(\phi_{z_{m,n}})$  for  $1 \leq n \leq N_m$ .

We end up with  $n$  words which represent document  $m$ . Note that these words are *not* necessarily distinct from one another; indeed, we are most interested in the words that have been repeated the most.

This is typically depicted with graphical plate notation as in Figure 19.3.

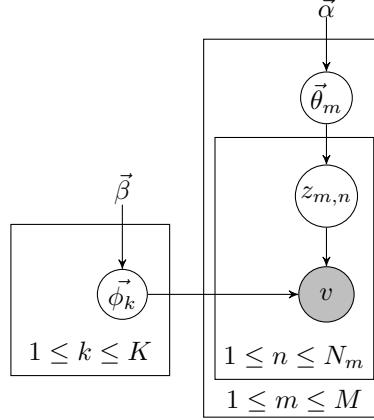


Figure 19.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables  $v$  are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each  $\phi_k$  and each  $\theta_m$ . This will allow us to understand what each topic is, as well as understand how each document is distributed over the  $K$  topics. In other words, we want to predict the topic of each document, and also which words best represent this topic. We can estimate these well if we know  $z_{m,n}$  for each  $m, n$ , collectively referred to as  $\mathbf{z}$ . Thus, we need to sample  $\mathbf{z}$  from the posterior distribution  $\mathbb{P}(\mathbf{z} \mid \mathbf{v}, \alpha, \beta)$ , where  $\mathbf{v}$  is the collection of words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting  $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$  (so as to condition on everything except the  $(m, n)$ -th entry), the conditional posterior distributions

$$\mathbb{P}(z_{m,n} = k \mid \mathbf{z}_{-(m,n)}, \mathbf{v}, \alpha, \beta)$$

have nice, closed form solutions, making them easy to sample from.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k \mid \mathbf{z}_{-(m,n)}, \mathbf{v}, \alpha, \beta) \propto \frac{\left( n_{(k,m,\cdot)}^{-(-m,n)} + \alpha \right) \left( n_{(k,\cdot,v)}^{-(-m,n)} + \beta \right)}{n_{(k,\cdot,\cdot)}^{(-m,n)} + V\beta}$$

where

- $n_{(k,m,\cdot)} =$  the number of words in document  $m$  assigned to topic  $k$
- $n_{(k,\cdot,v)} =$  the number of times term  $v$  is assigned to topic  $k$
- $n_{(k,\cdot,\cdot)} =$  the number of times topic  $k$  is assigned in the corpus
- $n_{(k,m,\cdot)}^{-(-m,n)} = n_{(k,m,\cdot)} - \mathbf{1}_{z_{m,n}=k}$
- $n_{(k,\cdot,v)}^{-(-m,n)} = n_{(k,\cdot,v)} - \mathbf{1}_{z_{m,n}=k}$
- $n_{(k,\cdot,\cdot)}^{-(-m,n)} = n_{(k,\cdot,\cdot)} - \mathbf{1}_{z_{m,n}=k}$

---

<sup>1</sup>The Dirichlet distribution  $\text{Dir}(x_1, \dots, x_s, \alpha_1, \dots, \alpha_s)$  usually requires the parameter  $\alpha$  to be a vector of length  $s$ , but when  $\alpha$  is a scalar, it is called the “concentration parameter” and behaves like a vector of length  $s$  whose entries are all equal to  $\alpha$ .

Thus, if we simply keep track of these count matrices, then we can easily create a Gibbs sampler over the topic assignments. This is actually a particular class of samplers known as *collapsed Gibbs samplers*, because we have collapsed the sampler by integrating out  $\theta$  and  $\phi$ .

We have provided for you the structure of a Python object LDACGS with several methods, listed at the end of this lab. The object defines attributes `n_topics`, `alpha`, and `beta` upon initialization. The method `buildCorpus()` then defines attributes `vocab` and `documents`, where `vocab` is a list of strings (terms), and `documents` is a list of dictionaries (a dictionary for each document). For dictionary  $m$  in `documents`, each entry is of the form  $n : v$ , where  $v$  is the index in `vocab` of the  $n^{th}$  word in document  $m$ .

The remainder of this lab will guide you through writing several more methods in order to implement the Gibbs sampler. The first step is to initialize the assignments and create count matrices  $n_{(k,m,:)}, n_{(k,:v,:)} \text{ and vector } n_{(k,:,:)}$ .

**Problem 3.** Complete the method `_initialize()` to initialize as attributes `n_words`, `n_docs`, the three count matrices, and the topic assignment dictionary `topics`.

To do this, you will need to initialize `nkm`, `nkv`, and `nk` to be zero arrays of the correct size. Matrix `nkm` corresponds to  $n_{(k,m,:)}$ , `nkv` to  $n_{(k,:v,:)}$ , and `nk` to  $n_{(k,:,:)}$ . You will then iterate through each word found in each document. In the second of these for-loops (for each word), you will randomly assign `k` as an integer from the correct range of topics. Then, you will increment each of the count matrices by 1, given the values for `k`, `m`, and `v`, where `v` is the index in `vocab` of the  $n^{th}$  word in document  $m$ . Finally, assign `topics` as given.

The next method fully outlines a sweep of the Gibbs sampler.

**Problem 4.** Complete the method `_sweep()`.

To do this, iterate through each word of each document. The first part of this method will undo what `_initialize()` did by decrementing each of the count matrices by 1. Then, call the method `_conditional()` to use the conditional distribution (instead of the uniform distribution used previously) to pick a more accurate topic assignment `k`. Finally, repeat what `_initialize()` did by incrementing each of the count matrices by 1, but this time using the more accurate topic assignment.

You are now prepared to write the full Gibbs sampler.

**Problem 5.** Complete the method `sample()`. The argument `filename` is the name and location of a .txt file, which can be read in by the provided method `buildCorpus()` to build the corpus. Stopwords are removed if the `stopwords` argument is provided. Note that in `buildCorpus()`, each line of `filename` is considered a document.

Initialize attributes `total_nkm`, `total_nkv`, and `logprobs` as zero arrays. `total_nkm` and `total_nkv` will be the sums of every `sample_rateth` `nkm` and `nkv` matrix respectively. `logprobs` is of length `burnin + sample_rate * n_samples` and will store each log-likelihood after each sweep of the sampler.

Burn-in the Gibbs sampler. After the burn-in, iterate further for `n_samples` iterations, adding `nkm` and `nkv` to `total_nkm` and `total_nkv` respectively, at every `sample_rateth` iteration. Also, compute and save the log-likelihood at each iteration in `logprobs` using the method `_loglikelihood()`.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on some of Ronald Reagan's State of the Union addresses, found in `reagan.txt`. Note that in `reagan.txt`, each line is an entire paragraph from one of Reagan's addresses, so your Gibbs sampler will consider each paragraph as a separate document.

**Problem 6.** Create an LDACGS object with 20 topics, letting  $\alpha$  and  $\beta$  be the default values. Run the Gibbs sampler, with a burn-in of 100 iterations, accumulating 10 samples, only keeping the results of every 10th sweep with the `reagan.txt` file. Use `stopwords.txt` as the stopwords file.

Plot the log-likelihoods. How many iterations did it take to burn-in?

Make sure to save the LDACGS object, it will be used in the next problem.

We can estimate the values of each  $\phi_k$  and each  $\theta_m$  as follows:

$$\hat{\phi}_{k,v} = \frac{n_{(k, \cdot, v)} + \beta}{V \cdot \beta + \sum_{v=1}^V n_{(k, \cdot, v)}}$$

$$\hat{\theta}_{m,k} = \frac{n_{(k, m, \cdot)} + \alpha}{K \cdot \alpha + \sum_{k=1}^K n_{(k, m, \cdot)}}$$

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions  $\phi_k$  by looking at the  $n$  terms with the highest probability, where  $n$  is small (say 10 or 20). We have provided a method `topterms` which does this for you.

**Problem 7.** Using the method `topterms()`, examine the topics for Reagan's addresses. If `n_topics=20` and `n_samples=10`, you should get the top 10 words that represent each of the 20 topics. Print out all 20 topics associated 10 words. For the top 5 topics, decide what their top 10 words jointly represent, and come up with a label for them.

We can use  $\hat{\theta}$  to find the documents (paragraphs) in Reagan's addresses that focus the most on each topic. The documents with the highest values of  $\hat{\theta}_k$  are those most heavily focused on topic  $k$ . For example, if you chose the topic label for topic  $p$  to be *the Cold War*, you can find the five highest values in  $\hat{\theta}_p$ , which will tell you which five documents (paragraphs) are most centered on the Cold War. For your convenience, the provided method `toplins()` accomplishes just that by printing out the top `n_lines` documents corresponding to each topic.

Let's take a moment to see what our Gibbs sampler has accomplished. By simply feeding in a group of documents, and with no human input, we have found the most common topics discussed, which are represented by the words most frequently used in relation to that particular topic. The only work that the user has done is to assign topic labels, saying what the words in each group have in common. As you may have noticed, however, these topics may or may not be *relevant* topics. You might have noticed that some of the most common topics were simply English particles (words such as *a*, *the*, *an*) and conjunctions (*and*, *so*, *but*). Industrial grade packages can effectively remove such topics so that they are not included in the results.

## Additional Material

### LDACGS Source Code

```

class LDACGS:
    """ Do LDA with Gibbs Sampling. """

    def __init__(self, n_topics, alpha=0.1, beta=0.1):
        """ Initializes attributes n_topics, alpha, and beta. """
        self.n_topics = n_topics
        self.alpha = alpha
        self.beta = beta

    def _buildCorpus(self, filename, stopwords_file=None):
        """ Reads the given filename, and using any provided stopwords,
            initializes attributes vocab and documents. In this lab,
            each line of filename is considered a document.

            vocab is a list of terms found in filename.

            documents is a list of dictionaries (a dictionary for each
            document); for dictionary m in documents, each entry is of
            the form n:v, where v is the index in vocab of the nth word
            in document m.
        """
        with open(filename, 'r') as infile: # Create vocab
            doclines = [line.rstrip().lower().split(' ') for line in infile]
        n_docs = len(doclines)
        self.vocab = list({v for doc in doclines for v in doc})

        self.docs = doclines # Save the documents for toplines()

        if stopwords_file: # If there are stopwords, remove them from vocab
            with open(stopwords_file, 'r') as stopfile:
                stops = stopfile.read().split()
            self.vocab = [x for x in self.vocab if x not in stops]
            self.vocab.sort()

        self.documents = [] # Create documents
        for i in range(n_docs):
            self.documents.append({})
            for j in range(len(doclines[i])):
                if doclines[i][j] in self.vocab:
                    self.documents[i][j] = self.vocab.index(doclines[i][j])

    def _initialize(self):
        """ Initializes attributes n_words, n_docs, the three count matrices,
            and the topic assignment dictionary topics.
    
```

```

Note that
n_topics = K, the number of possible topics
n_docs   = M, the number of documents being analyzed
n_words  = V, the number of words in the vocabulary

To do this, you will need to initialize nkm, nk, and nk
to be zero arrays of the correct size.
Matrix nkm corresponds to n_(k,m,.)
Matrix nk corresponds to n_(k,..v)
Matrix nk corresponds to n_(k,.,.)
You will then iterate through each word found in each document.
In the second of these for-loops (for each word), you will
randomly assign k as an integer from the correct range of topics.
Then, you will increment each of the count matrices by 1,
given the values for k, m, and v, where v is the index in
vocab of the nth word in document m.
Finally, assign topics as given.

"""
self.n_words = len(self.vocab)
self.n_docs = len(self.documents)

# Initialize the three count matrices
# The (k, m) entry of self.nkm is the number of words in document m ←
# assigned to topic k
self.nkm = np.zeros((self.n_topics, self.n_docs))
# The (k, v) entry of self.nkv is the number of times term v is ←
# assigned to topic k
self.nkv = np.zeros((self.n_topics, self.n_words))
# The (k)-th entry of self.nk is the number of times topic k is ←
# assigned in the corpus
self.nk = np.zeros(self.n_topics)

# Initialize the topic assignment dictionary
self.topics = {} # Key-value pairs of form (m,n):k

random_distribution = np.ones(self.n_topics) / self.n_topics
for m in range(self.n_docs):
    for n in self.documents[m]:
        # Get random topic assignment, i.e. k = ...
        # Increment count matrices
        # Store topic assignment, i.e. self.topics[(m,n)]=k
        raise NotImplementedError("Problem 3 Incomplete")

def _sweep(self):
    """ Iterates through each word of each document, giving a better
    topic assignment for each word.

```

```

To do this, iterate through each word of each document.
The first part of this method will undo what _initialize() did
by decrementing each of the count matrices by 1.
Then, call the method _conditional() to use the conditional
distribution (instead of the uniform distribution used
previously) to pick a more accurate topic assignment k.
Finally, repeat what _initialize() did by incrementing each of
the count matrices by 1, but this time using the more
accurate topic assignment.

"""

for m in range(self.n_docs):
    for n in self.documents[m]:
        # Retrieve vocab index for n-th word in document m
        # Retrieve topic assignment for n-th word in document m
        # Decrement count matrices
        # Get conditional distribution
        # Sample new topic assignment
        # Increment count matrices
        # Store new topic assignment
        raise NotImplementedError("Problem 4 Incomplete")

def sample(self, filename, burnin=100, sample_rate=10, n_samples=10, ←
          stopwords_file=None):
    """ Runs the Gibbs sampler on the given filename.

    The argument filename is the name and location of a .txt
    file, which can be read in by the provided method _buildCorpus()
    to build the corpus. Stopwords are removed if the stopwords
    argument is provided. Note that in buildCorpus(),
    each line of filename is considered a document.

    Initialize attributes total_nkm, total_nkv, and logprobs as
    zero arrays.
    total_nkm and total_nkv will be the sums of every
    sample_rate-th nkm and nkv matrix respectively.
    logprobs is of length burnin + sample_rate * n_samples
    and will store each log-likelihood after each sweep of
    the sampler.

    Burn-in the Gibbs sampler. After the burn-in, iterate further
    for n_samples iterations, adding nkm and nkv to total_nkm and
    total_nkv respectively, at every sample_rate-th iteration.
    Also, compute and save the log-likelihood at each iteration
    in logprobs using the method _loglikelihood().
    """

    self._buildCorpus(filename, stopwords_file)
    self._initialize()

```

```

self.total_nkm = np.zeros((self.n_topics, self.n_docs))
self.total_nkv = np.zeros((self.n_topics, self.n_words))
self.logprobs = np.zeros(burnin + sample_rate * n_samples)

for i in range(burnin):
    # Sweep and store log likelihood
    raise NotImplementedError("Problem 5 Incomplete")
for i in range(sample_rate * n_samples):
    # Sweep and store log likelihood
    raise NotImplementedError("Problem 5 Incomplete")
    if not i % sample_rate:
        # Accumulate counts
        raise NotImplementedError("Problem 5 Incomplete")


def _conditional(self, m, v):
    """ Returns the conditional distribution given m and w.
        Called by _sweep(). """
    dist = (self.nkm[:, m] + self.alpha) * (self.nkv[:, v] + self.beta) / ((←
        self.nk + self.beta * self.n_words)
    return dist / np.sum(dist)

def _loglikelihood(self):
    """ Computes and returns the log-likelihood. Called by sample(). """
    lik = 0

    for k in range(self.n_topics):
        lik += np.sum(gammaln(self.nkv[k, :] + self.beta)) - gammaln(np.sum←
            (self.nkv[k, :] + self.beta))
        lik -= self.n_words * gammaln(self.beta) - gammaln(self.n_words * ←
            self.beta)

    for m in range(self.n_docs):
        lik += np.sum(gammaln(self.nkm[:, m] + self.alpha)) - gammaln(np.←
            sum(self.nkm[:, m] + self.alpha))
        lik -= self.n_topics * gammaln(self.alpha) - gammaln(self.n_topics * ←
            self.alpha)

    return lik

def _phi(self):
    """ Initializes attribute phi. Called by topterm(). """
    phi = self.total_nkv + self.beta
    self.phi = phi / np.sum(phi, axis=1)[:, np.newaxis]

def _theta(self):
    """ Initializes attribute theta. Called by toplines(). """
    theta = self.total_nkm + self.alpha
    self.theta = theta / np.sum(theta, axis=1)[:, np.newaxis]

```

```
def toterms(self, n_terms=10):
    """ Returns the top n_terms of each topic found. """
    self._phi()
    vec = np.atleast_2d(np.arange(0, self.n_words))
    topics = []
    for k in range(self.n_topics):
        probs = np.atleast_2d(self.phi[k,:])
        mat = np.append(probs, vec, 0)
        sind = np.array([mat[:, i] for i in np.argsort(mat[0])]).T
        topics.append([self.vocab[int(sind[1, self.n_words - 1 - i])] for i in range(n_terms)])
    return topics

def toplines(self, n_lines=5):
    """ Print the top n_lines corresponding to each topic found. """
    self._theta()
    lines = np.zeros((self.n_topics, n_lines))
    for k in range(self.n_topics):
        args = np.argsort(self.theta[:, k]).tolist()
        args.reverse()
        lines[k, :] = np.array(args)[0:n_lines] + 1
    lines = lines.astype(int)

    for k in range(self.n_topics):
        print(f"TOPIC {k + 1}")
        for document in lines[k]:
            print(' '.join(self.docs[document]))
```

# 20 Gaussian Mixture Models

**Lab Objective:** *Understand the formulation of Gaussian Mixture Models (GMMs) and use the Expectation Maximization algorithm to estimate GMM parameters.*

Mixture models are a useful way to combine distributions together that allows us to describe much more complicated distributions than using just the standard list of named distributions. The essential idea of a mixture model is in its name: it is a mixture of several different models, or probability distributions. Each of these model is called a *component*. Each component has a certain probability associated with it, called its *weight*, that describes how likely it is for a sample from the model to come from that component. We denote the weight of the  $i$ -th component as  $w_i$ .

In this lab, we focus on *Gaussian Mixture Models*, or GMMs for short. In a GMM, each component is a multivariate Gaussian (normal) distribution. Each of these is parameterized by a mean  $\mu_i$  and a covariance matrix  $\Sigma_i$ .

A GMM with  $K$  components thus has parameters  $\theta = (w_1, \dots, w_K, \mu_1, \dots, \mu_K, \Sigma_1, \dots, \Sigma_K)$ . We can use the law of total probability to evaluate the density of a GMM, which is given by

$$P(z|\theta) = \sum_{k=1}^K w_k \mathcal{N}(z|\mu_k, \Sigma_k)$$

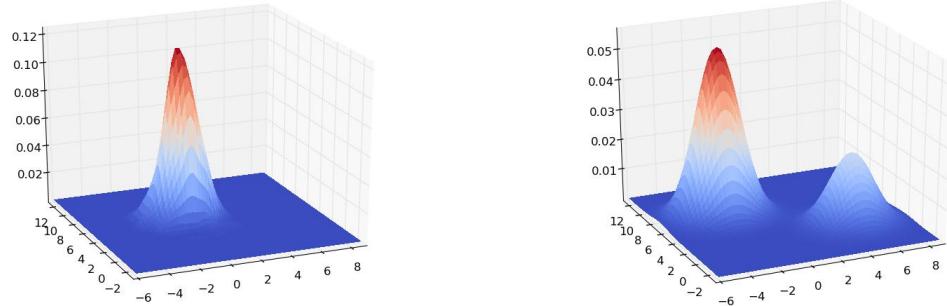
where

$$\mathcal{N}(z|\mu, \Sigma) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2} (z - \mu)^T \Sigma^{-1} (z - \mu)\right)$$

is the density function of a multivariate normal distribution.

It is important to keep in mind that a GMM does *not* arise from adding weighted multivariate normal random variables, but rather from weighting the responsibility of each multivariate normal random variable. The first case simply results in a different multivariate normal distribution. Refer to Figure 20.1 for a visualization of these two cases.

**Problem 1.** Throughout this lab, we will build a GMM class with various methods. Write the `__init__` method for this class. It should accept a parameter for the number of components and optional parameters for the weights, means, and covariance matrices which define the GMM, and store these.<sup>a</sup>



(a) Sum of weighted multivariate normal random variables. (b) Weighted mixture of multivariate normal random variables.

Figure 20.1

If we have  $K$  components and  $d$  dimensions, then the weights should have shape  $(K,)$ , the means  $(K, d)$ , and the covariances  $(K, d, d)$ . The parameters for the  $k$ -th component can be found as `weights[k]`, `means[k]`, `covars[k]`.

<sup>a</sup>If we don't have a good guess for the parameters of the GMM to pass into the class, it makes more sense to initialize these from the dataset we are training on, which we will do later in the `fit` method; hence, we let the parameters be optional here.

**Problem 2.** Write a method `component_logpdf` for your class that accepts a component  $k$  and a point  $z$  and computes

$$\log w_k + \log \mathcal{N}(z|\mu_k, \Sigma_k),$$

the logarithm of the contribution of the  $k$ -th component of the pdf. Also write a method `pdf` that accepts a point  $z$  and returns the probability density of the whole GMM at that point.

Hint: `scipy.stats.multivariate_normal.pdf` and `scipy.stats.multivariate_normal.logpdf` can be used to efficiently evaluate the multivariate normal pdf.

We will use the following initialization to test the next several problems. The initialization code has been provided for you in the 'Check Section' as the function 'init\_gmm':

```
gmm = GMM(n_components = 2,
           weights = np.array([0.6, 0.4]),
           means = np.array([[[-0.5, -4.0], [0.5, 0.5]]]),
           covars = np.array([
               [[1, 0], [0, 1]],
               [[0.25, -1], [-1, 8]],
           ]))
```

Your functions should give the following output:

```
>>> gmm.pdf(np.array([1.0, -3.5]))
0.05077912539363083
# Component 0
>>> gmm.component_logpdf(0, np.array([1.0, -3.5]))
-3.598702690175336
# Component 1
>>> gmm.component_logpdf(1, np.array([1.0, -3.5]))
-3.7541677982835004
```

Note that since this GMM is 2-dimensional, the input point must be an array of length 2.

In order to get credit for the next few problems, you must write tests in the 'Check Section' that show your functions give the proper outputs. The check for problem 2 has been written for you as an example.

In order to draw a value from a mixture model, we must first draw a variable  $X \sim \text{Cat}(w_1, \dots, w_K)$  that represents which component the sample comes from. We can then draw the sample  $Z \sim \mathcal{N}(\mu_X, \Sigma_X)$ . If we want to draw multiple samples, we need to repeat this process for each one (draw an  $X$  and then draw a  $Z$ ).

**Problem 3.** Write a method `draw` for the GMM class that randomly draws from the model. If  $m$  points are drawn and the GMM is  $d$ -dimensional, the returned array should have shape  $(m, d)$ .

The function '`check_problem3()`' is also provided. It will test your draw function by plotting your sample draw against your `pdf` function. If done correctly, running the check function should make 2 plots that look "good".

We now consider how to estimate the parameters of a GMM given some observed data  $Z = z_1, \dots, z_n$ . Ordinarily, a good approach would be to try to directly maximize the log-likelihood

$$l(\theta) = \sum_{i=1}^n \log \sum_{j=1}^K w_j \mathcal{N}(z_i | \mu_j, \Sigma_j).$$

However, this expression is very difficult to deal with using standard optimization methods, particularly because of the sum inside of the logarithm. A good alternative in this case is the *expectation maximization* (EM) algorithm. This is an iterative algorithm, where each step consists of maximizing a function that is designed to approximate the log-likelihood while being much easier to maximize.

Each iteration consists of two steps, the E-step and the M-step. Suppose our estimated parameters at the  $t$ -th iteration are  $\theta^t = (w_1^t, \dots, w_K^t, \mu_1^t, \dots, \mu_K^t, \Sigma_1^t, \dots, \Sigma_K^t)$ . Note that  $t$  is an index, not an exponent. For each data point  $z_i, 1 \leq i \leq n$  and each component  $1 \leq k \leq K$ , the E-step consists of computing

$$\begin{aligned} q_i^t(k) &= P(X_i = k | z_i, \theta^t) \\ &= \frac{P(z_i | X_i = k, \theta^t)}{P(z_i | \theta^t)} \\ &= \frac{w_k^t \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^K w_{k'}^t \mathcal{N}(z_i | \mu_{k'}^t, \Sigma_{k'}^t)} \end{aligned}$$

In order to accurately compute this quantity, however, we need to be more careful. It is possible that due to floating point underflow<sup>1</sup> that each term  $w_{k'}^t \mathcal{N}(z_i | \mu_{k'}, \Sigma_{k'})$  in the sum in the denominator becomes zero, which is a major problem. This particularly happens if the exponents in the multivariate normal densities all are large negative numbers. To avoid this problem, we can rescale the numerator and denominator. Let

$$\ell_{i,k} = \log w_k^t + \log \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t),$$

the logarithm of each term in the denominator. For each data point  $z_i$ , we can find

$$L_i = \max_{k'} \ell_{i,k'},$$

the largest of these logarithms. Then, we can rewrite the quantity we want to calculate as

$$\begin{aligned} q_i^t(k) &= \frac{w_k^t \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^K w_{k'}^t \mathcal{N}(z_i | \mu_{k'}^t, \Sigma_{k'}^t)} \\ &= \frac{e^{\ell_{i,k}}}{\sum_{k'=1}^K e^{\ell_{i,k'}}} \\ &= \frac{e^{\ell_{i,k}} e^{-L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'}} e^{-L_i}} \\ &= \frac{e^{\ell_{i,k} - L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'} - L_i}}. \end{aligned}$$

This rescaling makes the largest term in the denominator equal to 1, so computing  $q_i^t(k)$  in this way avoids underflow problems. Note that for the computation of any individual  $q_i^t(k)$ , the value  $L_i$  is a scalar that is the same for all components; however, you will have as many of these values as you have data points. As a reminder,  $i$  corresponds to the index of a data point and  $k$  corresponds to which component we are comparing it to.

**Problem 4.** Write a method `_compute_e_step` that calculates the  $q_i^t(k)$  as given by the E-step, given a collection of observations. Be sure to do the calculation in a way that avoids underflow, and use array broadcasting when possible.

Your method will accept an array of shape `(n, d)`, where `n` is the number of data points and `d` is the dimensionality of the data (i.e. each row is a data point). The array you produce should have shape `(n_components, n)` where `result[k, i] = q_i^t(k)` (i.e. each row is one component, and each column is a data point). The various intermediate values should have shapes similar to the following:

- The array of  $\ell_{i,k}$ s should have shape `(n_components, n)`
- The array of  $L_i$ s should have shape `(n, )`
- The array of the denominator values  $\sum_{k'=1}^K e^{\ell_{i,k'} - L_i}$  should also have shape `(n, )`

With the GMM from the example in Problem 2, you should get the following results:

---

<sup>1</sup>As a refresher, one way that floating point numbers are limited is that they cannot represent positive numbers arbitrarily close to zero; at some point, if the number in a computation becomes too small, the computer is forced to round it to zero, which is called *underflow*. The threshold is about  $10^{-323}$  for the 64-bit floating point numbers used in python. Even if underflow does not occur, very small floating points have greatly reduced precision, so it is generally good to avoid using them.

```
>>> data = np.array([
    [0.5, 1.0],
    [1.0, 0.5],
    [-2.0, 0.7]
])
>>> gmm._compute_e_step(data)
array([[3.49810771e-06, 5.30334386e-05, 9.99997070e-01],
       [9.99996502e-01, 9.99946967e-01, 2.93011749e-06]])
```

Complete the `check_problem4()` function in the Check Section to show your GMM gets the correct result.

Now that we have the  $q_i^t(k)$ , we can perform the M-step. This step consists of maximizing the function

$$Q^t(\theta) = \sum_{i=1}^n \sum_{k=1}^K q_i^t(k) \log w_k^t \mathcal{N}(z_i | \mu_k, \Sigma_k)$$

We then set

$$\theta^{t+1} = \operatorname{argmax}_{\theta} Q^t(\theta)$$

and iterate until the method appears to converge. In the case of GMMs, the maximizer  $\theta^{t+1}$  of  $Q^t(\theta)$  is given by

$$\begin{aligned} w_k^{t+1} &= \frac{1}{n} \sum_{i=1}^n q_i^t(k) \\ \mu_k^{t+1} &= \frac{\sum_{i=1}^n q_i^t(k) z_i}{\sum_{i=1}^n q_i^t(k)} \\ \Sigma_k^{t+1} &= \frac{\sum_{i=1}^n q_i^t(k) (z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^n q_i^t(k)} \end{aligned}$$

For details on the derivation of the maximizer, refer to the Volume 3 textbook.

**Problem 5.** Write a method `_compute_m_step` for your GMM class that takes the output of Problem 4 as the  $q_i^t(k)$  values and calculates the next iteration of weights and means using the formulas above. Be sure to use array broadcasting when possible. Coding the formula for the sigma update requires a rather involved approach using devious array broadcasting and a very useful function, `np.einsum`.<sup>a</sup>

```
# This gets the centered observations for the numerator
# by subtracting them as per the formula
obs_centered = np.expand_dims(Z, 0) - np.expand_dims(new_means, 1)
# This function creates the rest of the numerator and uses array
# broadcasting to implement the denominator. The K, n, d, and D tell
# np.einsum which axes to multiply and which to sum
new_covars =
```

```
np.einsum('Kn,Knd,KnD->KdD', q_values, obs_centered, obs_centered) / ←
    q_sum.reshape(-1, 1, 1)
```

The code above has been provided in the spec file, but a general understanding of the method will be useful in a variety of other situations. Return the updated parameters (weights, means, and covariance matrices).

With the same GMM and data as in Problem 4, you should get the following results:

```
>>> gmm._compute_m_step(data)
(array([0.3333512, 0.6666488]),
 array([[[-1.99983216, 0.69999044],
        [ 0.74998978, 0.75000612]]]),
 array([[[[ 4.99109197e-04, -2.91933135e-05],
        [-2.91933135e-05, 2.43594533e-06]],

       [[ 6.25109881e-02, -6.24997069e-02],
        [-6.24997069e-02, 6.24999121e-02]]]))
```

finish writing `check_problem5()` to show you have the correct outputs

<sup>a</sup>For a detailed explanation of `np.einsum`, see the Additional Materials section of the Advanced Numpy lab.

### Problem 6. Write a `fit` method for your GMM class.

First, if the GMM's parameters are uninitialized (set to `None`), initialize the parameters of the components. We want to do this in a way that the algorithm starts with reasonable values for the dataset. A good way to initialize the means is to randomly select points from the dataset. The covariance matrices can be initialized as diagonal matrices based on the variance of the data. Ensure that the weights you choose add up to 1.

Then, perform the expectation maximization algorithm. Use the functions you created in Problems 4 and 5 to calculate the parameters at each step. Repeat until the parameters converge. Use the following to measure the change in the parameters with each iteration:

```
change = (np.max(np.abs(new_weights - old_weights))
          + np.max(np.abs(new_means - old_means))
          + np.max(np.abs(new_covars - old_covars)))
```

The file `gmm_data.npy` contains a collection of data drawn from a two-dimensional GMM. Finish `check_problem6()` by creating a variable named `gmm` that is a GMM object initialized with 3 components and fitted to the data from `gmm_data.npy`.

The check function will plot the pdf of your trained gmm and a hexbin plot of the training data. They should look very similar to each other. Additionally, your class should take less than 15 seconds to train on the dataset. The check code will time your fit function and print the training time to make sure you are within that range.

## Clustering with GMMs

An important use of mixture models is for *clustering*. The objective of clustering is to take an unlabeled dataset and separate it into some number of clusters, which can then be labeled. This is an instance of *unsupervised learning*, as it is a machine learning task where the training algorithm does not need the true answers (in this case, the actual clusters).

In order to cluster a dataset using a GMM, we first need to train the GMM on that data. Then, we can assign each point a label by finding which component has the largest contribution to the pdf there. Written symbolically, for a data point  $z$ , we have

$$\text{Cluster}(z) = \operatorname{argmax}_k w_k \mathcal{N}(z|\mu_k, \Sigma_k).$$

Note that the number of clusters (components) is a hyperparameter that must be selected before a GMM is trained. In general, cross-validation or some other method must be used to find the right number of clusters.

**Problem 7.** Write a `predict` method for your class. Given a set of data points, return which cluster has the highest pdf density for each data point.

The file `classification.npz` contains a set of 3-dimensional data points (`X`) and their labels (`y`). Use your class with `n_components=4` to cluster the data. Plot the points with the predicted and actual labels, and compute and return your model's accuracy. Your class should take less than 30 seconds to train on this dataset. Make sure to time your `.fit()` or `.fit_predict()` and print the time spent in training to receive credit.

Note that the labels may be permuted; for instance, your model might cluster the points correctly, but swap the labels of clusters 1 and 2 compared to the true labels. The model would still be considered accurate in this case; we only care what the clusters are, not how the model labels them. To resolve this problem, we need to find the permutation of the labels that results in the highest accuracy. The following function does this in a way that is more efficient than directly checking all permutations:

```
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import confusion_matrix

def get_accuracy(pred_y, true_y):
    """
    Helper function to calculate the actually clustering accuracy,
    accounting for the possibility that labels are permuted.
    """
    # Compute confusion matrix
    cm = confusion_matrix(pred_y, true_y)
    # Find the arrangement that maximizes the score
    r_ind, c_ind = linear_sum_assignment(cm, maximize=True)
    return np.sum(cm[r_ind, c_ind]) / np.sum(cm)
```

For convenience, a method `fit_predict` for the class is also included in the specifications file that calls both `fit` and `predict` to make the clustering process simpler.

Clustering with GMMs is closely related to the K-means algorithm. In fact, K-means can be viewed as a special case of GMMs where the covariance matrices are all the identity. How might this affect its ability to cluster? We now compare the effectiveness of GMMs for classification on this dataset with K-means, as well as comparing to sklearn's implementation.

**Problem 8.** The function method `_comparison` initializes an instance of your GMM as well as a GMM and K-means object from Sklearn and compares the 3 based on accuracy and time to train. Run the function and observe the results, then answer in the markdown cell below why K-Means might have performed worse than the GMM's.

You may also find it interesting that sklearn's GMM is actually faster on this dataset than K-means despite GMM's being more complicated to train. This is in part because the dataset is rather low-dimensional. As the dimension of the dataset grows, GMMs suffer computationally from the curse of dimensionality much more than the K-means algorithm.

## Additional Materials

### Jax

Jax is a combination of both Autograd and XLA (Accelerated Linear Algebra) to provide high performance computations. It is a tool that can automatically differentiate various Python and NumPy code including `if` statement, `for` loops, recursion, and other native code types.

Jax provides a Numpy-like API to build machine learning models. Jax can only run on GPUs and TPUs which makes it more efficient than NumPy, which can only run on a CPU. The three main Jax functions include `jit`, `grad` and `vmap`.

- `jit`: Adding `@jax.jit` to the beginning of a function creates an optimized version of the function.
- `grad`: Used to compute the gradient or derivative of a function.
- `vmap`: This is used to vectorize your functions. Since list comprehension isn't available using Jax's version of NumPy, this can be helpful and used instead.

### Dynamax

Dynamax is a library that uses Jax for probabilistic state space models (SSMs). The SSMs that Dynamax is able to compute include Hidden Markov Models (HMMs), Linear and Nonlinear Gaussian state space models, and Generalized Gaussian state space models. More information can be found at the website: <https://probml.github.io/dynamax/>.



# 21

## Discrete Hidden Markov Models

**Lab Objective:** *Understand how to use discrete Hidden Markov Models.*

A common probabilistic model is the *hidden Markov model* (HMM). In an HMM, we have two sequences of random variables,  $(X_t)_{t=0}^{\infty}$  and  $(Z_t)_{t=0}^{\infty}$ . The  $X_t$  are called the *state sequence* or *hidden state*, and the  $Z_t$  are known as the *observation sequence*. We assume that the  $X_t$  form a Markov chain, i.e. the distribution of  $X_t$  is entirely determined by the value of  $X_{t-1}$ , and that the distribution of  $Z_t$  is determined by the value of  $X_t$ . We also typically assume that we only know the values of the  $Z_t$ , not the  $X_t$  (hence the name). We denote the state space as  $\mathcal{X}$  and the observation space as  $\mathcal{Z}$ , so that for all  $t$  we have  $X_t \in \mathcal{X}$  and  $Z_t \in \mathcal{Z}$ . Hidden Markov models are useful in many situations where we have indirect observations of a sequential or time-based process, including speech and handwriting prediction, text analysis, gene prediction, and many other areas.

In this lab, we explore HMMs with discrete state and observation spaces. Assume the state space  $\mathcal{X}$  and observation space  $\mathcal{Z}$  are finite sets where  $|\mathcal{X}| = n$  and  $|\mathcal{Z}| = m$ . For simplicity we relabel these sets as  $\mathcal{X} = \{0, 1, 2, \dots, n - 1\}$  and  $\mathcal{Z} = \{0, 1, 2, \dots, m - 1\}$ . We will also assume that the HMM is temporally homogeneous, i.e. that the transition probabilities do not change with  $t$ .

In this case, we can parameterize all such HMMs by  $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$  where  $\boldsymbol{\pi} \in \mathbb{R}^n$  represents the distribution of  $X_0$  (the initial state distribution),  $A$  is a  $n \times n$  column-stochastic matrix describing how  $X_t$  is affected by  $X_{t-1}$  (the state transition matrix), and  $B$  is a  $m \times n$  column-stochastic matrix describing how  $Z_t$  is affected by  $X_t$  (the state observation matrix). The entries of  $\boldsymbol{\pi}$ ,  $A$ , and  $B$  specifically are the following:

$$\begin{aligned}\boldsymbol{\pi}_i &= P(X_0 = i) \\ a_{ij} &= P(X_t = i | X_{t-1} = j) \\ b_{ij} &= P(Z_t = i | X_t = j)\end{aligned}$$

Finally, we let  $\mathbf{z} = [z_0, z_1, \dots, z_{T-1}]$  be a vector of observations, where each  $z_t$  is a draw from  $Z_t$ .

Given one or both of  $\boldsymbol{\theta}$  and  $\mathbf{z}$ , there are several questions we might want to answer:

1. What is the likelihood that our model generated the observation sequence? In other words, what is  $P(\mathbf{z} | \boldsymbol{\theta})$ ?
2. Given  $\mathbf{z}$ ,  $\boldsymbol{\theta}$ , and an integer  $0 \leq k \leq T - 1$ , what is the most likely value for the state  $X_k$  at time  $k$ ?
3. Given  $\mathbf{z}$  and  $\boldsymbol{\theta}$ , what is the most likely state sequence  $\mathbf{x}$  to have generated  $\mathbf{z}$ ?

4. How can we choose the parameters  $\theta$  that maximize  $P(z|\theta)$ ?

The first of these is answered by the *forward pass* algorithm; the second by the *backwards pass* algorithm; the third by the *Viterbi algorithm*; and the fourth is typically approached using the *Baum-Welch algorithm*, which is the special case of expectation maximization applied to an HMM. Throughout this lab, we will use all four of these algorithms.

**Problem 1.** Create a class called `HMM`. Create the constructor, which accepts arguments `pi`, `A`, and `B`. Save each of these as an attribute with the same name.

## The Forward Pass

The goal of the forward pass algorithm is to efficiently compute  $P(z|\theta)$ . Directly expanding this out as a sum over all state sequences requires a number of computations that grows exponentially with the length of the observation sequence, and is completely impractical. Instead, the forward pass algorithm splits this probability into separate values that can be easily computed recursively.

As the first step of the algorithm, consider the values

$$\alpha_t(i) = P(z_0, \dots, z_t, x_t = i | \theta).$$

The law of total probability gives us that

$$P(z|\theta) = \sum_{i \in \mathcal{X}} \alpha_{T-1}(i),$$

so finding the  $\alpha_t(i)$  lets us find  $P(z|\theta)$ . It can be shown that<sup>1</sup>

$$\begin{aligned} \alpha_0(i) &= \pi_i b_{z_0, i} \\ \alpha_t(i) &= b_{z_t, i} \sum_{j \in \mathcal{X}} \alpha_{t-1}(j) a_{ij} \end{aligned}$$

which allows us to efficiently compute the  $\alpha_t(i)$  iteratively. When we implement this algorithm, we will store the values of  $\alpha_t(i)$  in a single 2D array. The  $(t, i)$ -th entry of this array will be the value  $\alpha_t(i)$ .

**Problem 2.** Create a method `forward_pass` in your `HMM` class to implement the forward pass algorithm. This function should accept the observation sequence `z` (with shape `(T,)`) and return the array of  $\alpha_t(i)$  values (with shape `(T, n)`).

To test your code, use the following example HMM:

```
>>> pi = np.array([.6, .4])
>>> A = np.array([[.7, .4], [.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> z_example = np.array([0, 1, 0, 2])
>>> example_hmm = HMM(pi, A, B)
```

You should get the following output using the example HMM:

---

<sup>1</sup>For verification of the mathematics behind this and the other algorithms in this lab, refer to the Volume 3 textbook.

```
>>> alpha = example_hmm.forward_pass(z_example)
>>> print(np.sum(alpha[-1, :]) # the probability of the observation
0.009629599999
```

**Problem 3.** Consider the following (very simplified) model of the price of a stock over time as an HMM. The observation states will be the change in the value of the stock. For simplicity, we will group these into five values: large decrease, small decrease, no change, small increase, large increase, labeled as integers from 0 to 4. The hidden state will be the overall trends of the market. We'll consider the market to have three possible states: declining in value (bear market), not changing in value (stagnant), and increasing in value (bull market), labeled as integers from 0 to 2. Let the HMM modeling this scenario have parameters

$$\pi = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \quad A = \begin{bmatrix} 0.5 & 0.3 & 0 \\ 0.5 & 0.3 & 0.3 \\ 0 & 0.4 & 0.7 \end{bmatrix}, \quad B = \begin{bmatrix} 0.3 & 0.1 & 0 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.3 \\ 0.1 & 0.2 & 0.4 \\ 0 & 0.1 & 0.2 \end{bmatrix}$$

The file `stocks.npy` contains a sequence of 50 observations drawn from this HMM. What is the probability of this observation sequence given these model parameters? Use your implementation of the forward pass algorithm from Problem 2 to find the answer. Note that the answer is very small, because there are lots of possible observation sequences.

## The Backward Pass

The backward pass algorithm seeks to answer the second question: given an observation sequence, parameters for an HMM, and a specific timestep, what is the most likely state at that step? As with the first question, trying to directly compute the answer via expanding into a sum over individual terms is completely impractical. The backwards pass algorithm takes a different approach.

Define the function

$$\gamma_t(i) = P(X_t = i | \mathbf{z}, \boldsymbol{\theta}).$$

The answer to the second question is then given by  $\text{argmax}_{i \in \mathcal{X}} \gamma_t(i)$  for a fixed timestep  $t$ . In order to compute the  $\gamma_t(i)$  efficiently, the backwards pass breaks the problem up in a clever way to allow an efficient iterative solution. Consider the values

$$\beta_t(j) = P(z_{t+1}, z_{t+2}, \dots, z_{T-1} | X_t = j, \boldsymbol{\theta}).$$

where  $\beta_{T-1}(i) = 1$ . It can be shown that

$$\beta_t(j) = \sum_{i \in \mathcal{X}} a_{ij} \beta_{t+1}(i) b_{z_{t+1}, i}$$

and that

$$\gamma_t(i) = \frac{\alpha_t(i) \beta_t(i)}{P(\mathbf{z} | \boldsymbol{\theta})},$$

allowing efficient computation of these values. The backwards pass algorithm simply consists of iteratively computing the  $\beta_t(i)$  values and using those to compute  $\gamma_t(i)$ . Note that the backwards pass algorithm requires running the forward pass algorithm first, and iterates over  $t$  in the opposite order.

**Problem 4.** Create a method `backward_pass` in your HMM class to implement the backward pass algorithm. This function should accept the observation sequence  $\mathbf{z}$  (with shape  $(T,)$ ) and return two arrays of the  $\beta_t(i)$  and  $\gamma_t(i)$  values (each with shape  $(T, n)$ ).

To test your function, your code should produce the following output on the example HMM:

```
>>> beta, gamma = example_hmm.backward_pass(z_example, alpha)
>>> print(beta)
[[0.0302  0.02792]
 [0.0812  0.1244 ]
 [0.38     0.26    ]
 [1.        1.      ]]
>>> print(gamma)
[[0.18816981 0.81183019]
 [0.51943175 0.48056825]
 [0.22887763 0.77112237]
 [0.8039794  0.1960206 ]]
```

With your function and the stock model from Problem 3, answer the following question: given the observation sequence in `stocks.npy`, what is the most likely initial hidden state  $X_0$ ?

## Most Likely Sequence with the Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm that seeks to find the most likely sequence of hidden states  $\mathbf{x} = (x_0, \dots, x_{T-1})$  that satisfies

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}} P(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}).$$

The algorithm proceeds by considering the values

$$\eta_t(i) = \max_{x_0, \dots, x_{t-1}} P(x_0, \dots, x_{t-1}, X_t = i, z_0, \dots, z_t | \boldsymbol{\theta})$$

The Bellman optimality principal can be used to show that

$$\begin{aligned}\eta_0(i) &= b_{z_0, i} \pi_i, \\ \eta_t(i) &= \max_{j \in \mathcal{X}} b_{z_t, i} a_{ij} \eta_{t-1}(j),\end{aligned}$$

allowing us to compute these efficiently. The  $\eta_t(i)$  give the maximizing probabilities at each timestep assuming we are in a certain hidden state. To extract the most likely sequence from the  $\eta_t(i)$ s, we iterate backwards as follows:

$$\begin{aligned}x_{T-1}^* &= \underset{j \in \mathcal{X}}{\operatorname{argmax}} \eta_{T-1}(j) \\ x_{t-1}^* &= \underset{j \in \mathcal{X}}{\operatorname{argmax}} b_{z_t, x_t^*} a_{x_t^*, j} \eta_{t-1}(j) = \underset{j \in \mathcal{X}}{\operatorname{argmax}} a_{x_t^*, j} \eta_{t-1}(j).\end{aligned}$$

**Problem 5.** Creating a method `viterbi_algorithm` in your HMM class to implement the Viterbi algorithm. This function should accept the observation sequence `z` (with shape  $(T,)$ ) and return the most likely state sequence `x*` (as an array with shape  $(T,)$ ).

To test your function, it should output the following on the example HMM:

```
>>> xstar = example_hmm.viterbi_algorithm(z_example)
>>> print(xstar)
[1 1 1 0]
```

Apply your function to the stock market HMM from Problem 3. With the observation sequence from `stocks.npy`, what is the most likely sequence of hidden states? Is the initial state of the most likely sequence the same as the most likely initial state you found in Problem 4?

## Text Analysis with HMMs

We now turn to an interesting application of the Baum-Welch algorithm to train HMMs. We have coded most of the pieces needed for the Baum-Welch algorithm already in this lab. However, many of these require the computations to be done in a more careful way than we have presented here in order to prevent underflow. Instead of delving into the details of that, we will use the HMM implementation provided by the `hmmlearn` package, specifically the `hmmlearn.hmm.CategoricalHMM` class.

This class uses slightly different conventions and syntax from the HMM class that we have coded in this lab, so we will illustrate how to use this class on the data from `stocks.npy`.

```
import numpy as np
from hmmlearn import hmm
z = np.load("stocks.npy")
```

In the initializer, we specify the number of hidden states as the `n_components` argument (we will use 3):

```
h = hmm.CategoricalHMM(n_components=3)
```

To train the HMM, call the `fit` method. This method accepts the observation sequence `z`. However, it expects it to be an array with shape  $(T, 1)$ , so reshape it as follows when you pass it in:

```
h.fit(z.reshape(-1, 1))
```

Now the HMM is trained. To see the trained HMM parameters, use the following attributes:  $\pi$  is stored as `h.startprob_`,  $A$  is stored as `h.transmat_`, and  $B$  is stored as `h.emissionprob_`. However,  $A$  and  $B$  are transposed from the convention we are using, so extracting the parameters looks like the following:

```
pi = h.startprob_
A = h.transmat_.T
B = h.emissionprob_.T
```

The particular application we will use this for is to take some text and treat it as the observation sequence  $\mathbf{z}$  of an HMM. Using the Baum-Welch algorithm to train an HMM with this setup can reveal interesting information about the underlying text. We will specifically use the sequence of characters (after stripping out punctuation and converting everything to lower-case) as our observation sequence.

In order to convert the raw text into data we can use with the `hmmlearn` package, we need to read and process the text and then map the characters to integer values. The following code accomplishes this task:

```

import numpy as np
import string
import codecs

def vec_translate(a, my_dict):
    # translate numpy array from symbols to state numbers or vice versa
    return np.vectorize(my_dict.__getitem__)(a)

def prep_data(filename):
    """
    Reads in the file and prepares it for use in an HMM.
    Returns:
        symbols (dict): a dictionary that maps characters to their integer ↵
                        values
        obs_sequence (ndarray): an array of integers representing the read-in ↵
                                text
    """
    # Get the data as a single string
    with codecs.open(filename, encoding="utf-8") as f:
        data=f.read().lower() # and convert to all lower case
    # remove punctuation and newlines
    remove_punct_map = {ord(char):
                            None for char in string.punctuation+"\n\r"}
    data = data.translate(remove_punct_map)
    # make a list of the symbols in the data
    symbols = sorted(list(set(data)))
    # convert the data to a NumPy array of symbols
    a = np.array(list(data))
    # make a conversion dictionary from symbols to state numbers
    symbols_to_obsstates = {x: i for i, x in enumerate(symbols)}
    # convert the symbols in a to state numbers
    obs_sequence = vec_translate(a,symbols_to_obsstates)
    return symbols, obs_sequence.reshape(-1, 1)

```

**Problem 6.** The file `declaration.txt` contains the text of the Declaration of Independence.

Train an `hmmlearn.hmm.CategoricalHMM` on this data with  $N = 2$  states and  $M = \text{len}(\text{set}(\text{obs})) = 27$  observation values (26 lower case characters and 1 whitespace character). Train the HMM with `n_iter=200` and `tol=1e-4` (note that both of these are arguments to the constructor, not to the `fit` function).

Once the learning algorithm converges, analyze the state observation matrix  $B$ . Note which rows correspond to the largest and smallest probability values in each column of  $B$ , and check the corresponding characters. The code below displays typical results for a well-converged HMM. (Note that the `u` before the `"` indicates that the string should be unicode, which will be useful in the next problem.)

```
>>> B = h.emissionprob_.T
>>> for i in range(len(B)):
...     print(u"{}", {:0.4f}, {:0.4f}"
...           .format(symbols[i], *B[i,:]))
    , 0.0051, 0.3324
a, 0.0000, 0.1247
c, 0.0460, 0.0000
b, 0.0237, 0.0000
e, 0.0000, 0.2245
d, 0.0630, 0.0000
g, 0.0325, 0.0000
f, 0.0450, 0.0000
i, 0.0000, 0.1174
# ...
```

What do you notice about the columns of  $B$ ? (Hint: Look at the vowels). Write your observations in a markdown cell. If there is nothing apparent in your output, try re-running your HMM. Note that the order of the columns is completely arbitrary, and your code may switch the role of the two columns.

**Problem 7.** The file `WarAndPeace.txt` contains a portion of the Russian text of *War and Peace* by Tolstoy. Train an HMM on the text in this file with  $N = 2$  states as in Problem 6. Interpret/explain your results. Which Cyrillic characters appear to be vowels?



# 22

# Speech Recognition using CDHMMs

**Lab Objective:** *Understand how speech recognition via CDHMMs works, and implement a simplified speech recognition system.*

## Continuous Density Hidden Markov Models

Some of the most powerful applications of hidden Markov models, speech and voice recognition, result from allowing the observation space to be continuous instead of discrete. These are called *continuous density hidden Markov models* (CDHMMs). The two most common formulations are *Gaussian HMMs* and *Gaussian mixture model HMMs* (GMMHMMs). In this lab, we will focus on GMMHMMs.

A GMMHMM is an HMM where the observation sequence variables  $(Z_t)_{t=0}^{\infty}$  are distributed according to a Gaussian mixture model. To review, a Gaussian mixture model is a continuous multivariate distribution composed of  $K$  Gaussian distributions  $\mathcal{N}(\mu_k, \Sigma_k)$  (where  $\mu_k \in \mathbb{R}^M$  and  $\Sigma_k \in \mathbb{R}^{M \times M}$ ) with corresponding weights  $c_k$ , for  $1 \leq k \leq K$ . A useful way to think of an individual GMM  $Z$  is to think of it as a pair of random variables  $Y$  and  $Z$ . The variable  $Y$  is a categorical variable on  $\{1, 2, \dots, K\}$  with probabilities given by the  $c_k$  (i.e.  $P(Y = k) = c_k$ ) and determines which Gaussian  $Z$  is drawn from. We then draw  $Z \sim \mathcal{N}(\mu_Y, \Sigma_Y)$ . The density function of a GMM is given as

$$f(\mathbf{z}) = \sum_{k=1}^K c_k \mathcal{N}(\mathbf{z}; \mu_k, \Sigma_k).$$

For an example of what this looks like, refer to Figure 22.1. Note that this is a weighted sum of the density functions of Gaussian variables; remember that this is different from a sum of independent Gaussian random variables, which is just another Gaussian random variable!

GMMHMMs are then formulated similar to the discrete HMMs we have encountered before. We will assume that the hidden states  $X_t$  have  $N$  possible values. Then, the hidden state of the GMMHMM is parameterized by an initial state vector  $\pi \in \mathbb{R}^N$  and a state transition matrix  $A \in \mathbb{R}^{N \times N}$  (both of these are the same as in the discrete case). The observation state is parameterized by one GMM per hidden state. We denote these as follows: for each hidden state  $1 \leq i \leq n$ , the observation state is distributed according to the GMM with component weights  $\{c_{i,1}, \dots, c_{i,K}\}$ , component means  $\{\mu_{i,1}, \dots, \mu_{i,K}\}$ , and component covariance matrices  $\{\Sigma_{i,1}, \dots, \Sigma_{i,K}\}$ .

To sample from a GMMHMM, follow the following process for each time step:

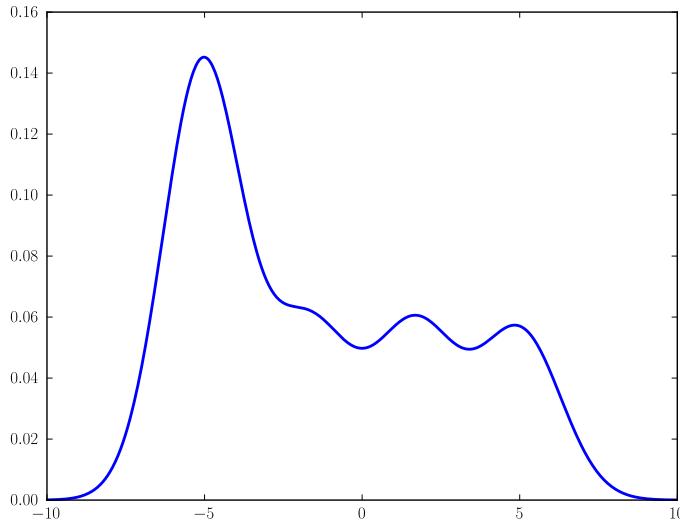


Figure 22.1: The probability density function of a mixture of Gaussians with four components.

- Determine the hidden state  $X_t$  (for  $X_0$  this is with  $\pi$ , and afterwards is determined by  $A$  and  $X_{t-1}$ ).
- Determine the GMM component  $Y_t$  by drawing from a categorical variable with probabilities  $c_{X_t,1}, \dots, c_{X_t,K}$  (so  $P(Y_t = j) = c_{X_t,j}$ ).
- Sample  $Z_t$  from the GMM by drawing from the normal distribution  $\mathcal{N}(\mu_{X_t,Y_t}, \Sigma_{X_t,Y_t})$ .

For an example of a sequence drawn from a GMMHMM, refer to Figure 22.2, which shows an observation sequence generated from a GMMHMM with two mixture component and two hidden states.

**Problem 1.** Consider the following GMMHMM with  $N = 3$  states, components of dimension  $M = 4$ , and  $K = 5$  components:

```
# NxN transition matrix
A = np.array([[.3, .3, .4], [.2, .3, .5], [.3, .2, .5]])
# NxK collection of component weights
weights = np.array([[.3, .2, .1, .2, .2], [.1, .3, .3, .2, .1],
                    [.1, .3, .2, .1, .3]])
# NxKxM collection of component means
means = np.array([np.floor(np.random.uniform(-100, 100, size = (5, 4)))
                  for i in range(3)])
# NxKx(MxM) collection of component covariance matrices
covars = np.array([[np.floor(np.random.uniform(1, 20))*np.eye(4)
                     for i in range(5)] for j in range(3)])
# (N,) ndarray initial state distribution
pi = np.array([.15, .15, .7])
```

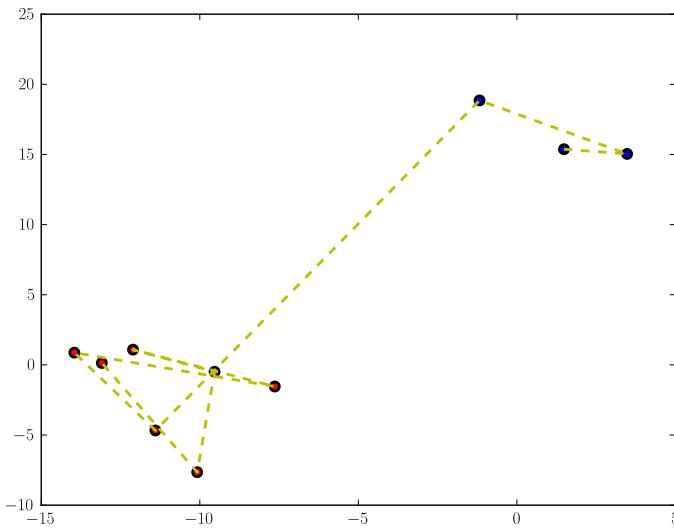


Figure 22.2: An observation sequence generated from a GMMHMM with two mixture components and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

The weight  $c_{i,k}$  is `weights[i,k]`, the mean  $\mu_{i,k}$  is `means[i,k,:]`, and the covariance matrix  $\Sigma_{i,k}$  is `covars[i,k,:,:]`.

Write a function `sample_gmmhmm` which accepts an integer  $T$ , and draws  $T$  samples from the above GMMHMM.

Use your function to draw  $T = 900$  samples. Use `sklearn.decomposition.PCA` with 2 components to plot the observations in two-dimensional space. Color the observations by state. How many distinct clusters do you see?

Hint: the function `np.random.choice` will be useful for drawing the hidden states and the GMM components, and `np.random.multivariate_normal` for the observation sequence. When plotting the samples, using the keyword argument `c` in `plt.scatter` allows you to specify the colors of the individual points.

## Speech Recognition and Hidden Markov Models

The questions we asked about discrete HMMs can also be asked about CDHMMs, and the algorithms that answer these questions are virtually identical to the discrete case. However, with continuous observations it is much more difficult to implement the algorithms in a numerically stable way. We will not have you implement any of the algorithms for CDHMMs yourself; instead, we will use the `hmmlearn` package.

Hidden Markov Models are the basis of modern speech recognition systems. The essential idea is that we will use the sound data as the observation sequence. However, there are a lot of details that we won't address in this lab; a fair amount of signal processing must precede the HMM stage, and more sophisticated approaches require the use of language models.

To transform the audio data into data for our HMMs, we will use the following process:

- Divide the audio into frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these intervals.
- Transform the audio signal into mel-frequency cepstral coefficients (MFCCs). This transformation is similar to a Fourier transform in that it breaks the sound signal into frequencies.
- Keep the first  $M = 10$  of these coefficients for each time frame. These will be our observation sequence in  $\mathbb{R}^M$ .

**Problem 2.** In the remainder of this lab, we will create a speech recognition system for the vocabulary of the following five words/phrases: “biology”, “mathematics”, “political science”, “psychology”, and “statistics”.

The `Samples` folder contains 30 recordings for each of the words/phrases in the vocabulary. These audio samples are 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. For each of the words, create a list holding the MFCC coefficients of the recordings of that word.

The function `scipy.io.wavfile.read` can be used to load the sound files, and the function `extract` in `MFCC.py` implements the MFCC coefficient algorithm:

```
from scipy.io import wavfile
import MFCC

samplerate, sound_data = wavfile.read(filename)
coefficients = MFCC.extract(sound_data)
```

Hint: it might be helpful to use a dictionary to store the lists of coefficients.

Industrial-grade speech recognition systems train a GMMHMM for each individual English *phoneme*, or distinct sound. This allows for a lot of versatility, but also comes with its own set of challenges: it requires a lot of training data, with the audio files split into individual phonemes, as well as requiring the true words to be written in terms of their phonemes. The setup also becomes much more complicated if we try to account for multiple dialects of English.

Instead, in this lab, we will train a GMMHMM for each of the five individual words in the vocabulary.<sup>1</sup> Recall, however, that the training procedure can get stuck in a poor local minimum. To combat this, we will train 10 GMMHMMs for each word (using a different random initialization of the parameters each time) and keep the model with the highest log-likelihood.

For training our HMM, we will use the `hmmlearn.hmm.GMMHMM` class. We will illustrate how to use this class. Let `data` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Then the model can be trained as follows:

```
import numpy as np
from hmmlearn import hmm

# hmmlearn expects the data to be in a single array:
data_collected = np.vstack(data)
```

<sup>1</sup>For a larger vocabulary, this requires a ludicrous number of GMMHMMs, which is why speech recognition systems don't typically do this.

```
# To separate the sequences, it requires the length of each:  
lengths = [item.shape[0] for item in data]  
  
# Initialize and train the model  
model = hmm.GMMHMM(n_components=5, covariance_type="diag")  
model.fit(data_collected, lengths=lengths)  
  
# Check the log-likelihood  
log_likelihood = model.monitor_.history[-1]
```

**Problem 3.** For each word, randomly split the list of MFCCs into a training set of 20 samples and a test set of the remaining 10 samples.

Use the training sets to train GMMHMMs on each word in the vocabulary. For each word in the vocabulary, train 10 GMMHMMs on the training set, using `n_components=5`. Keep the model with the highest log-likelihood for each word.

We have now trained our speech recognition system. With this model, if we are given a speech sample, how do we determine which word it is?

The method we use is as follows. First, we convert the sound into MFCC coefficients. Let  $\mathbf{z}$  denote these coefficients. Then, for each of the words in our vocabulary, we find  $\log P(\mathbf{z} \mid \text{word})$ , the log probability density of this observation sequence assuming that it came from that word's GMMHMM. This can be done with the `score` method of the `GMMHMM`. Then, whichever word's GMMHMM gives the highest log probability density will be our speech recognition model's prediction.

**Problem 4.** Write a `predict` function for your speech recognition model. In this function:

- Accept the MFCC coefficients of the speech sample to be predicted.
- Find the log probability density of the coefficients for each word's GMMHMM.
- Return the word with the highest probability as the speech recognition model's prediction.

**Problem 5.** For each of the five test sets, call your `predict` function on each sample, and find the proportion of each test set that your model predicts correctly. Display your results. How well does your model perform on this dataset?



# 23 Kalman Filter

**Lab Objective:** *Understand how to implement the standard Kalman Filter. Apply to the problem of projectile tracking.*

Measured observations are often prone to significant noise, due to restrictions on measurement accuracy. For example, most commercial GPS devices can provide a good estimate of geolocation, but only within a dozen meters or so. A Kalman filter is an algorithm that takes a sequence of noisy observations made over time and attempts to get rid of the noise, producing more accurate estimates than the original observations. To do this, the algorithm needs information about the system being observed.

Consider the problem of tracking a projectile as it travels through the air. Short-range projectiles approximately trace out parabolas, but a sensor that is recording measurements of the projectile's position over time will likely show a path that is much less smooth. Because we know something about the laws of physics, we can filter out the noise in the measurements using basic Newtonian mechanics, recovering a more accurate estimate of the projectile's trajectory. In this lab, we will simulate measurements of a projectile and implement a Kalman filter to estimate the complete trajectory of the projectile.

## Linear Dynamical Systems

The standard Kalman filter assumes that: (1) we have a linear dynamical system, (2) the state of the system evolves over time with some noise, and (3) we receive noisy measurements about the state of the system at each iteration. More formally, letting  $\mathbf{x}_k$  denote the state of the system at time  $k$ , we have

$$\mathbf{x}_0 = \boldsymbol{\mu}_0 + \mathbf{w}_0 \tag{23.1}$$

$$\mathbf{x}_k = F_{k-1}\mathbf{x}_{k-1} + G_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_k \quad k = 1, \dots, T \tag{23.2}$$

where  $F_k$  is a state-transition model,  $G_k$  is a control-input model,  $\mathbf{u}_k$  is a control vector, and  $\mathbf{w}_k$  is the noise present in state  $k$ . This noise is assumed to be drawn from a multivariate Gaussian distribution with zero mean and covariance matrix  $Q_k$ . We let  $\boldsymbol{\mu}_0$  be the expected value of the initial condition  $\mathbf{x}_0$ , i.e.  $\mathbf{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, Q_0)$ . The control-input model and control vector allow the assumption that the state can be additionally influenced by some other factor than the linear state-transition model.

We further assume that the states are “hidden,” and we only get the noisy observations

$$\mathbf{z}_k = H_k \mathbf{x}_k + \mathbf{v}_k \quad k = 0, \dots, T \quad (23.3)$$

where  $H_k$  is the observation model mapping the state space to the observation space, and  $\mathbf{v}_k$  is the observation noise present at iteration  $k$ . As with the aforementioned error, we assume that this noise is drawn from a multivariate Gaussian distribution with zero mean and covariance matrix  $R_k$ .

The dynamics stated above are all taken to be linear. Thus, for our purposes, the operators  $F_k$ ,  $G_k$ , and  $H_k$  are all matrices, and  $\mathbf{x}_k$ ,  $\mathbf{u}_k$ ,  $\mathbf{z}_k$ , and  $\mathbf{v}_k$  are all vectors.

We will assume that the transition and observation models, the control vector, and the noise covariances are constant, i.e. for each  $k$ , we will replace  $F_k, Q_k, H_k, R_k$ , and  $\mathbf{u}_k$  and with  $F, Q, H, R$ , and  $\mathbf{u}$ . We will also assume that  $G = I$  is the identity matrix, so it can safely be ignored.

**Problem 1.** Begin implementing a `KalmanFilter` class by writing an initialization method that stores the transition and observation models, noise covariances, and control vector. We provide an interface below:

```
class KalmanFilter(object):
    def __init__(self, F, Q, H, R, u):
        """
        Initialize the dynamical system models.

        Parameters
        -----
        F : ndarray of shape (n, n)
            The state transition model.
        Q : ndarray of shape (n, n)
            The covariance matrix for the state noise.
        H : ndarray of shape (m, n)
            The observation model.
        R : ndarray of shape (m, m)
            The covariance matrix for observation noise.
        u : ndarray of shape (n,)
            The control vector.
        """
        pass
```

We now derive the linear dynamical system parameters for a projectile traveling through  $\mathbb{R}^2$  undergoing a constant downward gravitational force of  $9.8 \text{ m/s}^2$ . The relevant information needed to describe how the projectile moves through space is its position and velocity. Thus, our state vector has the form

$$\mathbf{x}_k = \begin{pmatrix} s_k^x \\ s_k^y \\ V_k^x \\ V_k^y \end{pmatrix},$$

where  $s_k^x$  and  $s_k^y$  give the  $x$  and  $y$  coordinates of the position (in meters) at the  $k$ th time interval, and  $V_k^x$  and  $V_k^y$  give the horizontal and vertical components of the velocity (in meters per second) at the  $k$ th time interval, respectively.

How does the system evolve from one time step to the next? Assuming each time step is 0.1 seconds, it is easy enough to calculate the new position:

$$\begin{aligned}s_{k+1}^x &= s_k^x + (0.1)V_k^x \\ s_{k+1}^y &= s_k^y + (0.1)V_k^y.\end{aligned}$$

Further, since the only force acting on the projectile is gravity (we are ignoring things like wind resistance), the horizontal velocity remains constant:

$$V_{k+1}^x = V_k^x.$$

The vertical velocity, however, does change due to the effects of gravity. From basic Newtonian mechanics, we have

$$V_{k+1}^y = V_k^y - (0.1)(9.8).$$

In summary, over one time step, the state evolves from  $\mathbf{x}_k$  to  $\mathbf{x}_{k+1}$ , where

$$\mathbf{x}_{k+1} = \begin{pmatrix} s_k^x + (0.1)V_k^x \\ s_k^y + (0.1)V_k^y \\ V_k^x \\ V_k^y - 0.98 \end{pmatrix}.$$

From this equation, you can extract the state transition model  $F$  and the control vector  $u$ .

We now turn our attention to the observation model. Imagine that a radar sensor captures (noisy) measurements of the projectile's position as it travels through the air. At each time step, the radar transmits the observation  $z = (z_x, z_y)$  given by

$$\begin{aligned}z_x &= s_x + v_x \\ z_y &= s_y + v_y,\end{aligned}$$

where  $(v_x, v_y)$  is a noise vector assumed to be drawn from a multivariate Gaussian with mean zero and some known covariance. These equations indicate the appropriate choice of observation model.

**Problem 2.** Work out the transition and observation models  $F$  and  $H$ , along with the control vector  $\mathbf{u}$ , corresponding to the projectile. Assume that the noise covariances are given by

$$\begin{aligned}Q &= 0.1 \cdot I_4 \\ R &= 5000 \cdot I_2.\end{aligned}$$

Instantiate a `KalmanFilter` object with these values.

## Generating Testable Data

In real-world problems, we would only have access to the observation data, but for this lab, we would like to see for certain how well our Kalman filter is predicting the true states. To do this, we'll need a toy dataset consisting of a sequence of true states and corresponding observations, which we'll simulate from the dynamical system derived above. In addition to the system parameters, we need an initial state  $\mathbf{x}_0$  to get started. Computing the subsequent states and observations is simply a matter of following Equations 23.2 and 23.3. Bear in mind that the initial state  $\mathbf{x}_0$  does not have added noise, but the initial observation and all subsequent states and observations take noise into account.

**Problem 3.** Write a function that generates a state and observation sequence by evolving the dynamical system from a given initial state (the function `numpy.random.multivariate_normal` will be useful). To do this, implement the following:

```
def evolve(x0, N, F, Q, H, R, u):
    """
    Generate the first N states and observations from the dynamical system

    Parameters
    -----
    x0 : ndarray of shape (n,)
        The initial state.
    N : integer
        The number of time steps to evolve.
    F : ndarray of shape (n, n)
        The state transition model.
    Q : ndarray of shape (n, n)
        The covariance matrix for the state noise.
    H : ndarray of shape (m, n)
        The observation model.
    R : ndarray of shape (m, m)
        The covariance matrix for observation noise.
    u : ndarray of shape (n,)
        The control vector.

    Returns
    -----
    states : ndarray of shape (n, N)
        States 0 through N-1, given by each column.
    obs : ndarray of shape (m, N)
        Observations 0 through N-1, given by each column.
    """
    pass
```

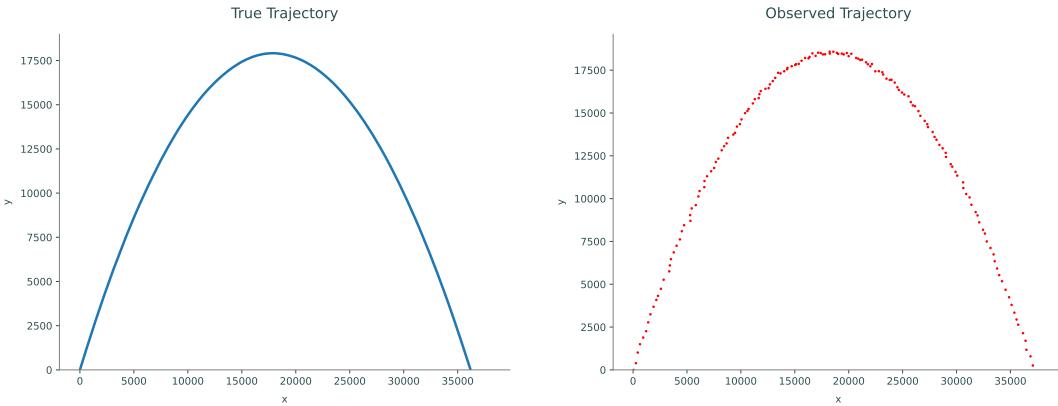


Figure 23.1: State sequence (left) and sampling of observation sequence (right).

To test our Kalman filter, simulate the true and observed trajectories of a projectile with initial state

$$\mathbf{x}_0 = \begin{pmatrix} 0 \\ 0 \\ 300 \\ 600 \end{pmatrix}.$$

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the  $y$  coordinate to return to 0). Plot both the true trajectory (as a blue line) and the observed trajectory (as red dots). Your results should qualitatively match those given in Figure 23.1.

## State Estimation with the Kalman Filter

The Kalman filter is a recursive estimator that smooths out the noise in real time, estimating each current state based on the past state estimate and the current measurement. This process is done by repeatedly invoking two steps: Predict and Update. The predict step is used to estimate the current state based on the previous state. The update step then combines this prediction with the current observation, yielding a more robust estimate of the current state.

To describe these steps in detail, we need additional notation. Let

- $\hat{\mathbf{x}}_{n|m}$  be the state estimate at time  $n$  given only measurements up through time  $m$ ; and
- $P_{n|m}$  be an error covariance matrix, measuring the estimated accuracy of the state at time  $n$  given only measurements up through time  $m$ .

The elements  $\hat{\mathbf{x}}_{k|k}$  and  $P_{k|k}$  represent the state of the filter at time  $k$ , giving the state estimate and the accuracy of the estimate, respectively. We evolve the filter recursively, as follows:

$$(Initialize) \quad \hat{\mathbf{x}}_{0|-1} = \boldsymbol{\mu}_0 \quad (23.4a)$$

$$P_{0|-1} = Q_0 \quad (23.4b)$$

$$\text{Predict} \quad \hat{\mathbf{x}}_{k|k-1} = F\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{u} \quad k = 1, \dots, T \quad (23.4c)$$

$$P_{k|k-1} = FP_{k-1|k-1}F^\top + Q \quad k = 1, \dots, T \quad (23.4d)$$

$$\text{Update} \quad K_k = P_{k|k-1}H^\top (HP_{k|k-1}H^\top + R)^{-1} \quad k = 0, \dots, T \quad (23.4e)$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k (\mathbf{z}_k - H\hat{\mathbf{x}}_{k|k-1}) \quad k = 0, \dots, T \quad (23.4f)$$

$$P_{k|k} = (I - K_k H)P_{k|k-1} \quad k = 0, \dots, T \quad (23.4g)$$

The more observations we have, the greater the accuracy of these estimates becomes (i.e the norm of the error covariance matrix converges to 0).

**Problem 4.** Use Equations (24.21) to add a method to your `KalmanFilter` class that estimates a state sequence corresponding to a given observation sequence and initial state estimate. Implement the following class method:

```
def estimate(self, x, P, z):
    """
    Compute the state estimates using the Kalman filter.
    If x and P correspond to time step k, then z is a sequence of
    observations starting at time step k+1.

    Parameters
    -----
    x : ndarray of shape (n,)
        The initial state estimate.
    P : ndarray of shape (n, n)
        The initial error covariance matrix.
    z : ndarray of shape (m, N)
        Sequence of N observations (each column is an observation).

    Returns
    -----
    out : ndarray of shape (n, N)
        Sequence of state estimates (each column is an estimate).
    """
    pass
```

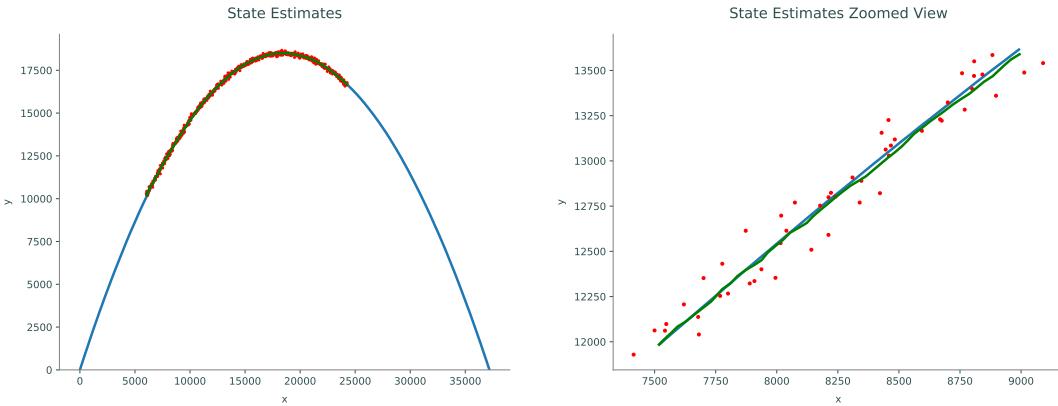


Figure 23.2: State estimates (green) together with observations (red) and true state sequence (blue).

Returning to the projectile example, we now assume that our radar sensor has taken observations from time steps 200 through 800 (take the corresponding slice of the observations produced in Problem 3). Using these observations, we seek to estimate the corresponding true states of the projectile. We must first come up with a state estimate  $\hat{\mathbf{x}}_{200}$  for time step 200, and then feed this into the Kalman filter to obtain estimates  $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$ .

**Problem 5.** Calculate an initial state estimate  $\hat{\mathbf{x}}_{200}$  as follows: For the horizontal and vertical positions, simply use the observed position at time 200. For the velocity, compute the average velocity between the observations  $\mathbf{z}_k$  and  $\mathbf{z}_{k+1}$  for  $k = 200, \dots, 208$ , then average these 9 values and take this as the initial velocity estimate. (Hint: `numpy.diff` is useful here.)

Using the initial state estimate,  $P_{200} = 10^6 \cdot Q$  and your Kalman filter, compute the next 600 state estimates, i.e. compute  $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$ . Plot these state estimates as a smooth green curve together with the radar observations (as red dots) and the entire true state sequence (as a blue curve). Zoom in to see how well it follows the true path. Your plots should be similar to Figure 23.2.

In the absence of observations, we can still estimate some information about the state of the system at some future time. We can do this by recognizing that the expected state noise  $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$  at any time  $k$ . Thus, given a current state estimate  $\hat{\mathbf{x}}_{n|m}$  using only measurements up through time  $m$ , the expected state at time  $n + 1$  is

$$\hat{\mathbf{x}}_{n+1|m} = F\hat{\mathbf{x}}_{n|m} + \mathbf{u}$$

**Problem 6.** Add a method to your class that predicts the next  $k$  states given a current state estimate but in the absence of observations. Do so by implementing the following method:

```
def predict(self, x, k):
    """
    Predict the next k states in the absence of observations.
    """
```

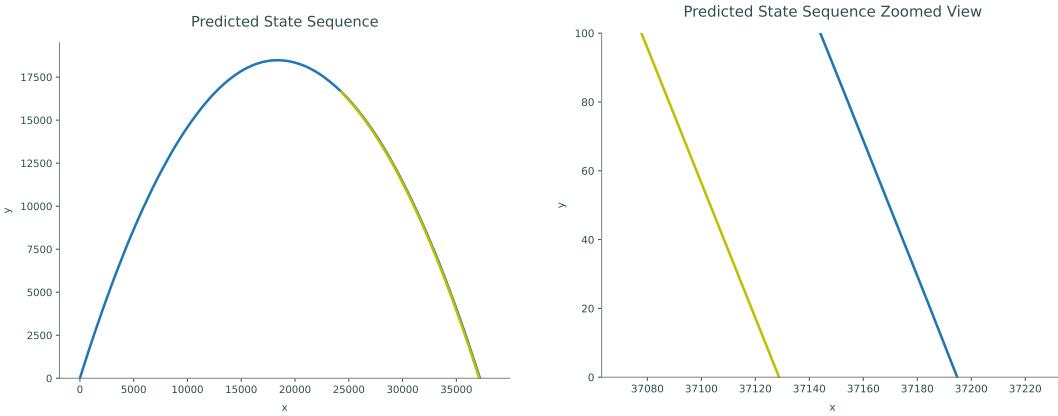


Figure 23.3: Predicted (yellow) vs. actual (blue) point of impact.

```

Parameters
-----
x : ndarray of shape (n,)
    The current state estimate.
k : integer
    The number of states to predict.

Returns
-----
out : ndarray of shape (n, k)
    The next k predicted states.
"""
pass

```

We can use this prediction routine to estimate where the projectile will hit the surface.

**Problem 7.** Using the final state estimate  $\hat{x}_{800}$  that you obtained in Problem 5, predict the future states of the projectile until it hits the ground. Predicting approximately the next 450 states should be sufficient.

Plot the actual state sequence together with the predicted state sequence (as a yellow curve), and observe how near the prediction is to the actual point of impact. Your results should be similar to those shown in Figure 23.3. The following code will help you plot both the actual and predicted state sequences together in a nice plot.

```

>>> predicted = kal.predict(estimate[:, -1], 450)

>>> # create a dynamic xlim to always plot both sequences together
>>> x1 = actual_states[0, :][np.where(actual_states[1, :] >= 0)][-1]
>>> x2 = predicted[0, :][np.where(predicted[1, :] >= 0)][-1]

```

```
>>> plt.xlim(min(x1, x2)-50, max(x1, x2)+50)
```

In the absence of observations, we can also reverse the system and iterate backward in time to infer information about states of the system prior to measured observations. The system is reversed by

$$\mathbf{x}_k = F^{-1}(\mathbf{x}_{k+1} - \mathbf{u} - \boldsymbol{\varepsilon}_{k+1}).$$

Considering again that  $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$  at any time  $k$ , we can ignore this term, simplifying the recursive estimation backward in time.

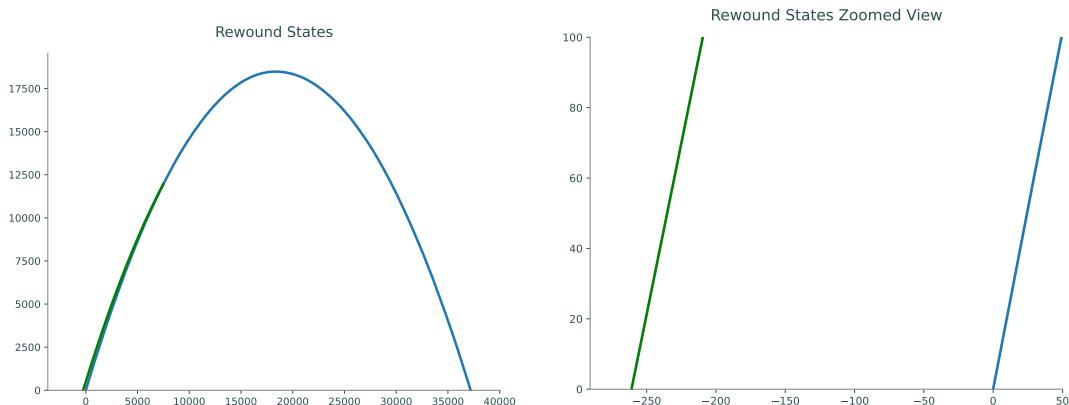


Figure 23.4: Predicted (green) vs. actual (blue) point of origin.

**Problem 8.** Add a method to your class that rewinds the system from a given state estimate, returning predictions for the previous states. Do so by implementing the following method:

```
def rewind(self, x, k):
    """
    Predict the k states preceding the current state estimate x.

    Parameters
    -----
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of preceding states to predict.

    Returns
    -----
    out : ndarray of shape (n, k)
        The k preceding predicted states.
    """
    pass
```

Returning to the projectile example, we can now predict the point of origin.

**Problem 9.** Using your state estimate  $\hat{\mathbf{x}}_{250}$ , predict the point of origin of the projectile along with all states leading up to time step 250. Note that you may have to take a few extra time steps to predict the point of origin. (The point of origin is the first point along the trajectory where the  $y$  coordinate is 0.) Plot these predicted states (in green) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 23.4.

Repeat the prediction starting with  $\hat{\mathbf{x}}_{600}$ . Compare to the previous results. Which is better? Why?

# 24

## ARMA Models

**Lab Objective:** ARMA( $p, q$ ) models combine autoregressive and moving-average models in order to forecast future observations using time-series. In this lab, we will build an ARMA( $p, q$ ) model to analyze and predict future weather data and then compare this model to statsmodels built-in ARMA package as well as the VARMAX package. Then we will forecast macroeconomic data as well as the future height of the Rio Negro.

### Time Series

A time series is any discrete-time stochastic process. In other words, it is a sequence of random variables,  $\{Z_t\}_{t=1}^T$ , that are determined by their time  $t$ . We let the realization of the time series  $\{Z_t\}_{t=1}^T$  be denoted by  $\{z_t\}_{t=1}^T$ . Examples of time series include heart rate readings over time, pollution readings over time, stock prices at the closing of each day, and air temperature. Often when analyzing time series, we want to forecast future data, such as what will the stock price of a company will be in a week and what will the temperature be in 10 days.

### ARMA( $p, q$ ) Models

One way to forecast a time series is using an ARMA model. The *Wold Theorem* says that any covariance-stationary time series can be well approximated with an ARMA model. An ARMA( $p, q$ ) model combines an autoregressive model of order  $p$  and a moving average model of order  $q$  on a time series  $\{Z_t\}_{t=1}^T$ . The model itself is a discrete-time stochastic process  $(Z_t)_{t \in \mathbb{Z}}$  satisfying the equation

$$Z_t = \mathbf{c} + \underbrace{\left( \sum_{i=1}^p \Phi_i Z_{t-i} \right)}_{\text{AR}(p)} + \underbrace{\left( \sum_{j=1}^q \Theta_j \varepsilon_{t-j} \right)}_{\text{MA}(q)} + \varepsilon_t \quad (24.1)$$

where each  $\varepsilon_t$  is an identically-distributed Gaussian variable with mean 0 and constant covariance  $\Sigma$ ,  $\mathbf{c} \in \mathbb{R}^n$ , and  $\Phi_i$  and  $\Theta_j$  are in  $M_n(\mathbb{R})$ .

## AR( $p$ ) Models

An AR( $p$ ) model works similar to a weighted random walk. Recall that in a random walk, the current position depends on the immediate past position. In the autoregressive model, the current data point in the time series depends on the past  $p$  data points. However, the importance of each of the past  $p$  data points is not uniform. With an error term to represent white noise and a constant term to adjust the model along the y-axis, we can model the stochastic process with the following equation:

$$Z_t = \mathbf{c} + \sum_{i=1}^p \Phi_i Z_{t-i} + \varepsilon_t \quad (24.2)$$

If there is a high correlation between the current and previous values of the time series, then the AR( $p$ ) model is a good representation of the data, and thus the ARMA( $p, q$ ) model will most likely be a good representation. The coefficients  $\{\Phi_i\}_{i=1}^p$  are larger when the correlation is stronger.

In this lab, we will be using weather data from Provo, Utah<sup>1</sup>. To check that the data can be represented well, we need to look at the correlation between the current and previous values.

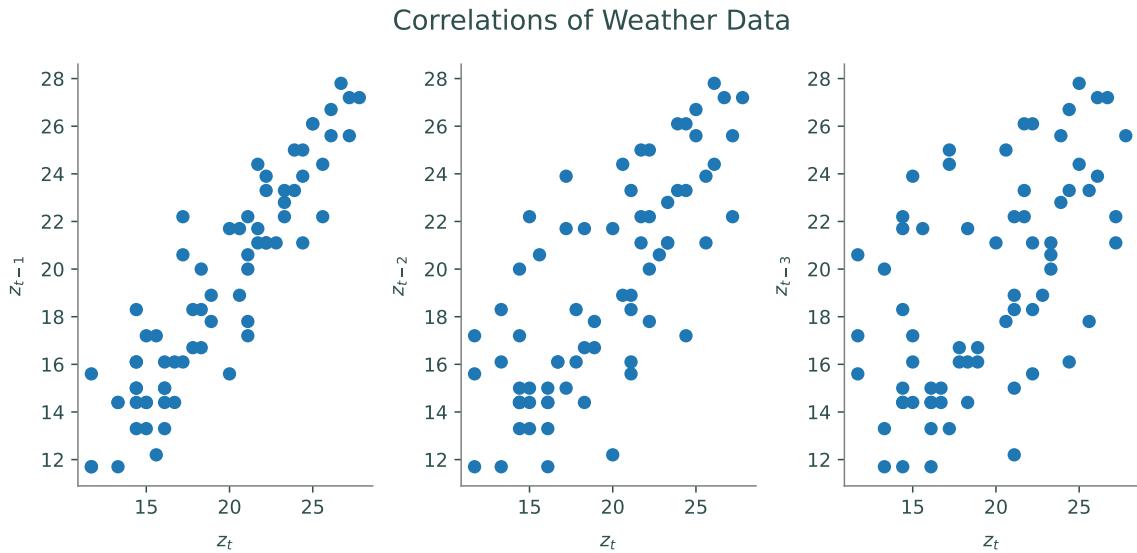


Figure 24.1: These graphs show that the weather data is correlated to its previous values. The correlation is weaker in each graph successively, showing that the further in the past the data is, the less correlated the data becomes.

## MA( $q$ ) Models

A moving average model of order  $q$  is used to factor in the varying error of the time series. This model uses the error of the current data point and the previous data points to predict the next datapoint. Similar to an AR( $p$ ) model, this model uses a linear combination (which includes a constant term to adjust along the y-axis..

<sup>1</sup>This data was taken from <https://forecast.weather.gov/data/obhistory/metric/KPVU.html>

$$Z_t = \mathbf{c} + \varepsilon_t + \sum_{i=1}^q \Theta_i \varepsilon_{t-i} \quad (24.3)$$

This part of the model simulates shock effects in the time series. Examples of shock effects include volatility in the stock market or sudden cold fronts in the temperature.

Combining both the AR( $p$ ) and MA( $q$ ) models, we get an ARMA( $p, q$ ) model which forecasts based on previous observations and error trends in the data.

## ARIMA( $p, d, q$ ) Models

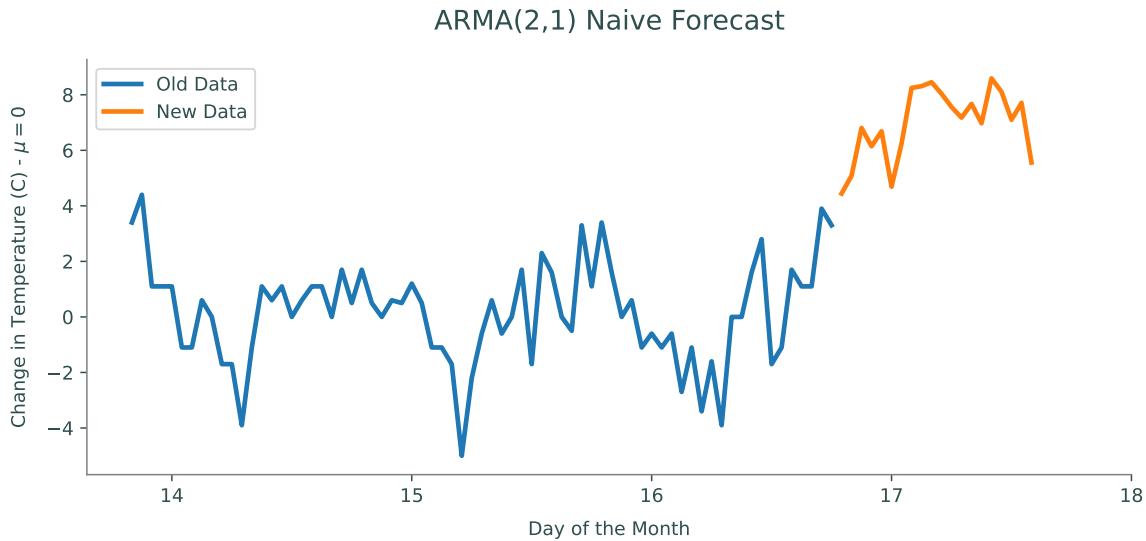
Not all ARMA models are covariance stationary. However, many time series can be made covariance stationary by differencing. Let  $\delta Z_t$  represent the time series  $z_t = Z_t - Z_{t-1}$  obtained by taking a difference of the terms. If the trend is linear a first difference is usually stationary. If the trend is quadratic a second difference may be necessary  $\delta^2 Z_t = \delta(\delta Z_t)$ . An ARIMA( $p, d, q$ ) model is a discrete-time stochastic process  $(Z_t)_{t \in \mathbb{Z}}$  satisfying the equation

$$\delta^d Z_t = \mathbf{c} + \underbrace{\left( \sum_{i=1}^p \Phi_i z_{t-i} \right)}_{\text{AR}(p)} + \underbrace{\left( \sum_{j=1}^q \Theta_j \varepsilon_{t-j} \right)}_{\text{MA}(q)} + \varepsilon_t \quad (24.4)$$

## Finding Parameters

One of the most difficult parts of using an ARMA( $p, q$ ) model is identifying the proper parameters of the model. For simplicity, at the beginning of this lab we discuss univariate ARMA models with parameters  $\{\phi_i\}_{i=1}^p$ ,  $\{\theta_i\}_{i=1}^q$ ,  $\mu$ , and  $\sigma$ , where  $\mu$  and  $\sigma$  are the mean and standard deviation of the error. Note that  $\{\phi_i\}_{i=1}^p$  and  $\{\theta_i\}_{i=1}^q$  determine the order of the ARMA model. For this lab, we will let  $\mathbf{c} = 0$ .

A naive way to use an ARMA model is to choose  $p$  and  $q$  based on intuition. Figure 24.1 showed that there is a strong correlation between  $z_t$  and  $z_{t-1}$  and between  $z_t$  and  $z_{t-2}$ . The correlation is weaker between  $z_t$  and  $z_{t-3}$ . Intuition then suggests to choose  $p = 2$ . By looking at the correlations between the current noise with previous noise, similar to Figure 24.1, it can also be seen that there is a weak correlation between  $z_t$  and  $\varepsilon_t$  and between  $z_t$  and  $\varepsilon_{t-1}$ . Between  $z_t$  and  $\varepsilon_{t-2}$  there is no correlation. For more on how these error correlations were found, see Additional Materials. Intuition from these correlations suggests to choose  $q = 1$ . Thus, a naive choice for our model is an ARMA(2, 1) model.

Figure 24.2: Naive forecast on `weather.npy`

**Problem 1.** Write a function `arma_forecast_naive()` that builds an ARMA( $p, q$ ) model. Your function should accept as parameters  $p$ ,  $q$ , and  $n$ , where  $p$  is the order of the autoregressive model,  $q$  is the order of the moving average model, and  $n$  is the number of observations to predict. Assume  $c = 0$ , and let  $\phi_i = .5$ ,  $\theta_i = .1$ , and  $\varepsilon_i \sim \mathcal{N}(0, 1)$  for all  $i$ .

The file `weather.npy` contains data on the temperature in Provo, Utah from 7:56 PM May 13, 2019 to 6:56 PM May 16, 2019, taken every hour. This time series is NOT covariance stationary, so to make it covariance stationary, take its first difference (Hint: you might find `np.diff()` helpful). We denote the new covariance stationary time series as  $\{z_t\}_{t=1}^T$ . Predict the next  $n$  observations for  $\{z_t\}$  by iterating through Equation 24.4.

Run your code on `weather.npy`, and plot the observed differences  $\{z_t\}_{t=1}^T$  followed by your predicted observations of  $z_t$ . For  $p=2$ ,  $q=1$ , and  $n=20$ , your plot should look similar to Figure 24.2, however, due to the variance of the error  $\varepsilon_t$ , the plot will not look exactly like Figure 24.2. The predictions may be higher or lower on the  $y$ -axis.

Let  $\Theta = \{\phi_i, \theta_j, \mu, \sigma\}$  be the set of parameters for an ARMA( $p, q$ ) model. Suppose we have a set of observations  $\{z_t\}_{t=1}^n$ . Our goal is to find the  $p, q$ , and  $\Theta$  that maximize the likelihood of the ARMA model given the data. Using the chain rule, we can factorize the likelihood of the model given this data as

$$p(\{z_t\} \mid \Theta) = \prod_{t=1}^n p(z_t \mid z_{t-1}, \dots, z_1, \Theta) \quad (24.5)$$

## State Space Representation

In a general ARMA( $p, q$ ) model, the likelihood is a function of the unobserved error terms  $\varepsilon_t$  and is not trivial to compute. Simple approximations can be made, but these may be inaccurate under certain circumstances. Explicit derivations of the likelihood are possible, but tedious. However, when the ARMA model is placed in state-space, the Kalman filter affords a straightforward, recursive way to compute the likelihood.

We demonstrate one possible state-space representation of an ARMA( $p, q$ ) model. Let  $r = \max(p, q + 1)$ . Define

$$\hat{\mathbf{x}}_{t|t-1} = [x_{t-1} \ x_{t-2} \ \cdots \ x_{t-r}]^\top \quad (24.6)$$

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \quad (24.7)$$

$$H = [1 \ \theta_1 \ \theta_2 \ \cdots \ \theta_{r-1}] \quad (24.8)$$

$$Q = \begin{bmatrix} \sigma & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad (24.9)$$

$$w_t \sim \text{MVN}(0, Q), \quad (24.10)$$

where  $\phi_i = 0$  for  $i > p$ , and  $\theta_j = 0$  for  $j > q$ . Note that Equation 24.2 gives

$$F\hat{\mathbf{x}}_{t-1|t-2} + w_t = \begin{bmatrix} \sum_{i=1}^r \phi_i x_{t-i} \\ x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-(r-1)} \end{bmatrix} + \begin{bmatrix} \varepsilon_t \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (24.11)$$

$$= [x_t \ x_{t-1} \ \cdots \ x_{t-(r-1)}]^\top \quad (24.12)$$

$$= \hat{\mathbf{x}}_{t|t-1} \quad (24.13)$$

We note that  $z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu$ .<sup>2</sup>

Then the linear stochastic dynamical system

$$\hat{\mathbf{x}}_{t+1|t} = F\hat{\mathbf{x}}_{t|t-1} + w_t \quad (24.14)$$

$$z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu \quad (24.15)$$

describes the same process as the original ARMA model.

**NOTE**

---

<sup>2</sup>For a proof of this fact, see Additional Materials.

Equation 24.15 involves a deterministic component, namely  $\mu$ . The Kalman filter theory developed in the previous lab, however, assumed  $\mathbb{E}[\varepsilon_t] = 0$  for the observations  $z_{t|t-1}$ . This means you should subtract off the mean  $\mu$  of the error from the time series observations  $z_{t|t-1}$  when using them in the predict and update steps.

### Likelihood via Kalman Filter

We assumed in Equation 24.10 that the error terms of the model are Gaussian. This means that each conditional distribution in 24.5 is also Gaussian, and is completely characterized by its mean and variance:

$$\text{mean } H\hat{\mathbf{x}}_{t|t-1} + \mu \quad (24.16)$$

$$\text{variance } HP_{t|t-1}H^\top \quad (24.17)$$

where  $\hat{\mathbf{x}}_{t|t-1}$  and  $P_{t|t-1}$  are easily found via the Kalman filter, during the Predict step. Given that each conditional distribution is Gaussian, the likelihood can then be found as

$$p(\{z_t\} | \Theta) = \prod_{t=1}^n \mathcal{N}(z_t | H\hat{\mathbf{x}}_{t|t-1} + \mu, HP_{t|t-1}H^\top). \quad (24.18)$$

**Problem 2.** Write a function `arma_likelihood()` that returns the log-likelihood of an ARMA model, given a time series  $\{z_t\}_{t=1}^T$ . This function should accept `filename` which contains the observations, and it should accept as parameters each parameter in  $\Theta$ . In this case, the time series should be the change in temperature of `weather.npy`, which is the first difference of the time series found in `weather.npy`, as was done in Problem 1. Adapt Equation 24.18 to calculate and return the log-likelihood of the ARMA( $p, q$ ) model as a `float`.

Use the provided `state_space_rep()` function to generate  $F, Q$ , and  $H$ . The function `kalman()` has also been provided to help calculate the means and covariances of each observation. Calling the function `kalman()` on a time series will return an array whose values are  $\hat{\mathbf{x}}_{t|t-1}$  and an array whose values are  $P_{t|t-1}$  for each  $t \leq n$ .

Hint: remember to subtract off the mean  $\mu$  from the inputted observation when using `kalman()`.

Also, when implementing Equation 24.18, you may find it best to use `scipy.stats.distributions.norm.pdf`, but keep in mind that this method accepts standard deviations, not variances. When implemented correctly, your function should match the following output:

```
>>> arma_likelihood(filename="weather.npy", phis=np.array([0.9]),
                     thetas=np.array([0]), mu=17., std=0.4)
-1375.1805469978776
```

## Model Identification

Now that we can compute the likelihood of a given ARMA model, we want to find the best choice of parameters given our time series. In this lab, we define the model with the "best" choice of parameters as the model which minimizes the AIC. The benefit of minimizing the AIC is that it rewards goodness of fit while penalizing overfitting. The AIC is expressed by

$$2k \left( 1 + \frac{k+1}{n-k} \right) - 2\ell(\Theta) \quad (24.19)$$

where  $n$  is the sample size,  $k = p + q + 2$  is the number of parameters in the model, and  $\ell(\Theta)$  is the maximum likelihood for the model class.

To compute the maximum likelihood for a model class, we need to optimize 24.18 over the space of parameters  $\Theta$ . We can do so by using an optimization routine such as `scipy.optimize.minimize` on the function `arma_likelihood()` from Problem 2. Use the following code to run this routine.

```
from scipy.optimize import minimize

# assume p, q, and time_series are defined
def f(x): # x contains the phis, thetas, mu, and std
    try:
        return -1*arma_likelihood(filename, phis=x[:i], thetas=x[i:i+j],
                                  mu=x[-2], std=x[-1])
    except np.linalg.LinAlgError:
        return np.inf

# create initial point
x0 = np.zeros(p + q + 2)
x0[-2] = time_series.mean()
x0[-1] = time_series.std()
sol = minimize(f, x0, method = "SLSQP")
sol = sol['x']
```

This routine will return a vector `sol` where the first  $p$  values are  $\{\phi_i\}_{i=1}^p$ , the next  $q$  values are  $\{\theta_i\}_{i=1}^q$ , and the last two values are  $\mu$  and  $\sigma$ , respectively. Note the wrapper `f(x)` returns the negative log-likelihood. This is because `scipy.optimize.minimize` finds the *minimizer* of  $f(x)$  and we are solving for the *maximum* likelihood.

To minimize the AIC, we perform *model identification*. This is choosing the order of our model,  $p$  and  $q$ , from some admissible set. The order of the model which minimizes the AIC is then the optimal model.

**Problem 3.** Write a function `model_identification()` that accepts `filename` containing the time series data and parameters `p_max` and `q_max` as integers. Determine which ARMA( $p, q$ ) model has the minimum AIC for all  $1 \leq p \leq p_{\text{max}}$  and  $1 \leq q \leq q_{\text{max}}$ . Then, return each parameter in  $\Theta$  of that model.

Hint: when calculating the AIC using Equation 24.19, bear in mind that  $-\ell(\Theta) = f(\text{sol})$  where `sol` is found in the code above and explained in the following paragraph.

Your code should replicate the following output up to at least 4 decimal places.

```
>>> model_identification(filename="weather.npy", p_max=4, q_max=4)
(array([ 0.7213538]), array([-0.26246426]), 0.359785001944352, ←
 1.5568374351425505)
```

## Forecasting with Kalman Filter

We have now identified the optimal ARMA( $p, q$ ) model. We can use this model to predict future states. The Kalman filter provides a straightforward way to predict future states by giving the mean and variance of the conditional distribution of future observations. Observations can be found as follows

$$z_{t+k} \mid z_1, \dots, z_t \sim \mathcal{N}(z_{t+k} \mid H\hat{x}_{t+k|t} + \mu, HP_{t+k|t}H^\top) \quad (24.20)$$

To evolve the Kalman filter, recall the predict and update rules of a Kalman filter.

$$\begin{aligned} (\text{Initialize}) \quad \hat{\mathbf{x}}_{0|-1} &= \boldsymbol{\mu}_0 \\ P_{0|-1} &= Q_0 \end{aligned}$$

$$\begin{aligned} \text{Predict} \quad \hat{\mathbf{x}}_{k|k-1} &= F\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{u} & k = 1, \dots, T \\ P_{k|k-1} &= FP_{k-1|k-1}F^\top + Q & k = 1, \dots, T \end{aligned}$$

$$\begin{aligned} \text{Update} \quad K_k &= P_{k|k-1}H^\top (HP_{k|k-1}H^\top + R)^{-1} & k = 0, \dots, T \\ \hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + K_k(\mathbf{z}_k - H\hat{\mathbf{x}}_{k|k-1}) & k = 0, \dots, T \\ P_{k|k} &= (I - K_kH)P_{k|k-1} & k = 0, \dots, T \end{aligned}$$

With ARMA, we define observational noise covariance  $R$  and drift term  $\mathbf{u}$  to both be 0.

### ACHTUNG!

Recall that the values returned by `kalman()` are conditional on the previous observation. To compute the mean and variance of future observations, the values  $x_{n|n}$  and  $P_{n|n}$  MUST be computed using the Update step. Once they are computed, only the Predict step is needed to find the future means and covariances.

**Problem 4.** Write a function `arma_forecast()` that accepts `filename` containing a time series, the parameters for an ARMA model, and the number `n` of observations to forecast. Calculate the mean and covariance of the future `n` observations using the Kalman filter.

To do this, use `state_space_rep()` to generate  $F$ ,  $Q$ , and  $H$ . Then, use `kalman()` (with  $\mu$  subtracted off from the covariance stationary time series  $\mathbf{z}_k$ ) to calculate  $\hat{\mathbf{x}}_{k|k-1}$  and  $P_{k|k-1}$ , respectively. Use the Update step on the last elements of  $\mathbf{z}_k$  (with  $\mu$  subtracted off),  $\hat{\mathbf{x}}_{k|k-1}$ , and  $P_{k|k-1}$  to find  $\hat{\mathbf{x}}_{k|k}$  and  $P_{k|k}$ . Then, iteratively use the Predict step to make future predictions of the mean and covariance. Recall that  $R$  and  $\mathbf{u}$  are both defined to be 0! Also, remember that once you find a mean  $\hat{\mathbf{x}}_{k|k-1}$  and covariance  $P_{k|k-1}$ , you must use Equations 24.16 and 24.17 to transform them back into observation space.

Plot the original observations as well as the mean of each future observation. Plot a 95% confidence interval (2 standard deviations away from the mean) around the means of future observations. Hint: the standard deviation is the square root of the covariance calculated.

The following code should create a plot similar to Figure 24.3.

```
# Get optimal model as found in the previous problem
phis, thetas, mu, std = np.array([0.72135856]), np.array([-0.26246788]), ←
    0.35980339870105321, 1.5568331253098422

# Forecast optimal mode
arma_forecast(filename="weather.npy", phis=phis, thetas=thetas,
               mu=mu, std=std, n=30)
```

How does this graph compare to the naive ARMA graph from Problem 1?

## Statsmodel ARMA

The module `statsmodels` contains a package that includes an ARMA model class. This is accessed through ARIMA model, which stands for Autoregressive Integrated Moving Average. This class also uses a Kalman Filter to calculate the MLE. When creating an ARIMA object, initialize the variables `endog` (the data) and `order` (the order of the model). The order is of the form  $(p, d, q)$  where  $d$  is the differences. To create an ARMA model, set  $d = 0$ . The object can then be fitted based on the MLE using a Kalman Filter.

```
from statsmodels.tsa.arima.model import ARIMA
# Intialize the object with weather data and order (1, 1)
>>> model = ARIMA(z, order=(p, 0, q), trend='c').fit(method="innovations_mle")

# Access p and q
>>> model.specification.k_ar
p
>>> model.specification.k_ma
q
```

As in the other problems, the time series passed in should be covariance stationary. The AIC of an ARMA model object is saved as the attribute `aic`. Since the AIC is much faster to compute using `statsmodels`, model identification is much faster. Once a model is chosen, the method `predict` will forecast  $n$  observations, where  $n$  is the number of known observations. It will return the mean of each future observation.

```
# Predict from the beginning of the model to 30 observations in the future
```

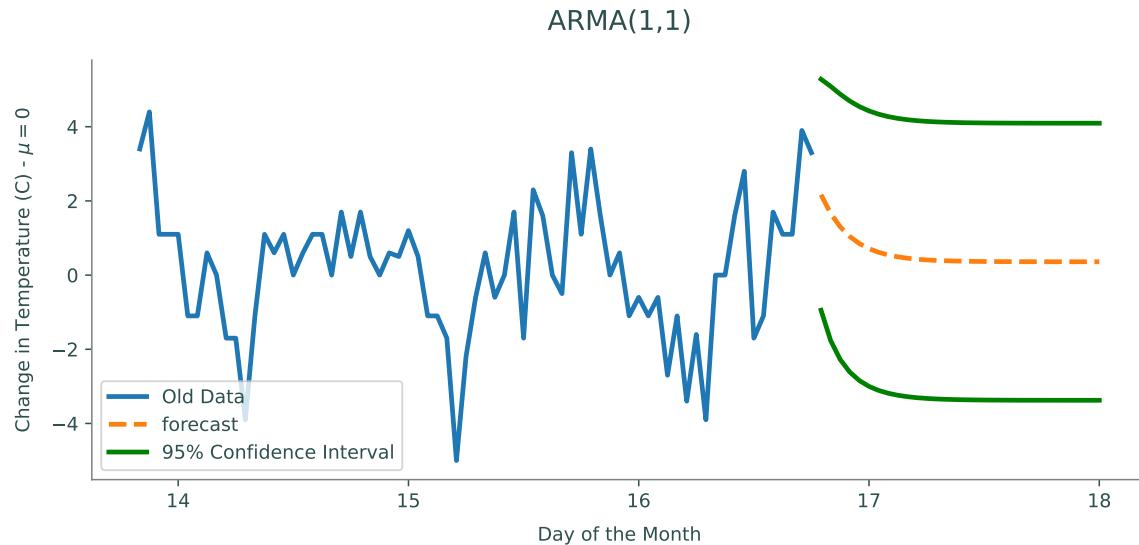


Figure 24.3: ARMA(1, 1) forecast on `weather.npy`

```
model.predict(start=0, end=len(data)+30)
```

**Problem 5.** Write a function `sm_arma()` that accepts `filename` containing a time series, integer values for `p_max` and `q_max`, and the number `n` of values to predict.

As in Problem 3, perform model identification to find the ARMA( $p, q$ ) model with the best AIC for  $1 \leq p \leq p_{\text{max}}$  and  $1 \leq q \leq q_{\text{max}}$ , but this time use `statsmodels`. Make sure the model is fit using the MLE.

Use the optimal model to predict `n` future observations of the time series. Plot the original observations along with the predicted observations from the beginning through `n` observations in the future, as given by `statsmodels`. **Return the AIC** of the optimal model.

For `p_max=3`, `q_max=3`, and `n=30`, your graph should look similar to Figure 24.4. How does this graph compare to Problem 1? Problem 4?

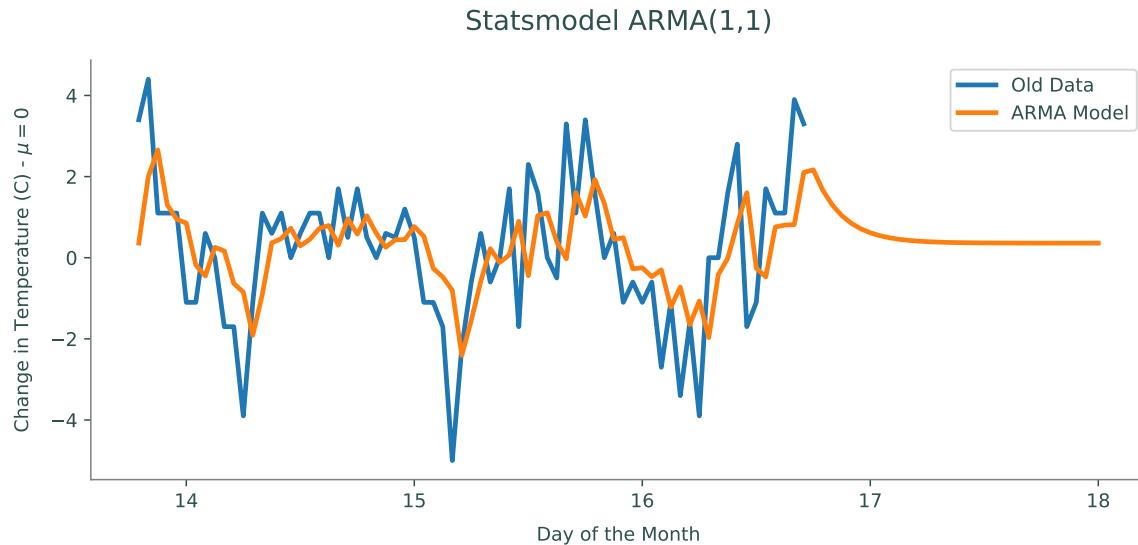


Figure 24.4: Statsmodel forecast on `weather.npy`.

## Statsmodel VARMA

Until now we have been dealing with univariate ARMA models. Multivariate ARMA models are used when we have multiple time series that can be useful in predicting one another. For example say we have two time series  $z_{t,1}$  and  $z_{t,2}$ . The multivariate ARMA(1,1) model is as follows:

$$z_{t,1} = c_1 + \phi_{11}z_{t-1,1} + \phi_{12}z_{t-1,2} + \theta_{11}\varepsilon_{t-1,1} + \theta_{12}\varepsilon_{t-1,2} \quad (24.22)$$

$$z_{t,2} = c_2 + \phi_{21}z_{t-1,1} + \phi_{22}z_{t-1,2} + \theta_{21}\varepsilon_{t-1,1} + \theta_{22}\varepsilon_{t-1,2} \quad (24.23)$$

This can be written in matrix form as shown in Equation 24.1. The module `statsmodels` contains a package that includes a VARMAX model class which can be used to create a multivariate ARMA model. VARMAX stands for Vector Autoregression Moving Average with Exogenous Regressors. An exogenous regressor is a time series that affects the model but is not affected by it. In the example below we have two time series corresponding to the price of copper and aluminum. Since aluminum is a substitute for copper, it is reasonable to assume the price of aluminum may help us predict the price of copper and vice versa.

```
from statsmodels.tsa.api import VARMAX
import statsmodels.api as sm

# Load in world copper data
data = sm.datasets.copper.load_pandas().data
# Create index compatible with VARMAX model
data.index = pd.period_range(start="1951", end="1975", freq='Y')

# Initialize and fit model
mod = VARMAX(data[["ALUMPRICE", "COPPERPRICE"]])
mod = mod.fit(maxiter=1000, disp=False)
```

```
# Predict the price of aluminium and copper until 1985
pred = mod.predict("1951", "1985")

# Get confidence intervals
forecast_obj = mod.get_forecast("1981")
all_CI = forecast_obj.conf_int(alpha=0.05)

# Plot predictions against true price
pred.plot(figsize=(10,4))
plt.plot(data["ALUMPRICE"], label="Actual ALUMPRICE")
plt.plot(data["COPPERPRICE"], label="Actual COPPERPRICE")
plt.legend()
plt.title("VARMA Predictions for World Copper Market Dataset")
```

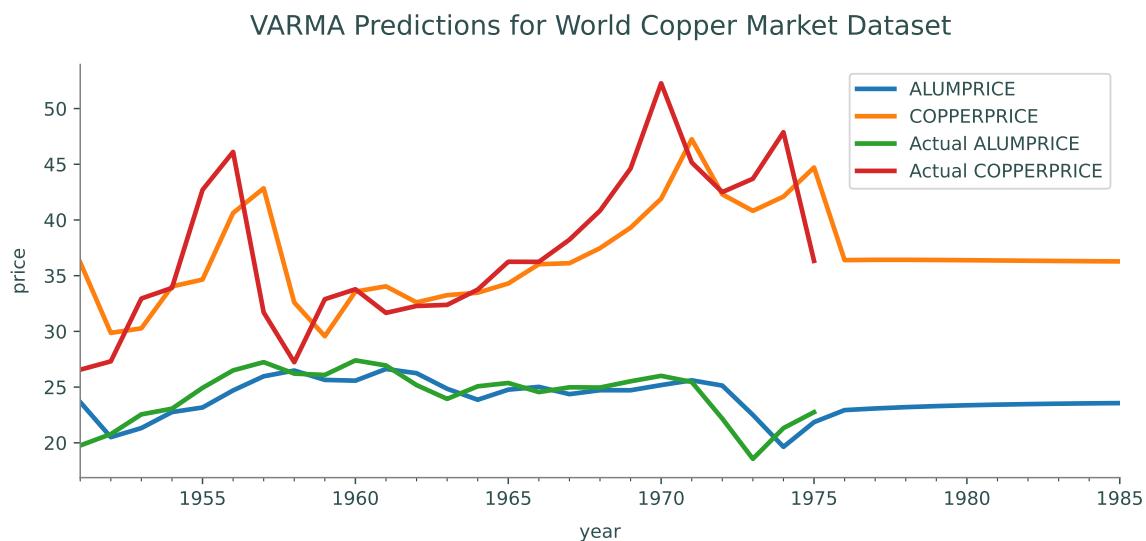


Figure 24.5: Statsmodel VAR(1) forecast.

**Problem 6.** Write a function `sm_varma()` that accepts start and end dates for forecasting. Use the statsmodels VARMAX class to forecast on macroeconomic data between the start and end dates. The following code shows how to obtain the data.

```
# Load in data
df = sm.datasets.macrodta.load_pandas().data
# Create DatetimeIndex
dates = df[["year", "quarter"]].astype(int).astype(str)
dates = dates["year"] + "Q" + dates["quarter"]
dates = dates_from_str(dates)
df.index = pd.DatetimeIndex(dates)
# Select columns used in prediction
```

```
df = df[["realgdp", "realcons", "realinv"]]
```

Initialize your VARMAX model with the `df` specified above, and include the parameter `freq="Q-DEC"`. Fit your model, and predict from the start date until the end date. Then, get the model forecast until the end date. Plot the original data, prediction, and a 95% confidence interval (2 standard deviations away from the mean) around the future observations. **Return the AIC** of the chosen model. The plot should be similar to Figure 24.6.

Hint: in the example above, `mod.predict("1951", "1985")` returns a dataframe of 2 columns that contain the predicted values of `"ALUMPRICE"` and `"COPPERPRICE"`, respectively, from the years 1951 to 1985. Also, `all_CI` is a dataframe where each column indicates the corresponding dataset and whether it is a lower or upper bound of a confidence interval determined by the alpha value. Thus, the column `"lower ALUMPRICE"` with `alpha=0.05` contains the lower bounds of a 95% confidence interval for the `"ALUMPRICE"` dataset.

The dataset `"realgdp"` contains the real gross domestic product, `"realcons"` contains real personal consumption expenditures, and `"realinv"` contains real gross private domestic investment. Since personal consumption and domestic investment are components of gross domestic product, it is reasonable to assume these time series will be useful in predicting one another.

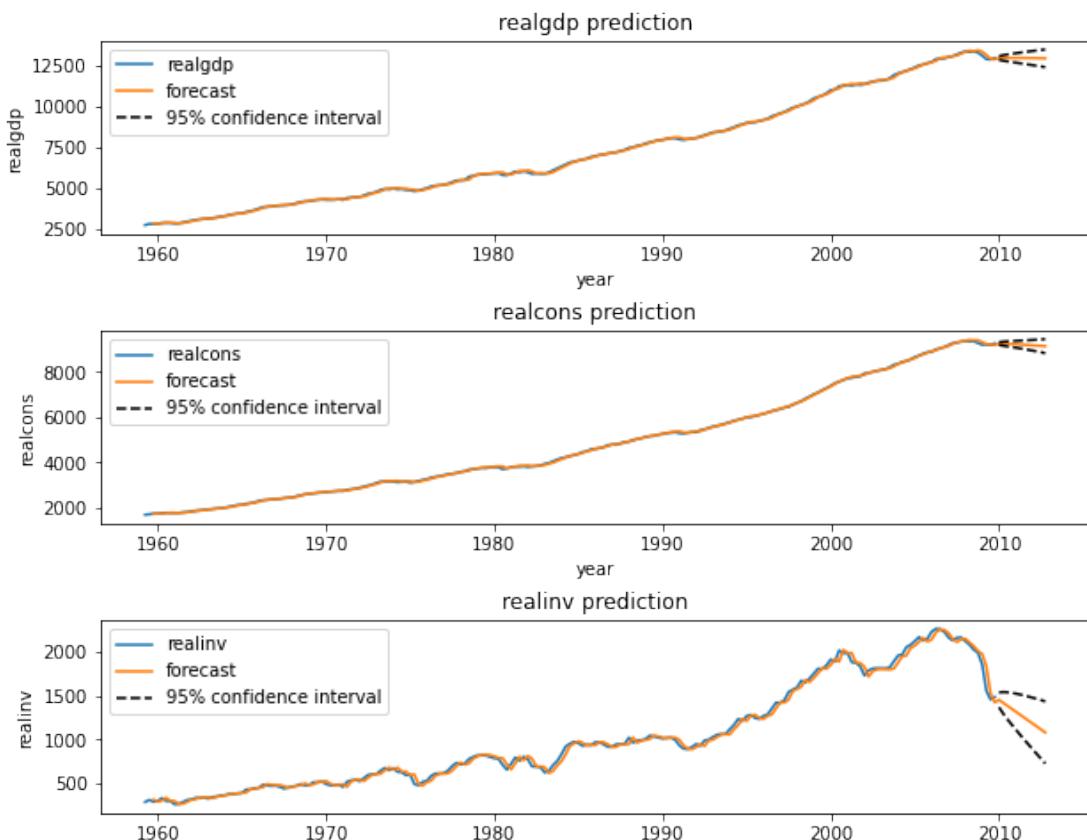


Figure 24.6: Macroeconomic data is forecasted 12 years in the future using statsmodels.

## Optional

The `statsmodels` package can help us perform model identification. The method `arma_order_select_ic` will find the optimal order of the ARMA model based on certain criteria. The first parameter `y` is the data. The data must be a NumPy array, not a Pandas DataFrame. The parameter `ic` defines the criteria trying to be minimized. The method will return a dictionary, where the minimal order of each criteria can be accessed.

```
>>> import statsmodels.api as sm
>>> from statsmodel.tsa.stattools import arma_order_select_ic as order_select
>>> import pandas as pd

>>> # Get sunspot data and give DateTimeIndex
>>> sunspot = sm.datasets.sunspots.load_pandas().data
>>> sunspot.index = pd.Index(pd.date_range("1700", end="2009", freq="A-DEC"))
>>> sunspot.drop(columns = ["YEAR"], inplace = True)

>>> # Find best order where p < 5 and q < 5
>>> # Use AICc as basis for minimization
>>> order = order_select(sunspot.values,max_ar=4,max_ma=4,ic=["aic", "bic"],←
    fit_kw={"method": "mle"})
>>> print(order["aic_min_order"])
(4,2)
>>> print(order["bic_min_order"])
(4,2)

>>> # Fit model
>>> # Note that we need to set the dimensionality to zero in order to have an ←
    ARMA model.
>>> model = ARIMA(sunspot,order = (4,0,2)).fit(method="innovations_mle")

>>> # Predict values from 1950 to 2012.
>>> prediction = model.predict(start="1950", end="2012")

>>> # Plot the prediction along with the sunspot data.
>>> fig, ax = plt.subplots(figsize=(13, 7))
>>> plt.plot(prediction)
>>> plt.plot(sunspot["1950": "2009"])
>>> ax.set_title("Sunspot Dataset")
>>> ax.set_xlabel("Year")
>>> ax.set_ylabel("Number of Sunspots")
>>> plt.show()
```

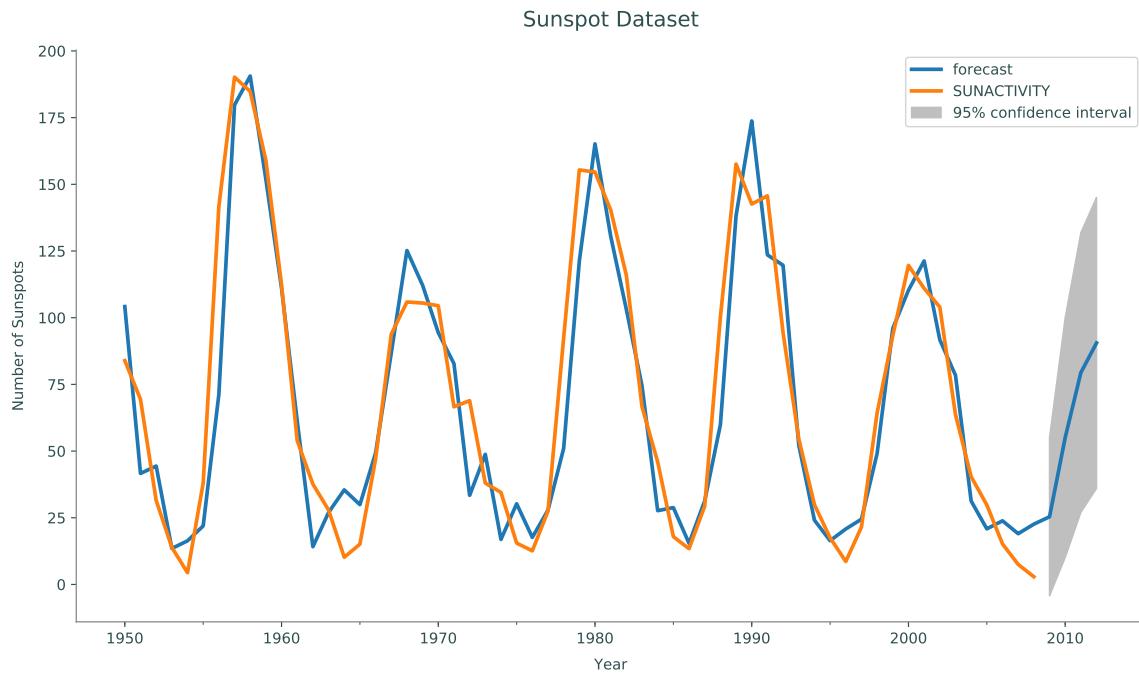


Figure 24.7: Sunspot activity data is forecasted four years in the future using `statsmodels`.

**Problem 7.** The dataset `manaus` contains data on the height of the Rio Negro from every month between January 1903 and January 1993. Write a function `manaus()` that accepts the forecasting range as strings `start` and `end`, the maximum parameter for the AR model `p` and the maximum parameter of the MA model `q`. The parameters `start` and `end` should be strings corresponding to a `DateTimeIndex` in the form `Y%M%D`, where `D` is the last day of the month.

The function should determine the optimal order for the ARMA model based on the AIC and the BIC. Then forecast and plot on the range given for both models and compare. Return the order of the AIC model and the order of the BIC model, respectively. For the range "`1983-01-31`" to "`1995-01-31`", your plot should look like Figure 24.8.

(Hint: The data passed into `arma_order_select_ic` must be a NumPy array. Use the attribute `values` of the Pandas DataFrame.)

To get the `manaus` dataset and set it with a `DateTimeIndex`, use the following code:

```
# Get dataset
raw = pydata("manaus")
# Convert to DateTimeIndex
manaus = pd.DataFrame(raw.values, index=pd.date_range("1903-01", "1993-01", ←
    freq='M'))
manaus = manaus.drop(0, axis=1)
# Set new column title
manaus.columns = ["Water Level"]
```

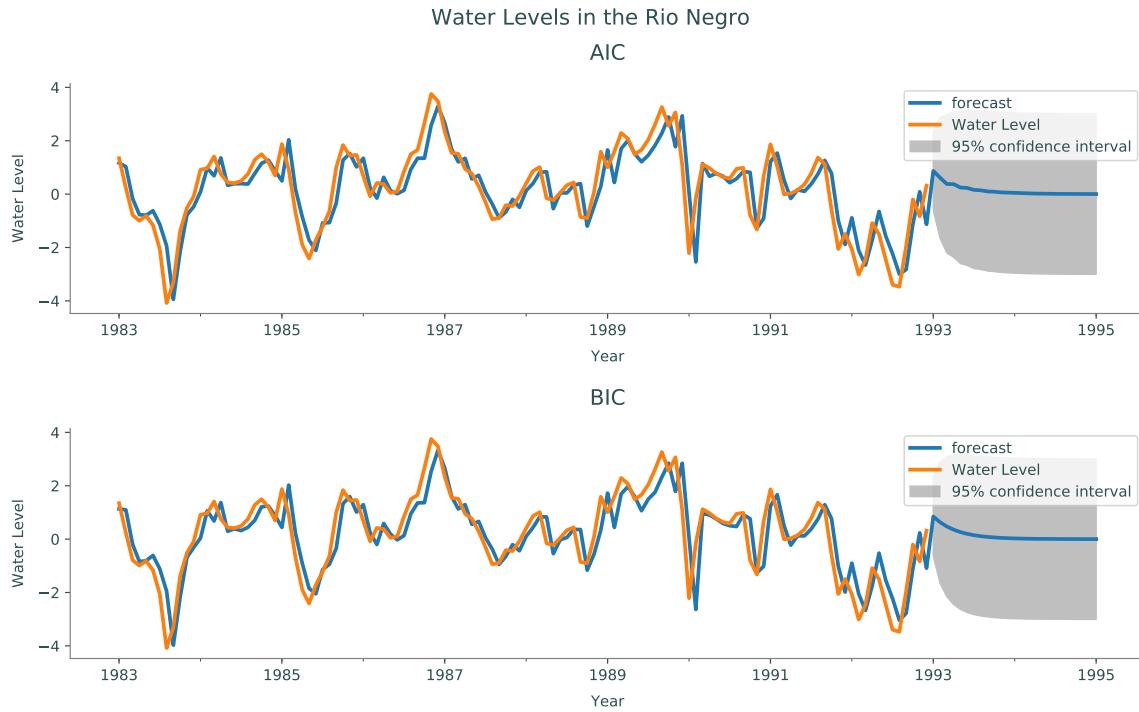


Figure 24.8: AIC and BIC based ARMA models of `manaus` dataset.

## Additional Materials

### Finding Error Correlation

To find the correlation of the current error with past error, the noise of the data needs to be isolated. Each data point  $y_t$  can be decomposed as

$$y_t = T_t + S_t + R_t, \quad (24.24)$$

where  $T_t$  is the overall trend of the data,  $S_t$  is a seasonal trend, and  $R_t$  is noise in the data. The overall trend is what the data tends to do as a whole, while the seasonal trend is what the data does repeatedly. For example, if looking at airfare prices over a decade, the overall trend of the data might be increasing due to inflation. However, we can break this data into individual years. We call each year a season. The seasonal trend of the data might not be strictly increasing, but have increases during busy seasons such as Christmas and summer vacations.

To find  $T_t$ , we use an  $M$ -fold method. In this case,  $M$  is the length of our season. We define the equation

$$T_t = \frac{1}{M} \sum_{-M/2 < i < M/2} y_{i+t}. \quad (24.25)$$

This means for each  $t$ , we take the average of the season surrounding  $y_t$ .

To find the seasonal trend, first subtract the overall trend from the time series. Define  $x_t = y_t - T_t$ . The value of the seasonal trend can then be found by averaging each day of the season over every season. For example, if the season was one year, we would find the average value on the first day of the year over all seasons, then the second, and so on. Thus,

$$S_t = \frac{1}{K} \sum_{i \equiv t \pmod{M}} x_i \quad (24.26)$$

where  $K$  is the number of seasons.

With the overall and seasonal trend known, the noise of the data is simply  $R_t = y_t - T_t - S_t$ . To determine the strength of correlations with the current error and the past error, plot  $y_t$  vs.  $R_{t-i}$  as in Figure 24.1.

### Proof of Equation 24.15

$$\sum_{i=1}^p \phi_i(z_{t-i} - \mu) + a_t + \sum_{j=1}^q \theta_j a_{t-j} = \sum_{i=1}^p \phi_i(H\hat{\mathbf{x}}_{t-i}) + a_t + \sum_{j=1}^q \theta_j a_{t-j} \quad (24.27)$$

$$= \sum_{i=1}^r \phi_i(x_{t-i} + \sum_{k=1}^{r-1} \theta_k x_{t-i-k}) + a_t + \sum_{j=1}^{r-1} \theta_j a_{t-j} \quad (24.28)$$

$$= a_t + \sum_{i=1}^r \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j \left( \sum_{i=1}^r \phi_i x_{t-j-i} + a_{t-j} \right) \quad (24.29)$$

$$= a_t + \sum_{i=1}^r \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j x_{t-k} \quad (24.30)$$

$$= x_t + \sum_{j=1}^{r-1} \theta_j x_{t-k} \theta_k x_{t-k} \quad (24.31)$$

$$= z_t. \quad (24.32)$$

# 25 Non-negative Matrix Factorization Recommender

**Lab Objective:** *Understand and implement the non-negative matrix factorization for recommendation systems.*

## Introduction

Collaborative filtering is the process of filtering data for patterns using collaboration techniques. More specifically, it refers to making prediction about a user's interests based on other users' interests. These predictions can be used to recommend items and are why collaborative filtering is one of the common methods of creating a recommendation system.

Recommendation systems look at the similarity between users to predict what item a user is most likely to enjoy. Common recommendation systems include Netflix's Movies you Might Enjoy list, Spotify's Discover Weekly playlist, and Amazon's Products You Might Like.

## Non-negative Matrix Factorization

Non-negative matrix factorization is one algorithm used in collaborative filtering. It can be applied to many other cases, including image processing, text mining, clustering, and community detection. The purpose of non-negative matrix factorization is to take a non-negative matrix  $V$  and factor it into the product of two non-negative matrices.

For  $V \in \mathbb{R}^{m \times n}$ ,  $0 \preceq W$ ,

$$\begin{array}{ll}\text{minimize} & \|V - WH\| \\ \text{subject to} & 0 \preceq W, 0 \preceq H \\ \text{where} & W \in \mathbb{R}^{m \times k}, H \in \mathbb{R}^{k \times n}\end{array}$$

$k$  is the rank of the decomposition and can either be specified or found using the Root Mean Squared Error (the square root of the MSE), SVD, Non-negative Least Squares, or cross-validation techniques.

For this lab, we will use the Frobenius norm, given by

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}.$$

It is equivalent to the square root of the sum of the diagonal of  $A^H A$ .

**Problem 1.** Create the `NMFRecommender` class, which will be used to implement the NMF algorithm. Initialize the class with the following parameters: `random_state` defaulting to 15, `tol` defaulting to `1e-3`, `maxiter` defaulting to 200, and `rank` defaulting to 2.

Add a method called `_initialize_matrices` that takes in  $m$  and  $n$ , the dimensions of  $V$ . Set the random seed so that initializing the matrices can be replicated.

```
>>> np.random.seed(self.random_state)
```

Then, using `np.random.random`, initialize  $W$  and  $H$  with randomly generated numbers between 0 and 1, where  $W \in \mathbb{R}^{m \times k}$  and  $H \in \mathbb{R}^{k \times n}$ . Return  $W$  and  $H$ .

Finally, add a method called `_compute_loss()` that takes as parameters  $V$ ,  $W$ , and  $H$  and returns the Frobenius norm of  $V - WH$ .

## Multiplicative Update

After initializing  $W$  and  $H$ , we iteratively update them using the multiplicative update step. There are other methods for optimization and updating, but because of the simplicity and ease of this solution, it is widely used. As with any other iterative algorithm, we perform the step until the `tol` or `maxiter` is met.

$$H_{ij}^{s+1} = H_{ij}^s \frac{((W^s)^T V)_{ij}}{((W^s)^T W^s H^s)_{ij}} \quad (25.1)$$

and

$$W_{ij}^{s+1} = W_{ij}^s \frac{(V(H^{s+1})^T)_{ij}}{(W^s H^{s+1} (H^{s+1})^T)_{ij}} \quad (25.2)$$

**Problem 2.** Add a method to the `NMF` class called `_update_matrices` that takes as inputs matrices  $V$ ,  $W$ ,  $H$  and returns  $W_{s+1}$  and  $H_{s+1}$  as described in Equations 25.1 and 25.2.

**Problem 3.** Finish the NMF class by adding a method `fit` that finds an optimal  $W$  and  $H$ .

It should accept `V` as a numpy array, perform the multiplicative update algorithm until the loss is less than `tol` or `maxiter` is reached, and return  $W$  and  $H$ .

Call the function `_initialize_matrices()` in order to run `_update_matrices()` and `_compute_loss()`.

Finally add a method called `reconstruct` that reconstructs and returns  $V$  by multiplying  $W$  and  $H$ . You may assume that `fit` has already been run (so `self.V` and `self.W` will already be initialized, in particular).

## Using NMF for Recommendations

Consider the following marketing problem where we have a list of five grocery store customers and their purchases. We want to create personalized food recommendations for their next visit. We start by creating a matrix representing each person and the number of items they purchased in different grocery categories. So from the matrix, we can see that John bought two fruits and one sweet.

$$V = \begin{pmatrix} John & Alice & Mary & Greg & Peter & Jennifer \\ 0 & 1 & 0 & 1 & 2 & 2 \\ 2 & 3 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 2 & 3 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

After performing NMF on  $V$ , we'll get the following  $W$  and  $H$ .

$$W = \begin{pmatrix} Component1 & Component2 & Component3 \\ 2.1 & 0.03 & 0. \\ 1.17 & 0.19 & 1.76 \\ 0.43 & 0.03 & 0.89 \\ 0.26 & 2.05 & 0.02 \\ 0.45 & 0. & 0. \end{pmatrix} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

$$H = \begin{pmatrix} John & Alice & Mary & Greg & Peter & Jennifer \\ 0.00 & 0.45 & 0.00 & 0.43 & 1.0 & 0.9 \\ 0.00 & 0.91 & 1.45 & 1.9 & 0.35 & 0.37 \\ 1.14 & 1.22 & 0.55 & 0.0 & 0.47 & 0.53 \end{pmatrix} \begin{array}{l} Component1 \\ Component2 \\ Component3 \end{array}$$

$W$  represents how much each grocery feature contributes to each component; a higher weight means it's more important to that component. For example, component 1 is heavily determined by vegetables followed by fruit, then coffee, sweets and finally bread. Component 2 is represented almost entirely by bread, while component 3 is based on fruits and sweets, with a small amount of bread.  $H$  is similar, except instead of showing how much each grocery category affects the component, it shows a much each person belongs to the component, again with a higher weight indicating that the person belongs more in that component. We can see the John belongs in component 3, while Jennifer mostly belongs in component 1.

To get our recommendations, we reconstruct  $V$  by multiplying  $W$  and  $H$ .

$$WH = \begin{pmatrix} John & Alice & Mary & Greg & Peter & Jennifer \\ 0.0000 & 0.9723 & 0.0435 & 0.96 & 2.1105 & 1.9011 \\ 2.0064 & 2.8466 & 1.2435 & 0.8641 & 2.0637 & 2.0561 \\ 1.0146 & 1.3066 & 0.533 & 0.2419 & 0.8588 & 0.8698 \\ 0.0228 & 2.0069 & 2.9835 & 4.0068 & 0.9869 & 1.0031 \\ 0.0000 & 0.2025 & 0.0000 & 0.1935 & 0.45 & 0.405 \end{pmatrix} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

Most of the zeros from the original  $V$  have been filled in. This is the **collaborative filtering** portion of the algorithm. By sorting each column by weight, we can predict which items are more attractive to the customers. For instance, Mary has the highest weight for bread at 2.9835, followed by fruit at 1.2435 and then sweets at .533. So we would recommend bread to Mary.

Another way to interpret  $WH$  is to look at a feature and determine who is most likely to buy that item. So if we were having a sale on sweets but only had funds to let three people know, using the reconstructed matrix, we would want to target Alice, John, and Jennifer in that order. This gives us more information than  $V$  alone, which says that everyone except Greg bought one sweet.

**Problem 4.** Use the `NMFRecommender` class to run NMF on  $V$ , defined above, with 2 components. Return  $W$ ,  $H$  as matrices, and the number of people who have higher weights in component 2 than in component 1 as a float.

## Sklearn NMF

Python has a few packages for recommendation algorithms: Surprise, CaseRecommender and of course SkLearn. They implement various algorithms used in recommendation models. We'll use SkLearn, which is similar to the `NMFRecommender` class, for the last problems.

```
from sklearn.decomposition import NMF

>>> model = NMF(n_components=2, init="random", random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
```

As mentioned earlier, many big companies use recommendation systems to encourage purchasing, ad clicks, or spending more time in their product. One famous example of a Recommendation system is Spotify's Discover Weekly. Every week, Spotify creates a playlist of songs that the user has not listened to on Spotify. This helps users find new music that they enjoy and keeps Spotify at the forefront of music trends.

**Problem 5.** Read the file `artist_user.csv` as a pandas dataframe with `index_col=0`. The rows represent users, with the user id in the first column, and the columns represent artists. For each artist  $j$  that a user  $i$  has listened to, the  $ij$  entry contains the number of times user  $i$  has listened to artist  $j$ .

Identify the rank, or number of components to use. Ideally, we want the smallest rank that minimizes the error. However, this rank may be too computationally expensive, as in this situation.

We'll choose the rank by using the following method:

1. First, calculate the frobenius norm of the dataframe and multiply it by 0.0001 (this will be our benchmark value)
2. Next, iterate through `rank= 10, 11, 12, 13, ...` and for each iteration:

- (a) Run NMF using `n_components=rank`, `init="random"`, `random_state=0` and `max_iter=500`
- (b) Reconstruct the matrix  $V$
- (c) Calculate the root mean square error (RMSE) by taking the square root of the MSE – calculated by `sklearn.metrics.mean_squared_error` – of the original dataframe and the reconstructed matrix  $V$
- (d) If the RMSE is less than the benchmark value, stop
3. Return the rank (as an integer) and the reconstructed matrix of this rank

Hint: the optimal rank can be found between 10 and 15, so you only actually need to iterate through `rank= 10, ..., 15`.

**Problem 6.** Write a function `discover_weekly` that takes in a user id and the reconstructed matrix from Problem 5, and returns a list of 30 artists to recommend as strings.

This list of strings should be sorted so that the first artist is the recommendation with the highest weight and the last artist is the least, and it should not contain any artists that the user has already listed to. Use the file `artists.csv` with `index_col=0`, to match the artist ID to their name.

As a check, the Discover Weekly for user 2 should return

```
[‘Britney Spears’, ‘Avril Lavigne’, ‘Rihanna’, ‘Paramore’, ‘Christina Aguilera’,
‘U2’, ‘The Devil Wears Prada’, ‘Muse’, ‘Hadouken!’, ‘Ke$ha’, ‘Good Charlotte’,
‘Linkin Park’, ‘Enter Shikari’, ‘Katy Perry’, ‘Miley Cyrus’, ‘Taylor Swift’,
‘Beyoncé’, ‘Asking Alexandria’, ‘The Veronicas’, ‘Mariah Carey’, ‘Martin L. Gore’,
‘Dance Gavin Dance’, ‘Erasure’, ‘In Flames’, ‘3OH!3’, ‘Blur’, ‘Kelly Clarkson’,
‘Justin Bieber’, ‘Alesana’, ‘Ashley Tisdale’]
```



# 26 Deep Learning

**Lab Objective:** *Deep Learning is a popular method for machine learning tasks that have large amounts of data, including image recognition, voice recognition, and natural language processing. In this lab, we use PyTorch to write a convolution neural net to classify images. We also look at one of the challenges of deep learning by performing an adversarial attack on our model.*

## Intro to Neural Networks

An *artificial neural network* is a machine learning tool inspired by the idea of neurons passing information between each other to learn. The network is composed of layers of neurons, usually called *nodes*, that are connected in various ways. Each connection has a *weight* based on its importance, which is used as information is passed through the network from one layer to the next. For example, in Figure 26.1, the yellow input is passed to the first layer, blue, then the green layer, and then to the final output layer.

The middle layers for a neural network are considered “hidden” because they’re not directly viewable to the outside world. You can view them but they will be a mess of seemingly random numbers, not the helpful classification labels you would get from an end layer.

In a neural network, the input is often images, text, or sounds represented as a vector of real numbers. In order to evaluate a network, these values are then multiplied by the weight of each pertinent edge, and all respective values are added up to create the node value. A vector called the *bias* of the layer is added to each respective node, finalizing the linear combination step. An *activation function* takes these node values and applies a nonlinear transformation to them, which allows the model to learn complex nonlinear transformations between the input and output. Without these activation functions, the network would be linear and exactly the same as a network with no hidden layers. Mathematically, a typical neural network looks like the nested function composition

$$f_N(\mathbf{x}) = \mathbf{a}(W_k \mathbf{a}(\dots \mathbf{a}(W_2 \mathbf{a}(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \dots) + \mathbf{b}_k)$$

where  $\mathbf{a}$  is the activation function, and  $W_i$  and  $\mathbf{b}_i$  are the weights and biases of each layer of the neural network. The model is trained by adjusting the weights and biases, typically using a variant of gradient descent, until the model output accurately matches the training labels.

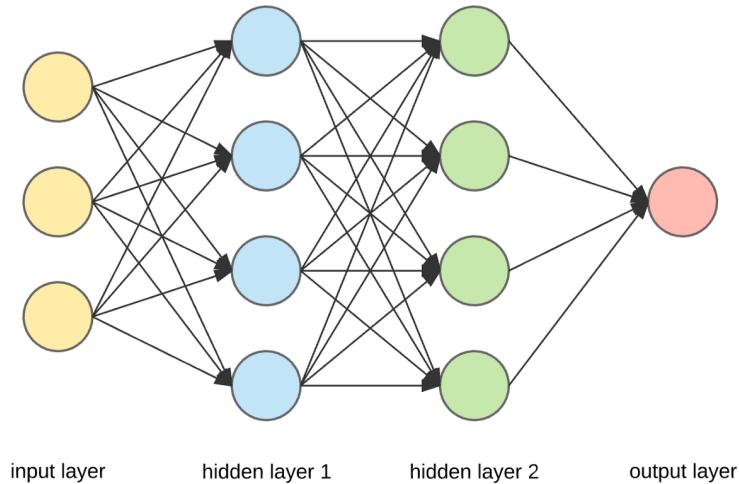


Figure 26.1: A high level diagram of an artificial neural network.

## Intro to PyTorch

PyTorch is an open source machine learning library developed by Facebook AI Research. It's mainly used for fast GPU processing of deep neural networks (neural networks with many hidden layers). GPUs (graphics processing units) are designed to compute thousands of operations at once and are vital for parallelizing the operations used by neural networks, which is why they are used here. For more information and documentation on PyTorch, visit <https://pytorch.org/>

We will be working in Google's Colaboratory, <https://colab.research.google.com/notebooks/intro.ipynb>. Colab notebooks use Google's cloud servers, which have a built-in GPU. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a popup called **Notebook Settings**. Under *Hardware Settings*, select GPU.

We can verify that GPU is enabled by calling `torch.cuda.is_available()`. If this function returns `False`, a GPU is not available, and the code will be run on the CPU.

### ACHTUNG!

Colab has a limit on the amount of GPU time available, and the longer one uses their GPUs on the same Google account, the less priority one has. As we will use this feature for both this lab and the Recurrent Neural Networks lab, it is advisable to not leave the GPU active while you are not using it. Instead, switch to CPU while you are developing your code, then only switch to GPU once you're ready to fully train your model (we'll explain what that means later).

Note that enabling or disabling the GPU resets the notebook environment, so be sure that you save the results of any important computations before doing so, and you may need to re-upload data to your session too. Also be aware that along with the limit on GPU time, Colab sessions also time out and reset after a period of user inactivity, even if cells are running and independent of whether CPU or GPU is being used.

If you still run out of GPU time, Kaggle—a platform for data science and data science competitions—also offers free GPU time in a notebook environment very similar to Colab's.

CUDA is a parallel computing platform for GPU computing. The PyTorch package `torch.cuda` interfaces with this package and allows code to be run on the GPU.

PyTorch represents vectors and arrays using *tensors*. A tensor is a data structure similar to a numpy array that is designed to be compatible with GPUs. Like a numpy array, it has a shape, data type, and can be multi-dimensional.

In order for a tensor to be used by the GPU, it must be stored on the GPU. A tensor can be sent directly to the GPU using `variable.cuda()`; however, if the GPU is not available, this will cause an error. A more flexible approach is store which device we are doing computations on as a variable `device`. Then, we can send our variables to the correct location in both cases by using `variable.to(device)`:

```
>>> import torch

>>> x = torch.tensor([3., 4.])                      # Create tensor on CPU
>>> y = torch.tensor([1., 2.]).cuda()                # Create tensor on GPU

# Create the device, choosing GPU if available
>>> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
>>> z = torch.tensor([1., 2.]).to(device)            # Create tensor on device
```

You can check which device a variable is on by displaying it. If it is on a GPU, it will list which number it is. `cuda:0` means that the device running is the default GPU. If you are using a machine that has multiple GPUs, you can set the device to be a specific GPU by changing the number. In Colab, only `cuda:0` is available.

```
>>> x
tensor([3., 4.])                                     # Check location of x (CPU)

>>> x = x.to(device)                                # Move x to GPU
>>> x
tensor([3., 4.], device="cuda:0")                    # Check location of x (GPU 0)
```

## ACHTUNG!

Cross-GPU operations are not allowed. This means that the model and data must all be on the same device. If the model is called on data that is on a different device, say the model is located on the GPU and the data is on the CPU, you will get the following runtime exception:

```
RuntimeError: Input type (torch.FloatTensor) and weight type (torch.cuda.FloatTensor) should be the same.
```

If you get this error, you will need to move one the variables so that they are all on the same device.

## Data

For this lab, we will be using the CIFAR10 dataset. It consists of 60,000 images of size  $32 \times 32$ , represented as a  $3 \times 32 \times 32$  matrix, where the 3 channels describe the colors using RGB. The images are evenly split into ten classes represented by the numbers 0 – 9: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

For convenience, the dataset is already split into a training and a testing set; however, we will also add a *validation* set.

Using a train-validate-test split is good practice in general, because usually we will want to test and compare different models and hyperparameter choices iteratively until we arrive at one that works well for our problem. Once we are done training, we would like to use the test set to determine how well our model fits the data. However, if we use the test set to compare how well each of our models performs, it effectively becomes a second train set that we are learning by trying different models, and using the test set to determine how well our model works on the whole dataset is no longer really valid. As such, it is better practice to use a three-way split. We train each model on the train set, use the validation set to compare the models, and use the test set to determine if our final model appears to fit the data well.

The CIFAR10 dataset is split into a train set of 50,000 images and a test set of 10,000 images. We will split the original train set into a new train set of 40,000 images and a validation set of 10,000 images. To use the data, we must transform it into PyTorch tensors. We also will normalize the data, as this generally improves the results. We will normalize the values to have mean 0 and standard deviation 1 for each component. Finally, we will split the dataset and place it inside a `torch.utils.data.DataLoader` class for easier manipulation.

To load the dataset, we use the `torchvision.datasets.CIFAR10` function, which accepts a folder for the data to be stored in. Some important keyword arguments are listed in Table 26.1.

Parameter	Explanation
<code>train</code>	Whether to get the training data or the test data.
<code>download</code>	Whether to download the data. You usually only need this the first time you access the dataset. Note that restarting Google Colab will require re-downloading the data, however.
<code>transform</code>	Applies the given <code>transform</code> when loading the data. This transform always should convert the data into a PyTorch tensor.

Table 26.1: Parameters of the `datasets.CIFAR10` loading function

We can use the `transform` parameter in particular to easily normalize our data. PyTorch has a module `torchvision.transforms` to make creating these transformations easier. In this case, we want to use `transforms.ToTensor` to convert the data into tensors, and then `transforms.Normalize` to normalize the data. The `Normalize` object accepts the desired mean and standard deviation after normalization. We can use `transforms.Compose` to combine these together into a single transform object:

```
>>> from torchvision import transforms

# Transform data into a tensor and normalize
>>> transform = transforms.Compose([
```

```
...     transforms.ToTensor(),
...     transforms.Normalize((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))
...])
```

We can then load the data:

```
>>> from torchvision import datasets

# Download the CIFAR10 training data to ../data
>>> train_data = datasets.CIFAR10("../data", train=True, download=True, ←
    transform=transform)
```

The data can then be accessed using indexing. Each data point is a tuple consisting of the  $3 \times 32 \times 32$  image and its class. You can also see the specs of the dataset by calling it without an index.

```
# Get the first training data point
>>> train_data[0]
(tensor([[[ 0.2314, ..., 0.5804],
          [ 0.0627, ..., 0.4784],
          ...,
          [ 0.7059, ..., 0.3255],
          [ 0.6941, ..., 0.4824]],

         [[ 0.2431, ..., 0.4863],
          [ 0.0784, ..., 0.3412],
          ...,
          [ 0.5451, ..., 0.2078],
          [ 0.5647, ..., 0.3608]],

         [[ 0.2471, ..., 0.4039],
          [ 0.0784, ..., 0.2235],
          ...,
          [ 0.3765, ..., 0.1333],
          [ 0.4549, ..., 0.2824]]]), 6)

# Get the class of the first training data point
>>> train_data[0][1]
6

# Get the specs of the CIFAR10 training set
>>> train_data
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ../data
    Split: Train
    StandardTransform
    Transform: Compose(
        ToTensor()
        Normalize(mean=(0.0, 0.0, 0.0), std=(1.0, 1.0, 1.0))
```

)

**Problem 1.** Create the `device` variable as indicated above. Download the CIFAR10 training and test datasets, transform them into tensors, and normalize them as described above.

PyTorch has a special class `DataLoader` that splits the data into batches for easy manipulation. Sending individual data points to the GPU one at a time to be processed by our model is very inefficient, as it makes it impossible for the GPU to parallelize the computations. Instead, we use *batches*, and send multiple data points together. Using larger batch sizes allows us to take advantage of GPUs, speeding up the training time. Storing all of the data on the GPU is, however, generally impossible due to memory constraints. Using too large of a batch size will cause out of memory issues, and tends to reduce the effectiveness of training. Typical batch sizes are powers of 2: 32, 64, 128, 256.

The `DataLoader` class accepts the dataset as its first argument. The dataset can be a dataset object like the one we created above, a list containing the data points, or any iterable. For the train set, we will first split the loaded data into two lists to create the actual train and validation sets. The dataset object does *not* support fancy indexing, so this step should be done with list comprehension:

```
>>> actual_train_data = [train_data[i] for i in range(40_000)]  
  
>>> from torch.utils.data import DataLoader  
  
# Create a DataLoader from the shuffled training data  
>>> train_loader = DataLoader(actual_train_data, batch_size=36, shuffle=True)  
# and similarly for the validation set
```

The data is not ordered by its classes, so directly indexing like this will put a good mixture of all of the classes into both sets. For the test set, we can just directly pass the dataset object into the `DataLoader`. Some other useful parameters of the `DataLoader` class are listed in Table 26.2.

Parameter	Explanation
<code>batch_size</code>	The size of batch to use
<code>shuffle</code>	Whether to shuffle the data
<code>num_workers</code>	The number of processes to use, in order to load the data in parallel

Table 26.2: Parameters of the `DataLoader` object

Once we have the data in the `DataLoader` class, we can iterate through it to get data points. We can turn it into an iterator using the `iter` method, and then get batches one-at-a-time using the `next` method:

```
# Get the 36 images of size 3x32x32 and labels in the first batch  
>>> dataiter = iter(train_loader)  
>>> images, labels = next(dataiter)  
>>> images.size()  
torch.Size([36, 3, 32, 32])
```

```
>>> images[0].size()
torch.Size([3, 32, 32])

>>> labels[0]
tensor(8)
```

This method is particularly useful if we just need a few images. We can also directly iterate through all of the images using a `for` loop:

```
>>> for batch, (x, y_truth) in enumerate(train_loader):
...     # Move to the GPU
...     x, y_truth = x.to(device), y_truth.to(device)
...     # ...
```

This will be a more convenient method for training.

**Problem 2.** Split the data into train, validate, and test sets, and create DataLoaders for each one. The train set should have 40,000 data points and the test and validate sets should each have 10,000 data points. Use a batch size of 32 for the training set and 1 for the validation and test sets. Specify `shuffle=True` for the training set, and `shuffle=False` for the validation and test sets (this is common practice in deep learning).

## Neural Networks in PyTorch

Before creating a good model for this dataset, we will start with a simple model to illustrate how to set up a neural network in PyTorch. This model will use only fully-connected linear layers and activation functions. First, we need to import the `nn` module, which contains all of the classes we need for this:

```
from torch import nn
```

### Simple layers

A linear layer takes an input vector  $x$  and outputs  $Ax + b$  for a learned weight matrix  $A$  and bias vector  $b$ . This is implemented in Pytorch as `nn.Linear(in_features, out_features)`, where `in_features` is the length of the input vector and `out_features` is the desired length of the output vector. This is called a *fully-connected* layer, because every entry of  $A$  and  $b$  are allowed to be nonzero.

After each layer, we want to pass the values through an *activation function*. This allows the model to be nonlinear, allowing it to learn much more complicated behaviors than it would otherwise. The most commonly used activation function is the Rectified Linear Unit (ReLU) function:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

This activation function avoids many issues that other activation functions have, and is used almost universally. For the final activation function, however, we will use a different activation function: the *softmax* function

$$\text{Softmax}(x_1, \dots, x_n) = \left( \frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \dots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}} \right).$$

The components of the output of the softmax function are all non-negative and sum to 1. This allows the output of the final layer to be interpreted as probabilities, which is useful for classification. The component with the highest probability will be the neural network's prediction for the input image. This also enables the use of cross-entropy as a very natural loss function, which will be discussed later. These two activation functions are available as `nn.ReLU` and `nn.Softmax`.

### ACHTUNG!

The cross-entropy loss function from PyTorch, `nn.CrossEntropyLoss`, combines logarithmic softmax `log_softmax` and the negative log likelihood loss `NLLLoss`, which means you should NOT apply softmax as the last layer of your model.

Performing softmax twice can cause numerical instability in the model's accuracy.

## Creating a model

To create a neural network in PyTorch, we begin by creating a class that inherits from `nn.Module`:

```
class NNExample(nn.Module):
```

The class `nn.Module` handles internals so that training is simpler, as well as providing a variety of useful methods. In the initializer of our class, we need to call the *superconstructor* `super().__init__()` to initialize the `nn.Module` itself. Then, we initialize all of the layers we want to use in our model. For this example, we will use two fully-connected linear layers with activation functions after each. Since our inputs are  $3 \times 32 \times 32$  tensors and linear layers only work with vectors, we will also include an `nn.Flatten` layer.

```
def __init__(self):
    # Initialize nn.Module
    super().__init__()

    # Create our layers
    self.flatten = nn.Flatten()
    self.linear1 = nn.Linear(in_features=3*32*32, out_features=100)
    self.relu = nn.ReLU()
    self.linear2 = nn.Linear(in_features=100, out_features=10)
```

We need to set all of these layers as members of our class in order for them to be properly detected in the training process. Notice that the input dimension of the first layer (`linear1`) is equal to the dimension of the (flattened) input image ( $3 \times 32 \times 32$ ), but from there, the output dimension is chosen arbitrarily to be 100. The second layer (`linear2`) must then have input dimension equal to the output dimension of `linear1`, but its output dimension is chosen to be 10, which is the number of classes possible in the CIFAR10 dataset.

Lastly, we define the `forward()` method, which calls all of the layers on an input image to give us the output:

```
def forward(self, x):
    x_flat = self.flatten(x)
    x_layer1 = self.relu(self.linear1(x_flat))
    output = x_layer1
    return output
```

Even though each layer is really a class (note how we initialize them in `__init__()`), we can call them as if they are functions. Any layer that contains *learnable parameters* (for example, the weights present in linear layers), must be called individually in the `forward()` method, as otherwise this would force it to reuse the parameters and reduce training effectiveness. However, for layers that do not have learnable parameters, such as activation functions, we can safely reuse them and call them multiple times in the `forward()` method. Hence, `nn.ReLU()` only needs to be defined once in the `__init__()` method, even when it may be called multiple times in the `forward()` method.

This neural network would likely perform very poorly on the CIFAR10 dataset, however; only using fully-connected linear layers does not work well for images. For a better method, we will turn to *convolutional neural networks*.

Convolutional neural networks (CNNs) are a type of neural network that use *convolution layers*. They also commonly use *pooling layers*. They are particularly well-suited to working with images, such as the CIFAR10 dataset. We now discuss these components and how to use them in PyTorch.

## Convolution Layers

A convolution layer takes a two-dimensional array of weights called a *kernel* (sometimes called a filter) and multiplies it by the input at all possible locations, “sliding” around the input. It is particularly useful when working with images, as it preserves and extracts spacial structures, unlike fully-connected linear layers.

Consider the following  $5 \times 5$  input image and  $3 \times 3$  kernel:

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

$5 \times 5$  Input Image

1	0	-1
1	0	-1
1	0	-1

$3 \times 3$  Kernel

To get each value in the output, the kernel is multiplied element-wise by  $3 \times 3$  squares inside the input and summed. For the top left square in this example, the output is

$$2 \cdot 1 + 4 \cdot 0 + 7 \cdot (-1) + 9 \cdot 1 + 7 \cdot 0 + 1 \cdot (-1) + 8 \cdot 1 + 3 \cdot 0 + 4 \cdot (-1) = 7.$$

$2 \cdot 1$	$4 \cdot 0$	$7 \cdot (-1)$	6	2
$9 \cdot 1$	$7 \cdot 0$	$1 \cdot (-1)$	2	1
$8 \cdot 1$	$3 \cdot 0$	$4 \cdot (-1)$	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Images

7	...	
:		

3×3 Output

The 7 represents a feature of the  $3 \times 3$  block in the top left corner. With a trained network applied to the image, these features can represent things such as lines, curves, and colors, or even more complicated objects like a nose.

The *stride* of a convolution is how much the kernel slides at a time as it passes over the input. For example, if the kernel slides one spot over (has a stride of 1), there will be 9 submatrices inside the input image that will be used, and we would get a  $3 \times 3$  matrix as output. If we used a stride of 2 instead, only the top-left, top-right, bottom-left, and bottom-right submatrices would be used.

Notice that as the kernel slides around the image, the inside values are used in more multiplications than the outside value, causing us to lose information, especially about the corners. If we want to keep more information about the edges of the image, we can use *padding*. Padding consists of adding a border around the input, usually filled with zeros. This can also allow the output of the layer to be the same size as the input.

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

5×5 input image

0	0	0	0	0	0	0
0	2	4	7	6	2	0
0	9	7	1	2	1	0
0	8	3	4	5	8	0
0	4	3	3	1	2	0
0	5	2	1	5	3	0
0	0	0	0	0	0	0

5×5 input image padded with 0

Each dimension of the output for a convolution layer is calculated as follows:

$$\frac{\text{input size} - \text{kernel size} + 2 \cdot \text{padding size}}{\text{stride}} + 1$$

In our example with stride 1, kernel size 3, and no padding, the output size is  $(5 - 3 + 2 \cdot 0)/1 + 1 = 3$ . It is good to ensure that the stride always divides the numerator, as otherwise the kernel will be applied asymmetrically to the image. Calculating the output dimension of a layer is necessary since the following layer will have its *input* dimension size equal to the previous *output* dimension size.

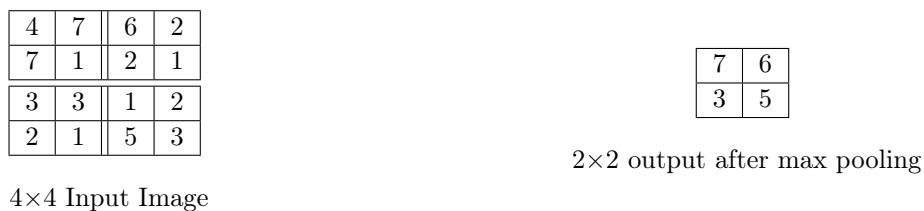
One last feature that we need to discuss is *channels*. Image data (including ours) typically has three channels, representing the red, green, and blue contents of pixels. Channels act like different “layers” of the image or of the convolutional output. In a convolutional layer, there is one kernel for each pair of input channel and output channel. If we have  $n$  input and  $m$  output channels, then we will have  $mn$  total kernels that are learned individually. Each kernel is applied to its corresponding input channel as described above. Then, each output channel is determined as the sum of the results of the convolutions of all of its kernels, plus a learned bias value.

Convolutional layers are represented in PyTorch with `nn.Conv2d`. The constructor of this class requires three parameters `in_channels`, `out_channels`, and `kernel_size`. It also has optional arguments `stride` (default 1) and `padding` (default 0). Note that for each of the kernel size, stride, and padding, only an integer needs to be specified, and it will be used for both the x and y directions. The following creates a convolutional layer that accepts an image with 3 channels, output 8 channels, and uses a  $3 \times 3$  kernel with stride 1 and no padding:

```
layer = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3)
```

## Pooling layers

Pooling layers are used to reduce the size of the image while retaining important information. The input image is broken into small pieces, called *pools*, each of which is condensed to a single number. The most common form of pooling is *max pooling*, where the output of each pool is just the maximum of its inputs.



Max pooling has the particularly nice property of making the output remain similar if the input image is shifted slightly.

Max pooling layers are represented in PyTorch as `nn.MaxPool2d`. They accept a single parameter `kernel_size`; this is the size of each of the pools. Using  $2 \times 2$  pools is the most common. The following creates a pooling layer that uses a  $2 \times 2$  pool size:

```
layer = nn.MaxPool2d(kernel_size=2)
```

## Parameters

When working with neural networks, it can be useful to know how many learnable parameters our model has. This particularly dictates the amount of space needed to store the model. The number of parameters in each layer depends on the type of layer and its input and output sizes. Table 26.3 lists how to calculate this number for the layer types we have discussed.

For example, the example neural network above would have

$$\begin{aligned} &(\text{first linear layer}) \quad (3 \cdot 32 \cdot 32 + 1) \cdot 100 \\ &(\text{second linear layer}) \quad + (100 + 1) \cdot 10 = 308310 \text{ parameters,} \end{aligned}$$

and the example convolutional layer would have

$$(3 \cdot 3^2 + 1) \cdot 8 = 224 \text{ parameters.}$$

Layer type	Number of Parameters
Linear	$(\text{in\_features} + 1) * \text{out\_features}$
Convolutional	$(\text{in\_channels} \cdot \text{kernel\_size}^2 + 1) * \text{out\_channels}$
Pooling	No parameters
Flatten	No parameters
Activation functions	No parameters

Table 26.3: Parameter counts for layer types we use in this lab

**Problem 3.** Create a class for a convolutional neural network that accepts images as  $3 \times 32 \times 32$  tensors and returns 1D tensors of length 10, representing its predicted probabilities of each class. Include at least the following:

- Three convolutional layers, each followed by an activation function
- A max pooling layer
- Two linear layers

Choose the size of the layers so that your model has at least 50,000 parameters (use Table 26.3), and print out this calculation in the Jupyter notebook file. In practice, specifications of model architecture (i.e. number of layers, layer sizes, etc.) are chosen quite arbitrarily until something works. As such, you may customize your model architecture to your liking, provided your model meets the requirements specified above.

Hint: It can be very helpful to keep track of the size of the image after each step, so you know what the input size should be for the next step. The max pooling layer should occur immediately after a convolutional layer is passed through an activation function. Additionally, you will need a `nn.Flatten` layer after the convolutional layers and before the linear layers. You can check that your model works correctly by passing an (unsqueezed) image through it as demonstrated in the following code:

```
>>> model = NNExample()
>>> model(images[0].unsqueeze(0))
tensor([[0.0952, 0.1120, 0.1019, 0.0992, 0.0955, 0.0984, 0.0886, 0.1326, ←
    0.0966, 0.0800]], grad_fn=<AddmmBackward0>)
```

Note that your neural network will predict different probabilities for each of the categories.

## Training the Model

Now that we have data and a model all set up, we need to train the model on the data. We do this by iterating through the `DataLoader`, calling the model on the data, determining how well the model classified the data, and then optimizing the model weights. We use a loss function, called the *objective*, to calculate the loss, which is the difference between the model's predicted labels and the actual labels of the data. A common classification loss function is Cross Entropy Loss

$$L_{CE} = - \sum_i a_i \log(p_i),$$

where  $i$  represents each data point,  $p_i$  is the softmax probability for each data point, and  $a_i$  is the label for each data point. PyTorch's `nn.CrossEntropyLoss()` conveniently handles all of this.

Once the loss is calculated by the objective, we can use it to optimize the model weights to make the loss smaller. This can be done through *backpropagation*, which calculates the partial derivatives of the loss function with respect to each weight, and then uses gradient descent to update every weight. PyTorch has several predefined methods for optimization, but we'll use the popular Adam algorithm. PyTorch accumulates gradients when backpropagating, which is sometimes desireable, but in our case it would cause the loss to increase. To prevent this, we need to zero out the gradients before we perform each backpropagation. PyTorch streamlines this entire training sequence in a very clean way, as shown in the following:

```
>>> import torch.optim as optim

# Define the objective and optimizer
>>> objective = nn.CrossEntropyLoss()
>>> optimizer = optim.Adam(model.parameters(), lr=1e-4)

# For each iteration of the DataLoader, do the following
>>> optimizer.zero_grad()           # Zero out the gradients
>>> y_hat = model(x)             # Predict labels
>>> loss = objective(y_hat, y_truth) # Calculate loss
>>> loss.backward()               # Backpropagate to compute gradients
>>> optimizer.step()              # Optimize and update the weights
```

An *epoch* is a complete training sequence that trains over the entire DataLoader. To improve the model's accuracy, we can train over many epochs. A good guideline is to train the model for the number of epochs it takes for the loss to stop decreasing.

The loss is calculated using the training data, but at the end of each epoch we also want to know how well the model performs with the validation data. Before we determine the validation accuracy, we need to switch our model to evaluation mode so it doesn't continue training. This is done by the simple command `model.eval()`, but it's important to switch the model back to training mode at the start of each epoch using `model.train()`. The validation accuracy is determined by simply iterating through the validation DataLoader, and seeing if the model can correctly predict each data point. The validation accuracy is then computed by dividing the number of correct predictions by the total number of data points in the validation DataLoader.

```
>>> model.eval()                  # switch to evaluation mode
>>> validation_score = 0
>>> for x, y_truth in validation_loader:
>>>     x, y_truth = x.to(device), y_truth.to(device)
>>>     y_hat = model(x)
>>>     if y_truth == y_hat.argmax(1): # compare with greatest probability
>>>         validation_score += 1
>>> validation_accuracy = validation_score / len(validation_loader)
```

The validation accuracy does not determine the model's final accuracy. The final accuracy is computed in the same way as the validation accuracy, but this time using the testing data, and it's only computed one time, when the model finishes training entirely.

TQDM is a python package that displays the progress of a for-loop, which can help estimate the remaining time. TQDM is initialized outside the loop, then updated inside the loop, as follows:

```
>>> from tqdm import tqdm

>>> loop = tqdm(total=len(train_loader), position=0)

>>> for epoch in range(num_epochs):
>>>     loop.set_description(f"epoch:{epoch}, loss:{loss.item():.4f}")
>>>     loop.update()

>>> loop.close()

epoch:1 loss:1.8585: : 1402it [00:17, 79.69it/s]
```

### ACHTUNG!

Near the beginning of this lab we suggested developing your code while on CPU and only switching to GPU once ready to fully train your model. A good practice is to train your model for two epochs while on CPU before switching to GPU to run the full training procedure from start to finish (and then switching back to CPU if continuing to develop solutions to other problems). This ensures that your model and training loop are functioning correctly, rather than wasting GPU time debugging.

You may still run into bugs when you switch to GPU that didn't surface when on the CPU. These often result from forgetting to specify the correct device for all components, including the model and data (using `variable.to(device)`).

Another strategy to speed up debugging is to test your model on a small subset of the data to ensure things are running and processing correctly, then scaling to larger data for more robust tests. This ensures that simple issues are caught early before extensive time is spent processing large datasets.

**Problem 4.** Send your model to the device and instantiate the objective and optimizer. Train your model with a TQDM display, and calculate the Validation Accuracy after each epoch. Begin by initializing your TQDM loop, then for each epoch, do the following:

1. Set your model to training mode (`model.train()`)
2. Instantiate an empty `loss_list`
3. For each batch in `train_loader`:
  - (a) Send `x` and `y_truth` to device
  - (b) Zero out the gradients
  - (c) Use model to predict labels of `x`

- (d) Calculate loss between predicted labels and `y_truth`
  - (e) Append loss (`loss.item()`) to `loss_list` (the `.item()` feature extracts the element from a tensor with only one element)
  - (f) Update TQDM loop
  - (g) Backpropagate to compute gradients
  - (h) Optimize and update the weights
4. Save the loss mean as the mean of the losses in `loss_list`
  5. Set your model to evaluation mode (`model.eval()`)
  6. Calculate and save validation accuracy

Finish the training by closing your TQDM loop. Train for 10 epochs, saving the mean loss and validation accuracy for each epoch. Plot the mean losses and validation accuracies, which should resemble Figure 26.2. Lastly, print the final test score using the testing data, as described above.

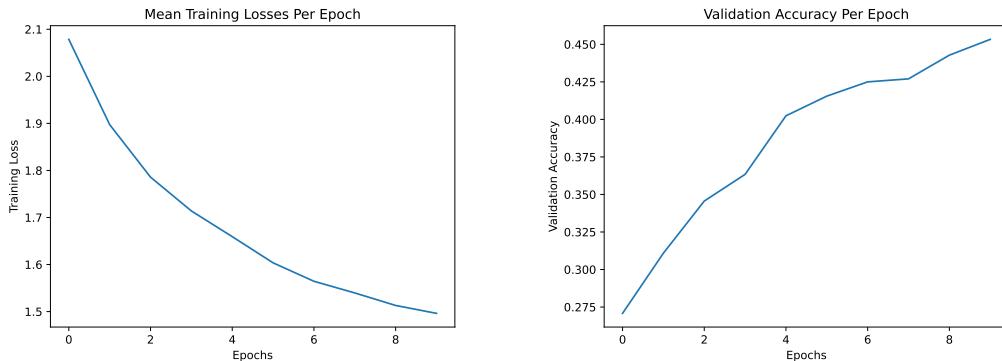


Figure 26.2: Training Loss and Validation Accuracy for a CNN on CIFAR10.

## Adversarial Attacks

Just like any algorithm or software, deep learning is susceptible to attacks. For deep learning models, this vulnerability most often manifests in the model being extremely sensitive to certain types of changes in the input that really should not matter. This results in the model giving nonsensical results, which, while amusing, can cause major problems. Examples of adversarial attacks against neural networks range from adding a small amount of noise to a picture of a panda, resulting in the model classifying the image as a gibbon with 99% confidence [GSS15] to fooling facial recognition by printing a pair of eyeglasses [GKB17]. When designing machine learning models, it is important to be aware of these issues so that their impact can be mitigated.

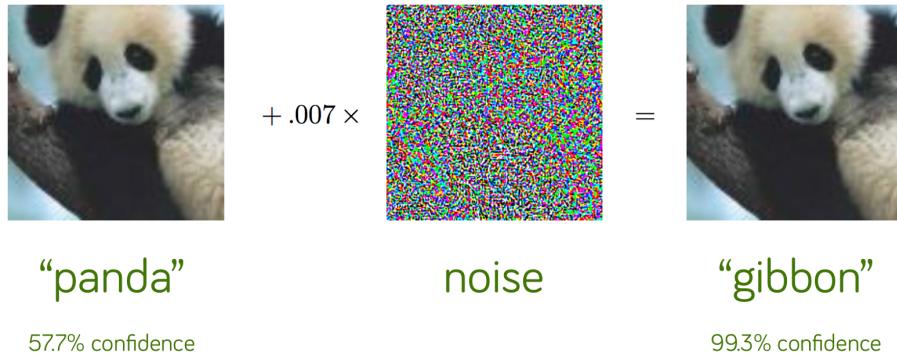


Figure 26.3: A slight modification to a correctly-classified image of a panda results in the model confidently classifying it as a gibbon, despite the image not having changed in any substantial way.

The example of modifying the image of a panda is an attack called the *Fast Gradient Sign Method (FGSM)*. FGSM is a *white-box attack*, meaning that the attacker has access to the model; this is in contrast with a *black-box attack* where the attacker only has access to the model’s inputs and outputs.

During model training, gradients are used to adjust the model weights so that loss is minimized. In FGSM, the gradient is instead used to perturb the input image in a direction that *maximizes* the loss, using the following equation:

$$x_{\text{perturbed}} = x + \varepsilon \text{Sign}(\nabla_x \text{Loss}(\theta, x, y))$$

where  $x$  is the input,  $y$  is the label, and  $\theta$  is the model parameters.

We can calculate this perturbation in PyTorch as follows. To calculate the gradient of the model with respect to a piece of data  $x$ , we first need to set  $x.\text{requires\_grad} = \text{True}$  and call  $x.\text{retain\_grad}()$ . Then, we zero out the optimizer’s gradient, run  $x$  through the model, and compute the loss, similar to training. After this, the gradient of the output with respect to  $x$  can be obtained using the attribute  $x.\text{grad}.\text{data}$ .

The following function `fgsm_attack` accepts a model and an image and performs the FGSM attack, returning the perturbed image:

```

def fgsm_attack(model, optimizer, objective, x, y, eps):
    """
    Performs the FGSM attack on the given model and data point x with label y.
    Returns the perturbed data point.
    """
    # Calculate the gradient
    x.requires_grad = True
    x,retain_grad()
    optimizer.zero_grad()
    output = model(x)
    loss = objective(output, y)
    loss.backward()
    data_grad = x.grad.data
    # Perturb the images
    x_perturbed = x + eps * data_grad.sign()

    return x_perturbed

```

We will use this function to explore this type of adversarial attack on our neural network.

**Problem 5.** Write a function that loops through the test data using the function `fgsm_attack` to perturb the images and using your trained model from Problem 4.

Run your function for each epsilon in  $[0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]$ , and plot epsilon against the model's accuracy.

Display the perturbed version of the first image in the test data for each epsilon, using the following code. Be sure to show the old and new labels for each perturbed image. Make sure the original image is classified correctly. Your figure should look similar to Figure 26.4.

```

# Move the image to cpu and convert to numpy array
>>> ex = perturbed_data.squeeze().detach().cpu().numpy()

# Plot the image
>>> img = ex / 2 + 0.5      # unnormalize
>>> plt.imshow(np.transpose(img, (1, 2, 0)))

```

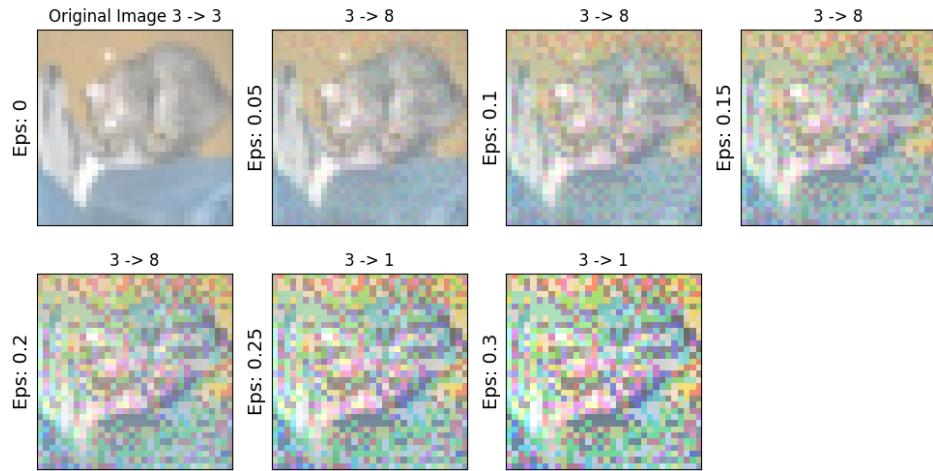


Figure 26.4: The first modified image for different values of epsilon.

## Additional Materials

### TensorBoard

TensorBoard is a visualization toolkit for neural networks. It was originally built for Tensorflow, but also can be used with PyTorch. The main features of TensorBoard include model visualization, dimensionality reduction, tracking and visualizing metrics, and displaying data.

To create a tensorboard, run the following code:

```
>>> import os
>>> from torch.utils.tensorboard import SummaryWriter
>>> %load_ext tensorboard
>>> logs_base_dir = "runs"
>>> os.makedirs(logs_base_dir, exist_ok=True)
>>> %tensorboard --logdir {logs_base_dir}
```

The TensorBoard homepage will show up inline:

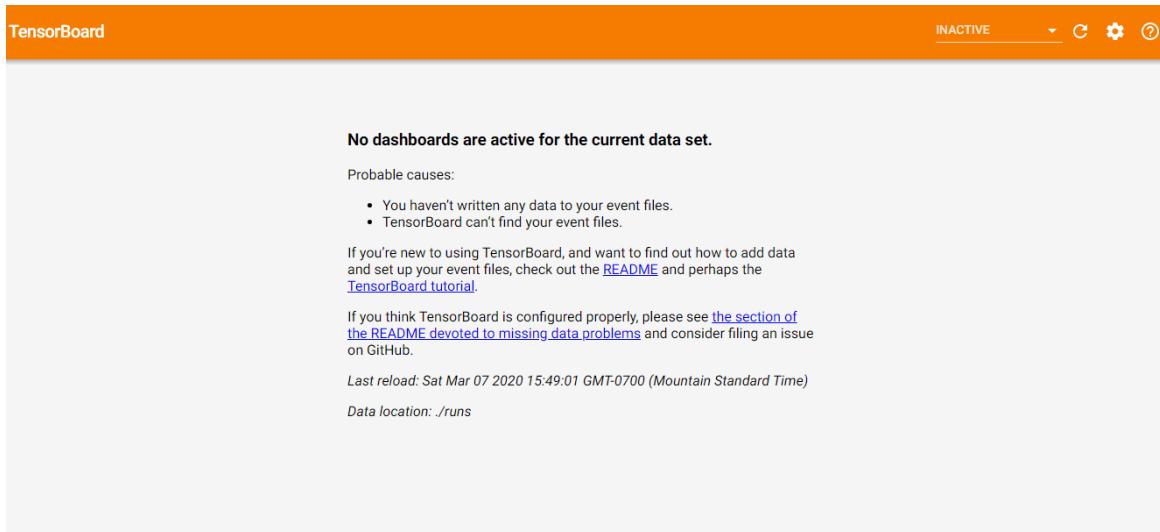


Figure 26.5: The home page of an empty TensorBoard.

We write to TensorBoard using `SummaryWriter`. It writes to files in the `logs_base_dir` that are used by TensorBoard to display information. You can view the `logs_base_dir` directory by selecting the file icon on the far left of the page. For example, we can create an interactive graph of our model.

```
>>> tb = SummaryWriter()
>>> tb.add_images("Image", images)
>>> tb.add_graph(model, images)
>>> tb.close()
```

This updates our TensorBoard with a `GRAPHS` tab, which describes the model. If it doesn't show up automatically, press the refresh button in the top right corner of the TensorBoard. You can explore the model by clicking on the components.

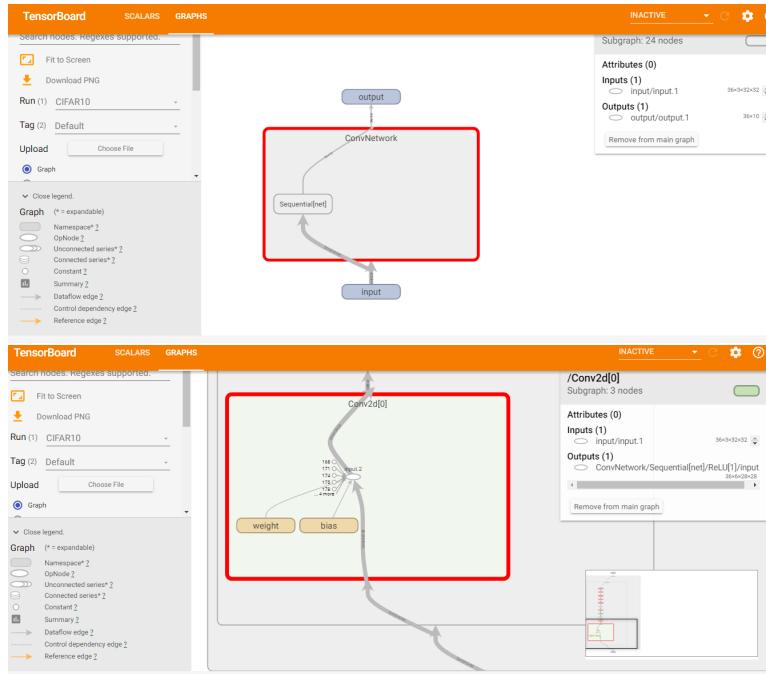


Figure 26.6: Examples of TensorBoard Graph Tab.

The following items can be added to TensorBoard, with more information at <https://pytorch.org/docs/stable/tensorboard.html>.

- add\_scalar/s
- add\_image/s
- add\_figure
- add\_text
- add\_graph
- add\_hparams

To save the training loss, write a function that returns a matplotlib figure of the training loss plot. Then use `tb.add_figure(figure_name, plot_loss())`.

```
writer.add_figure("Training Loss",plot_loss())
```

**Problem 6.** Create a TensorBoard for this project that includes the network, a plot of iterations versus training loss and a plot of iterations versus test accuracy from the training done in Problem 4.

# 27

# Recurrent Neural Networks

**Lab Objective:** *Recurrent Neural Networks are powerful machine learning algorithms that accept sequences as inputs and can process temporal data. In this lab, we generate a Mozart-like piano sonata using the Long Short-term Memory RNN.*

## ACHTUNG!

As is common when working with Neural Networks, this lab requires a large amount of data processing which can be quite time consuming to debug. One of the best ways to mitigate this is to debug your code using small subsets of the data rather than the entire dataset; that way, you can quickly catch simple errors before running your code on the entire dataset.

## Recurrent Neural Networks

Convolutional Neural Networks work well for problems like image classification where the inputs and outputs are independent and of fixed size. However, many problems do not have these constraints. For example, what if we want to predict the next word in a sentence? This is clearly not independent since the output for one iteration becomes the input for the next iteration. *Recurrent Neural Networks* (RNNs) address these issues by using sequences as the input, output, or both, allowing for temporal dynamic behavior. They perform the same task for every element of the sequence, hence their recurrent nature. Each task uses the input as well as recent previous information, called memory, from the network to create the output. Even if the input is not sequential, it is possible to process it sequentially using RNNs, resulting in powerful learning algorithms.

## Data

For this lab, we will use Google Colab and its GPU capability. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a pop-up called *Notebook Settings*. Under *Hardware Settings*, select *GPU*. We recommend mounting a Google Drive to the notebook to make loading and saving data easier. This will save the data if the notebook is disconnected; if the data is saved to the Colab directory, the entire project must be rerun. To mount a Google Drive, run

```
>>> from google.colab import drive
>>> drive.mount("/content/drive")
```

Follow the instructions in the cell to authorize the account.

If you need to refresh your drive connection, you can run

```
>>> drive.mount("/content/drive", force_remount = True).
```

## Download Data

We will be using a collection of Mozart piano sonatas as the data to train on. For easy download, run the following function to save the files to `filepath` in the Google Drive folder.

```
def download_data(filepath):
    if not os.path.exists(os.path.join(filepath, "mozart.sonatas.tar.gz")):
        datasets.utils.download_url("https://github.com/Foundations-of-Applied-←
            Mathematics/Data/raw/master/Volume3/mozart.sonatas.tar.gz", filepath,←
            "mozart.sonatas.tar.gz", None)

    print("Extracting 'mozart.sonatas.tar.gz'")
    gzip_path = os.path.join(filepath, "mozart.sonatas.tar.gz")
    with open(gzip_path.replace(".gz", ""), "wb") as out_f, gzip.GzipFile(←
        gzip_path) as zip_f:
        out_f.write(zip_f.read())

    print(f"Untarring 'mozart.sonatas.tar'")
    tar_path = os.path.join(filepath, "mozart.sonatas.tar")
    z = tarfile.TarFile(tar_path)
    z.extractall(tar_path.replace(".tar", ""))

>>> download_data("drive/MyDrive/Colab")
Downloading https://raw.githubusercontent.com/Foundations-of-Applied-←
    Mathematics/Data/master/RNN/mozart.sonatas.tar.gz to drive/MyDrive/Colab/←
    mozart.sonatas.tar.gz

Extracting mozart.sonatas.tar.gz
Untarring mozart.sonatas.tar
```

## Parsing the Data

Music21 is a musical toolkit for Python developed by MIT.<sup>1</sup> It can read and write music files with the .mid extension, which are MIDI files, standing for Musical Instrument Digital Interface files. Midi files contain information on music, like which notes are played, how loud each note is, and for how long each note is held.

---

<sup>1</sup><https://web.mit.edu/music21/doc/index.html>.

There are two important object types: Notes and Chords. A Note object is comprised of three attributes. The `pitch` and `octave` give information about the frequency of the Note. There are seven pitches: A, B, C, D, E, F, and G. These pitches repeat, doubling the frequency of the vibration of the previous matching pitch. The interval over which the frequency of a note is doubled is called an octave. A piano has seven octaves, and the middle of the keyboard is called `middle c`. In Music21, it is represented by C4, where 4 is the octave. Lastly, the `offset` is the temporal location of the Note in the file. Chord objects contain multiple Note objects that are played at the same time.

```
from music21 import converter, instrument, note, chord, stream

# Read the file piano_sonata_279.mid
midi = converter.parse("piano_sonata_279.mid")
notes_to_parse = instrument.partitionByInstrument(midi).parts.stream().recurse()
()

# Display the Note and Chord objects, their pitches and offsets
for element in notes_to_parse:
    if isinstance(element, note.Note):
        print(element, element.pitch, element.offset)
    elif isinstance(element, chord.Chord):
        print(element, element.pitches, element.offset)

<music21.note.Note E> E5 803.0
<music21.note.Note F> F5 803.5
<music21.chord.Chord B3 B2> (<music21.pitch.Pitch B3>, <music21.pitch.Pitch B2>) 803.5
<music21.note.Note G> G5 804.0
<music21.note.Note F> F5 804.5
<music21.chord.Chord C4 C3> (<music21.pitch.Pitch C4>, <music21.pitch.Pitch C3>) 804.5
<music21.note.Note E-> E-5 805.0
<music21.note.Note D> D5 805.5
<music21.chord.Chord E-3 E-4> (<music21.pitch.Pitch E-3>, <music21.pitch.Pitch E-4>) 805.5
```

```
# Helper function to parse through a Chord object
def order_pitches(pitches):
    """ pitches: element.pitches object where element is a chord.Chord
        returns: sorted list of strings for each pitch in the chord
    """
    return sorted(list(set([str(n) for n in pitches])))
```

**Problem 1.** Download the data. Write a function that accepts the path to the .mid files, parses the files, and returns a list of the 114215 Notes and Chords as strings. There are many element types in MIDI files, so be sure to only look for Notes and Chords. For the Chords, join the pitches of the Notes in the Chords with a . as in ("D3.D2").

Print the length of your list and the number of unique Notes and Chords.

```
# Example of a part of the list
["A5", "C6", "G3.C4", "A5", "B-5", "A5", "G5", "D3.D2"]
```

Hint: An easy way to get the list of mozart sonata file names is with the following code.

```
import glob

>>> glob.glob(filepath + "/mozart_sonatas/mozart_sonatas/*.mid")
```

Also, you'll want to wrap `element.pitch` with `str()` to convert it into a string. Furthermore, the `.join()` method may be useful when constructing the Chord strings.

For the remainder of this lab, we will refer to the notes and chords in the list created in Problem 1 simply as pitches. In order for this data to be applied to an RNN, we need to create sequences. We do this by looping through the list of pitches and slicing it into lists of a given length. The label for each sequence, or the correct pitch we want the RNN to predict, is the element immediately following the sequence. So for a sequence length of 10, given elements 1 through 10 as a sequence, element 11 would be the label.

Since RNNs only accept numbers, we need to convert the pitches to integers. Using the sample list in Problem 1 as an example, we would map "A5" to 0, "C6" to 1, "G3.C4" to 2, "A5" to 0 again, and so on. The PyTorch DataLoader accepts a list of lists, where each element is of the form `[sequence, label]`, where the sequence is a PyTorch Long tensor, and the label is an integer. So in our case, the sequence will be a tensor of integers representing pitches while the label will be the integer representing the first pitch that follows the sequence.

```
# Example Data
example_data = [169, 269, 165, 187, 24, 366, 353, 269, 260, 233, 223, 169,
                162, 366, 353, 269, 260, 233, 223, 169, 162, 24, 8, 269, 260, 91]

# Create sequences as Long Tensors
first_sequence = torch.LongTensor(example_data[0:10])
second_sequence = torch.LongTensor(example_data[1:11])
first_label = example_data[10]
second_label = example_data[11]

# Example of data points formatted for the DataLoader, [sequence, label]
>>> [first_sequence, first_label]
[tensor([169, 269, 165, 187, 24, 366, 353, 269, 260, 233]), 223]

>>> [second_sequence, second_label]
[tensor([269, 165, 187, 24, 366, 353, 269, 260, 233]), 169]
```

**Problem 2.** Using the list returned in Problem 1, create the training, validation, and testing DataLoaders. Make sure to do all the following steps:

- Convert the pitches to integers.
- Split the data into Long tensors of length 10.
- Create the labels.
- Randomly split the data into training, validation, and test sets using a 70/15/15 split (use `torch.utils.data.random_split(data,lengths)` where `lengths=[0.7, 0.15, 0.15]`).
- Create the DataLoaders for these sets of data, using `batch_size=128` for the training data and `batch_size=32` for the validation and test data; also, set `shuffle=True` for the training data and `False` for the validation and test data (this is common practice in Deep Learning).

Print the length of each DataLoader (they should be 624, 536, and 536, respectively).

Hint: To keep all batches the same size, drop the last training batch in the DataLoader with the parameter `drop_last=True`.

## LSTM

While RNNs have the ability to look at short-term history, like the previous word in a sentence, they lack longer term contexts. For example, predicting the last word in the "The boat is in the *water*" is relatively easy. Consider the following two sentences separated by some other text: "I grew up in France ... I speak fluent *French*." It's clear that the last word will be a language, but we need the previous information of France to correctly identify which language. RNNs can't remember this information due to exploding and vanishing gradients.

*Long Short-Term Memory* (LSTM) networks are a popular RNN variation capable of long-term memory that solve this problem. They are used extensively in speech recognition, machine translation, and text-to-speech programs. Every step in the LSTM has three inputs: the current input, the short-term memory (hidden state) from the previous input, and the long-term memory (cell state). There are three gates that regulate these three types of memory. The *Input Gate* decides what information will be added to the long-term memory, the *Forget Gate* chooses which information should be kept in the long-term memory, and the *Output Gate* creates the new short-term memory.

## Defining the Network Layers

In PyTorch, the memory is a tuple (hidden state, cell state) and must be initialized before the LSTM layer is called. Usually, the hidden state initialization function is defined in the network class and is called during the training loop for each batch. The LSTM layer can be stacked, with the input from one layer going directly to the next layer; `num_layers` is how many stacked LSTM layers there are in the model. The `hidden_size` is the number of features in the hidden layer. This can be any size, but for this lab we will use 256.

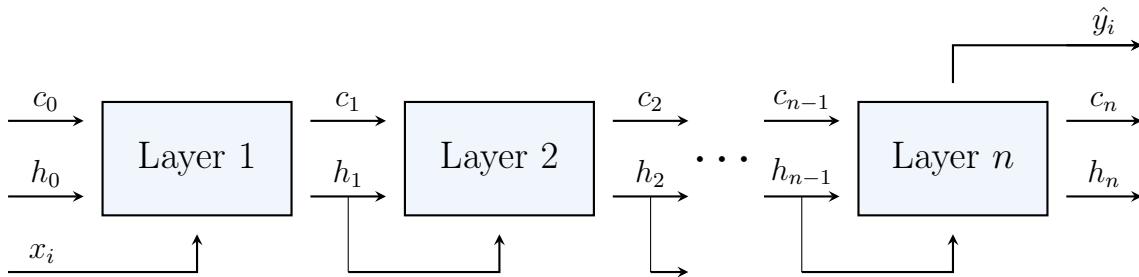


Figure 27.1: PyTorch implementation of an LSTM. The LSTM takes as input two initial memory states (hidden, cell) as well as the  $i$ th datapoint  $x_i$  from a batch, and outputs the updated memory states and predicted output  $\hat{y}_i$ . The input for each stacked layer is the hidden state from the previous layer, and  $n$  is equal to `num_layers`. Note that the PyTorch LSTM runs all datapoints in a batch in parallel to maximize efficiency.

```

class RNN(nn.Module):
    """ Recurrent Neural Network Class """

    def __init__(self):
        super(RNN, self).__init__()

        # Define function to initialize hidden states
        def init_hidden(self, batch_size):
            weight = next(self.parameters()).data
            h0 = weight.new(self.num_layers, batch_size, self.hidden_size).zero_().to(device)
            h1 = weight.new(self.num_layers, batch_size, self.hidden_size).zero_().to(device)
            return (h0, h1)
  
```

Before calling the LSTM layer, we will use an embedding layer to store the words. The embedding layer is a lookup table that takes in indices and outputs the word embeddings. This is PyTorch's method of one-hot encoding, a process in which variables are converted to binary for better predictions. The first parameter is the number of words in the dictionary; in our case, there are around 668 possible notes and chords. The second parameter is the embedding dimension. 32 and 64 are good choices for the embedding dimension.

The LSTM layer has 5 parameters. The first three have already been discussed. The parameter `batch_first` is a boolean that indicates if the batch size is the first or the second dimension in the input tensor. Since we are using the DataLoader, the batch size will be the first dimension and `batch_first=True`. If the last parameter, `dropout`, is defined, a Dropout layer is added after each LSTM layer, except the last. During a Dropout layer, elements of the input tensor are randomly zeroed out with probability  $p$ , and the output is scaled. This is sometimes used for regularization to improve the network. However, we will *NOT* add a Dropout layer to our model, because we will instead use a BatchNorm1d layer. BatchNorm1d layers normalize the input and have as parameters the number of features of the input. Thus, if we were to use both Dropout to BatchNorm1d layers, the input would be normalized over fewer nodes than the input actually contains, which would throw off the scaling of the model.

Ordinarily, the last layer would be a softmax activation function. Softmax rescales a tensor to  $[0, 1]$  with the sum of all elements equal to 1. Thus the output of Softmax can be thought of as a probability vector. However, the Cross Entropy Loss function which we use for this model already performs a Softmax for us. Doing the softmax twice is unnecessary and can be detrimental as it can distort learning. Notice that all of the layers: Embedding, LSTM, Linear, BatchNorm1d, and LogSoftmax are initialized in the `__init__()` function.

```
class RNN(nn.Module):
    """ Example class for LSTM model """

    def __init__(self, n_notes, embedding_dim):
        super(RNN, self).__init__()

        self.hidden_size = 256
        self.num_layers = 3      # number of layers in the LSTM
        self.n_notes = n_notes   # number of unique pitches
        self.embedding = nn.Embedding(n_notes, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, self.hidden_size,
                           self.num_layers, batch_first=True)
        self.batch1 = nn.BatchNorm1d(self.hidden_size)
        self.linear = nn.Linear(self.hidden_size, self.n_notes)

    def forward(self, x, hidden):
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)
        out = self.batch1(lstm_out[:, -1])
        # Output from final step is passed forward
        return self.linear(out), hidden
```

During training, when the model is called, the input is embedded and then passed to the LSTM layer with the hidden states. The hidden state output is saved for the next batch while the LSTM output from the final step is sent through the rest of the model. To prevent differentiating the hidden states, we must call the `detach()` method before taking a backwards step. This disables automatic differentiation on the hidden states during training. Because we don't do a backwards step during testing, we don't need to worry about detaching the hidden states during testing.

```
# Initialize the model
model = RNN()

# Hidden state training demonstration
for epoch in range(30):
    for x_truth, y_truth in train_loader:

        # Initialize the hidden states
        (h0, h1) = model.init_hidden(train_batch_size)

        # Pass data through the model to get output and new hidden states
        output, (h0, h1) = model(x_truth, (h0, h1))
```

```
# Disable automatic differentiation on the hidden states
h0 = h0.detach()
h1 = h1.detach()
```

## A Faster Way to Calculate Validation Accuracy

We would like to periodically check our model's validation accuracy as our model is training, but this takes time. One way to save time is to only calculate validation accuracy every  $n$  epochs. Another way is to increase the batch size of the validation DataLoader; this is the method we use in this lab, which is why we set the validation and test DataLoader batch sizes to 32 in Problem 2. The problem is, if we compare the predicted labels of a large batch to their true values at the same time, the probability that every data point is correct is small, so the validation accuracy will be mostly 0. The way to work around this is to compare the predicted labels of each batch with their true values separately, not together. An example of how this might be done is demonstrated in the following:

```
validation = 0
model.eval()
for x_truth, y_truth in validation_loader:
    x_truth, y_truth = x_truth.to(device), y_truth.to(device)
    (h0, h1) = model.init_hidden(val_batch_size)
    y_hat, _ = model(x_truth, (h0, h1))

    # sum how many elements equal each other between the true and predicted
    # batches, then divide by the number of elements in the batch
    validation += sum(torch.eq(y_truth, y_hat.argmax(1))) / val_batch_size

mean_validation_accuracy = validation.item() / len(validation_loader)
```

### ACHTUNG!

If you train your model using a GPU and see an error that begins like this:

```
RuntimeError: CUDA error: device-side assert triggered
```

it means that an error occurred while your code was being run on the GPU. Hardware complications make it more difficult to pass information about exceptions that occur on the GPU to the CPU. You can often get more debugging information about your specific error by using

```
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
```

although this will involve restarting your kernel.

You can also switch from the GPU to the CPU and likely get more descriptive error messages from the CPU.

**Problem 3.** Create an LSTM network class. Have a hidden layer size of 256, and include at least 3 LSTM layers. Also have at least 2 Linear layers. The last LSTM layer and each of the Linear layers except for the last Linear layer should be followed by a BatchNorm1d layer, for at least 2 total BatchNorm layers.

Initialize the model. Define the loss as CrossEntropyLoss, and define the optimizer as RMSprop.

```
optimizer = torch.optim.RMSprop(model.parameters(), lr=.001)
```

Train the model for 30 epochs. Make sure to reinitialize the hidden states ( $h_0$ ,  $h_1$ ) for each training batch. After taking a backwards step during training, scale the gradients using

```
nn.utils.clip_grad_norm_(model.parameters(), 5)
```

This will ensure that the gradients are reasonably sized so that the model can learn.

At the end of every epoch, calculate the validation accuracy and mean loss on the validation data. Remember to change the model to `eval()` mode when running the validation data and then `train()` when running on the training data. The hidden states ( $h_0$ ,  $h_1$ ) will also need to be reinitialized for each validation batch.

Once the training is complete, plot the training and validation losses versus epochs on the same plot. Also, plot the validation accuracy versus epochs. Then, print the final test accuracy by running the finished model on the test data.

Hint: While training this model for 30 epochs on a GPU should take less than 5 minutes, you may want to test your code by only training it for 2 epochs, and then when everything works the way it should, train it for the whole 30 epochs. After 2 epochs, your model should have a validation accuracy of 10 – 20%.

### ACHTUNG!

Colab has a 12 hour limit on the amount of GPU available, and the longer one runs, the less priority it has. If you follow the instructions given in this lab correctly, training should take less than 5 minutes. Nevertheless, you may still wish to save the training progress of your model's weights after each epoch. If you wish to do so, you may include this block of code in your for loop.

```
torch.save({
    "epoch": epoch_number,
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    "loss": loss}, filename)
```

Then, if at any point the notebook has disconnected, all you need to do is reinitialize the model, loss, and optimizer, and then run this function to load the saved model.

```
def load_model(filename):
    """ Load a saved model to continue training or evaluate """
    device = torch.device("cuda:0" if torch.cuda.is_available() ←
        else "cpu")

    # n_notes is the number of unique pitches
    model = RNN(n_notes, embedding_dim)
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.RMSprop(model.parameters(), lr=.001)

    checkpoint = torch.load(filename, map_location=torch.device("cpu←
        "))
    model.load_state_dict(checkpoint["model_state_dict"])
    optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
    last_epoch = checkpoint["epoch"]
    loss = checkpoint["loss"]
    model.eval() # Toggle evaluation mode

    return model, criterion, optimizer
```

## Generating Music

Now that we have trained the model, we can create our own piano sonata excerpt by predicting a new sequence of notes. We will start with an initial sequence of notes, predict what note should follow, and then shift the sequence to include this new note in order to predict the next one, and we'll repeat this process for as long as we like.

Specifically, first select a random sequence from the test data, and initialize an empty list of predictions. Then, for each note we wish to predict, initialize the hidden states using `model.init_hidden(batch_size)`, and then get a prediction by inputting the sequence and these hidden states into the model, just as we did in the training step. The argmax of this prediction (`prediction.argmax().item()`) is an integer representing a pitch. Append this integer to our list of predictions, and then update the sequence by appending this integer to it, and then by dropping the first entry of the sequence. Repeat this process for each note we wish to predict. The list of predictions (which is a list of integers) can then be converted into pitches using the same method we used to initially convert the pitches into integers.

**Problem 4.** Write a function that randomly chooses a sequence in the test data (which has length 10) and predicts the next  $n$  elements, defaulting to 500. Convert the predicted elements to pitches, and return them as a list of length  $n$ . It should look similar to

```
['D4', 'C#4', 'F#5', 'G5', 'A5', 'C6', 'G3.C4', 'B-5', 'A5', 'G5', 'A5']
```

Now we need to convert our list of pitches into Music21 Notes and Chords objects. For each element in the list of pitches, we first determine if it's a note or a chord by the presence of a . (period) in the string. Music21 Note objects are created using the pitch and instrument type. If the element is a chord, we can create a Muisc21 Chord object by first creating a list of Note objects for each note in the chord.

Music21 Note and Chord objects must also have a specified *offset*. The offset indicates at what timestep each object is to be played. The first object will have an offset of 0, and the offset will increment for each following object. The simplest way to choose each offset is look at the distribution of offsets in the original dataset and choose a set amount to increment the offset each time. Since the most common offset (0.0) results in notes being played at the same time, we'll ignore it and choose to increment the offset by either 0.25 or 0.5. For a more advanced option, you could randomly generate which offset to use based on a probability distribution that reflects the following:

0.0	1999
0.25	1167
0.5	507

Table 27.1: The three most common offset distance and their frequency in the Mozart data.

In summary, while looping through our predicted pitches, if we should come accross a Chord, the code to create a Music21 Chord object would look like the following:

```
notes = []

# Create Note objects for each note in the chord
for pitch in chord_pitches:
    new_note = note.Note(pitch)
    # Specify Piano as the instrument type
    new_note.storedInstrument = instrument.Piano()
    notes.append(new_note)
```

```
# Create a Chord object using list of Note objects
new_chord = chord.Chord(notes)

# Specify offset for this o
```

Finally, we write the list of Music21 objects to a midi file and save it.

```
midi_stream = stream.Stream(output_notes)
midi_stream.write("midi", fp=file_location)
```

You can embed and play the file in your notebook using the following code, which first converts the .midi file into a .wav file.

```
!apt install fluidsynth
!cp /usr/share/sounds/sf2/FluidR3_GM.sf2 ./font.sf2
!pip install midi2audio
from midi2audio import FluidSynth
from IPython.display import Audio

FluidSynth("font.sf2").midi_to_audio("file_location", "new_file_location.wav")
Audio("new_file_location.wav")
```

**Problem 5.** Convert the predictions from Problem 4 into Music21 Note and Chord objects and save it as "`mozart.mid`". Embed your music file into the notebook.

Part II  
Appendices





# NumPy Visual Guide

**Lab Objective:** NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to  $n$ -dimensional arrays.

## Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax  $[a:b]$  can be read as “the  $a$ th entry up to (but not including) the  $b$ th entry.” Similarly,  $[a:]$  means “the  $a$ th entry to the end” and  $[:b]$  means “everything up to (but not including) the  $b$ th entry.”

$$A[1] = A[1,:] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[:,2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[1:,:2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[1:-1,1:-1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

$$B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\text{np.hstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\text{np.vstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = [ \times \quad \times \quad \times \quad \times ]$$

$$y = [ * \quad * \quad * \quad * ]$$

$$\text{np.hstack}((x, y, x)) = [ \times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times ]$$

$$\text{np.vstack}((x, y, x)) = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix}$$

$$\text{np.column_stack}((x, y, x)) = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

## Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> for more in-depth examples and broadcasting rules.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad x = [10 \quad 20 \quad 30]$$

$$A + x = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ + \\ 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$A + x.reshape((1, -1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

## Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$A.sum(axis=0) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [4 \quad 8 \quad 12 \quad 16]$$

$$A.sum(axis=1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [10 \quad 10 \quad 10 \quad 10]$$



# B

# Matplotlib Syntax and Customization Guide

**Lab Objective:** *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at <https://matplotlib.org/>.*

## Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.
2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
<code>add_subplot()</code>	Add a single subplot to the current figure
<code>axes()</code>	Add an axes to the current figure
<code>clf()</code>	Clear the current figure
<code>figure()</code>	Create a new figure or grab an existing figure
<code>gca()</code>	Get the current axes
<code>gcf()</code>	Get the current figure
<code>subplot()</code>	Add a single subplot to the current figure
<code>subplots()</code>	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

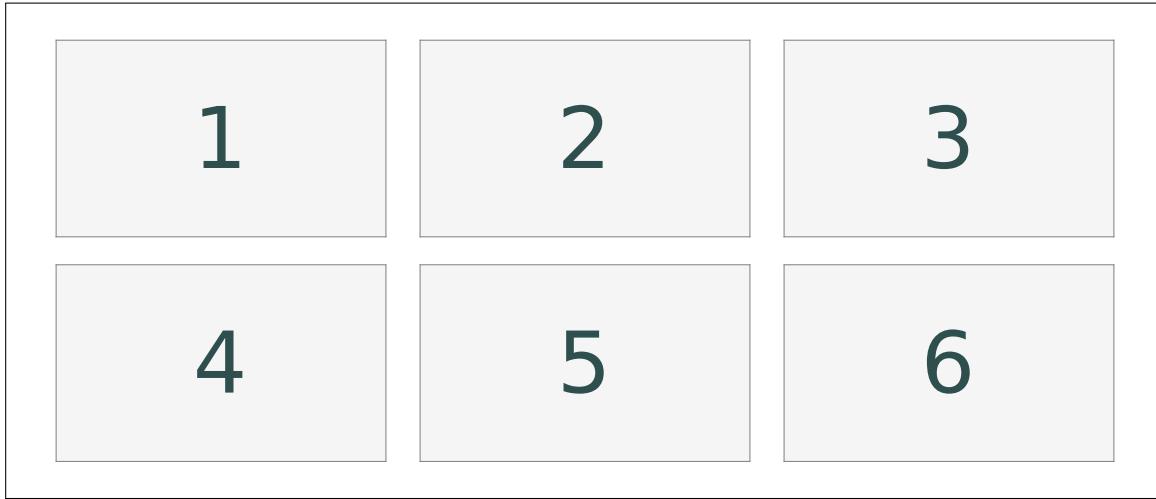


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

## ACHTUNG!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the *x*- and *y*-axis in the current axes, such as the *x* and *y* limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

# Plot Customization

## Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at [https://matplotlib.org/stable/gallery/style\\_sheets/style\\_sheets\\_reference.html](https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html).

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    #
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

## Plot layout

### Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the *x* and *y* ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin`, `xmax`, `ymin`, `ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the *x*- and *y*-scales equal (i.e. makes circles circular).

Function	Description
<code>axis()</code>	set the $x$ - and $y$ -limits of the plot
<code>grid()</code>	add gridlines
<code>xlim()</code>	set the limits of the $x$ -axis
<code>ylim()</code>	set the limits of the $y$ -axis
<code>xticks()</code>	set the location of the tick marks on the $x$ -axis
<code>yticks()</code>	set the location of the tick marks on the $y$ -axis
<code>xscale()</code>	set the scale type to use on the $x$ -axis
<code>yscale()</code>	set the scale type to use on the $y$ -axis
<code>ax.spines[side].set_position()</code>	set the location of the given spine
<code>ax.spines[side].set_color()</code>	set the color of the given spine
<code>ax.spines[side].set_visible()</code>	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an  $x$ - or  $y$ -coordinate of zero, respectively. The others are passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an  $x$ - or  $y$ -coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of  $\sin(x)$ . The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```

>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()

```

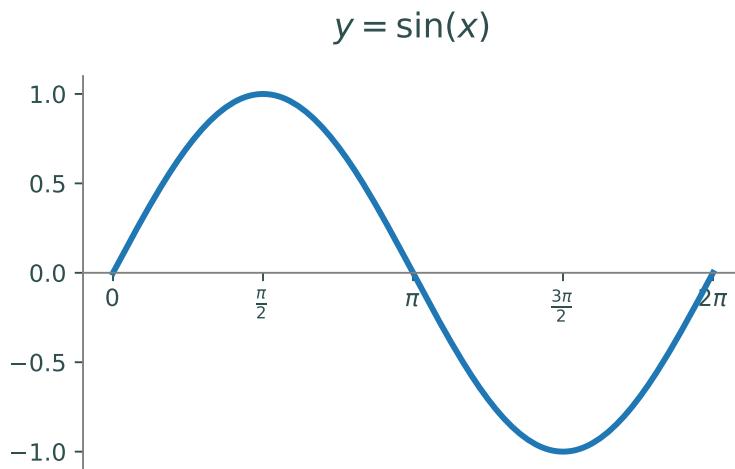


Figure B.2: Plot of  $y = \sin(x)$  with axes modified for clarity

### Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

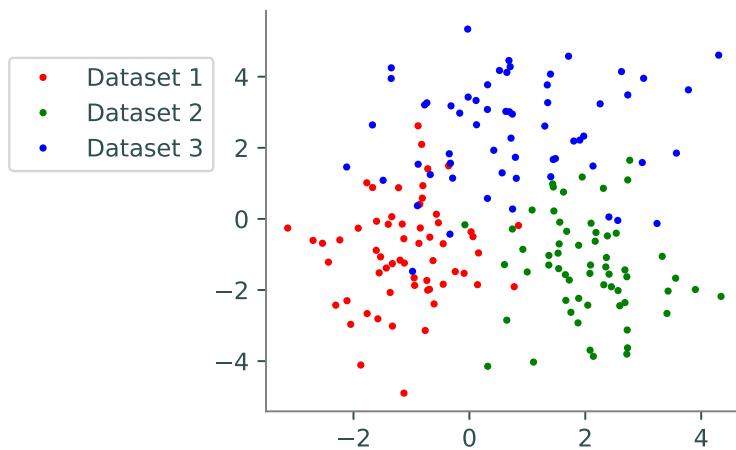


Figure B.3: Example of repositioning axes.

## Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specify colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"#0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at [https://matplotlib.org/stable/gallery/color/named\\_colors.html](https://matplotlib.org/stable/gallery/color/named_colors.html). If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color	Code	Color
'b'	blue	'y'	yellow
'g'	green	'k'	black
'r'	red	'w'	white
'c'	cyan	'CO' - 'C9'	Default colors
'm'	magenta		

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of 1.0 corresponds to fully opaque, and 0.0 corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

## Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at [https://matplotlib.org/stable/gallery/color/colormap\\_reference.html](https://matplotlib.org/stable/gallery/color/colormap_reference.html). Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
<code>annotate()</code>	adds a commentary at a given point on the plot	<code>annotate('text',(x,y))</code>
<code>arrow()</code>	draws an arrow from a given point on the plot	<code>arrow(x,y,dx,dy)</code>
<code>colorbar()</code>	Create a colorbar	<code>colorbar()</code>
<code>legend()</code>	Place a legend in the plot	<code>legend(loc='best')</code>
<code>text()</code>	Add text at a given position on the plot	<code>text(x,y,'text')</code>
<code>title()</code>	Add a title to the plot	<code>title('text')</code>
<code>suptitle()</code>	Add a title to the figure	<code>suptitle('text')</code>
<code>xlabel()</code>	Add a label to the $x$ -axis	<code>xlabel('text')</code>
<code>ylabel()</code>	Add a label to the $y$ -axis	<code>ylabel('text')</code>

Table B.4: Text and annotation functions in Matplotlib

## Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with L<sup>A</sup>T<sub>E</sub>X, a system for creating technical documents.<sup>1</sup> To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be  $\frac{1}{2} \sin(x^2)$ :

```
>>> plt.title(r"\frac{1}{2}\sin(x^2)")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of  $(x,y)$  can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point  $(0,0)$  corresponds to the bottom-left of the current subplot, and  $(1,1)$  corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'\sin(x)')
>>> plt.plot(x, np.cos(x), 'g', label=r'\cos(x)')
>>> plt.plot(x, -np.sin(x), 'b', label=r'-\sin(x)')
```

<sup>1</sup>See <http://www.latex-project.org/> for more information.

```
# Create the legend
>>> plt.legend()
```

## Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker		vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Matplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:  
>>> plt.plot(x, y, 'r.')  
  
#To change the size:  
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)  
>>> plt.scatter(x, y, marker='+', s=12)
```

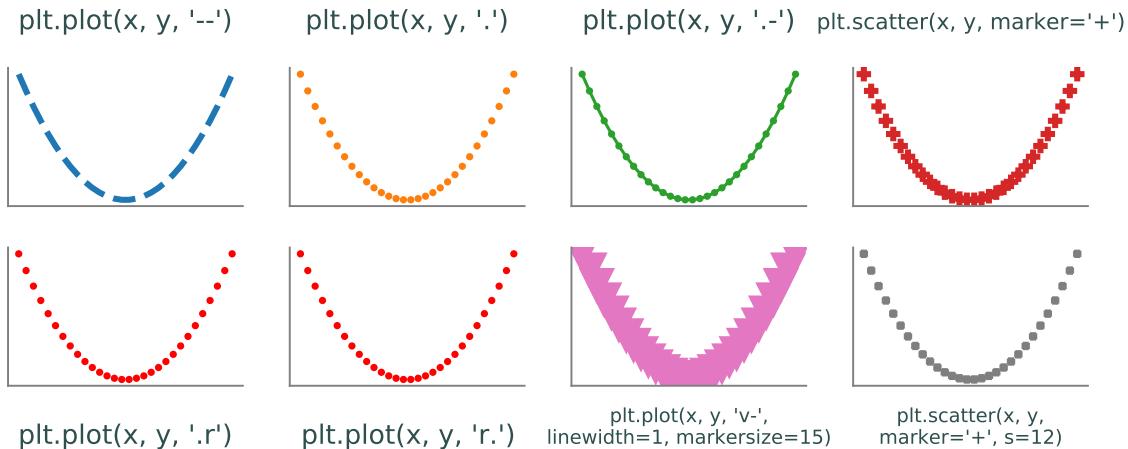


Figure B.4: Examples of setting line and marker styles.

## Plot Types

Matplotlib has functions for creating many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
<code>bar</code>	makes a bar graph	<code>bar(x,height)</code>
<code>barh</code>	makes a horizontal bar graph	<code>barh(y,width)</code>
<code>boxplots</code>	makes one or more boxplots	<code>boxplots(data)</code>
<code>contour</code>	makes a contour plot	<code>contour(X,Y,Z)</code>
<code>contourf</code>	makes a filled contour plot	<code>contourf(X,Y,Z)</code>
<code>imshow</code>	shows an image	<code>imshow(image)</code>
<code>fill</code>	plots lines with shading under the curve	<code>fill(x,y)</code>
<code>fill_between</code>	plots lines with shading between two given y values	<code>fill_between(x,y1, y2=0)</code>
<code>hexbin</code>	creates a hexbin plot	<code>hexbin(x,y)</code>
<code>hist</code>	plots a histogram from data	<code>hist(data)</code>
<code>pcolormesh</code>	makes a heatmap	<code>pcolormesh(X,Y,Z)</code>
<code>pie</code>	makes a pie chart	<code>pie(x)</code>
<code>plot</code>	plots lines and data on standard axes	<code>plot(x,y)</code>
<code>plot_surface</code>	plot a surface in 3-D space	<code>plot_surface(X,Y,Z)</code>
<code>polar</code>	plots lines and data on polar axes	<code>polar(theta,r)</code>
<code>loglog</code>	plots lines and data on logarithmic x and y axes	<code>loglog(x,y)</code>
<code>scatter</code>	plots data in a scatterplot	<code>scatter(x,y)</code>
<code>semilogx</code>	plots lines and data with a log scaled x axis	<code>semilogx(x,y)</code>
<code>semilogy</code>	plots lines and data with a log scaled y axis	<code>semilogy(x,y)</code>
<code>specgram</code>	makes a spectrogram from data	<code>specgram(x)</code>
<code>spy</code>	plots the sparsity pattern of a 2D array	<code>spy(Z)</code>
<code>triplot</code>	plots triangulation between given points	<code>triplot(x,y)</code>

Table B.6: Some basic plotting functions in Matplotlib.

## Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

## Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

## Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

## Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

## Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of  $z = (x^2 + y) \sin(y)$ . The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
```

```

>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()

```

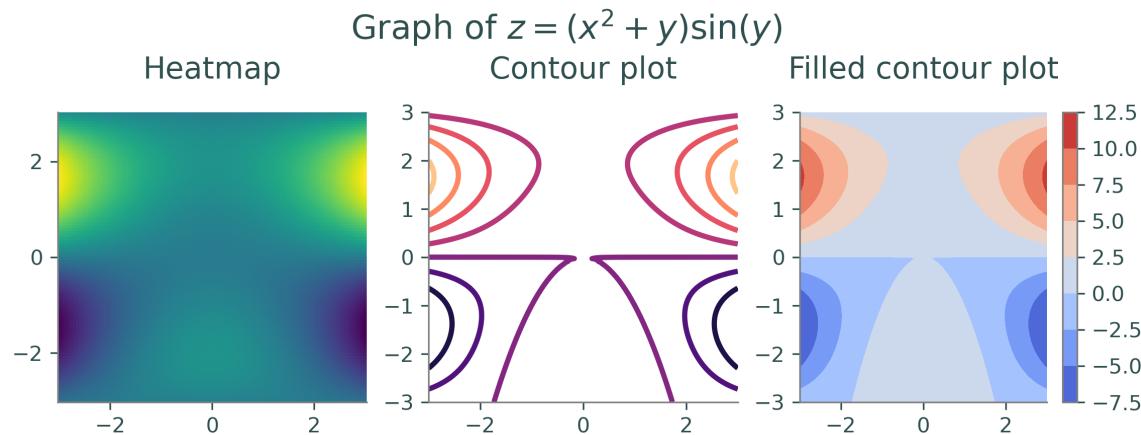


Figure B.5: Example of heatmaps and contour plots.

## Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D  $n \times m$  array for a grayscale image, or a 3-D  $n \times m \times 3$  array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range [0, 1]. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

## 3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

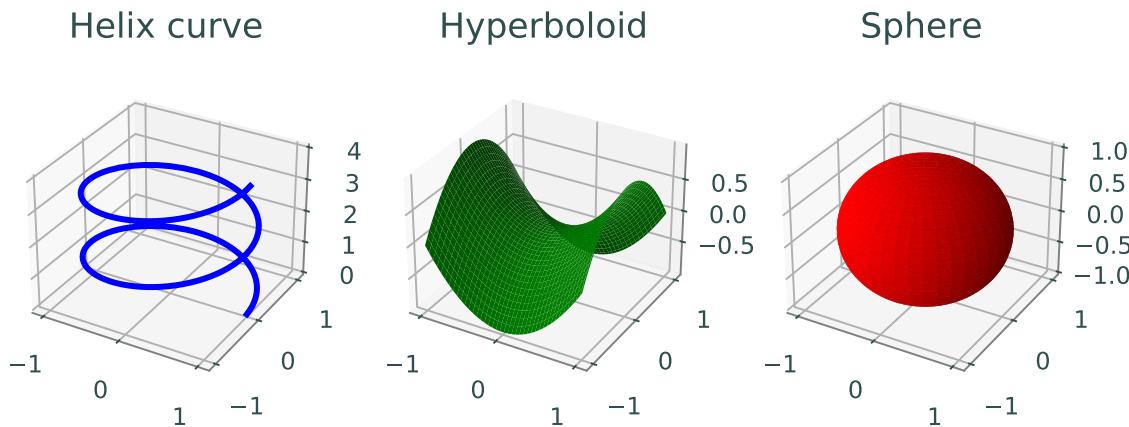


Figure B.6: Examples of 3-D plotting.

## Additional Resources

### rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the "`figure.dpi`" parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at [https://matplotlib.org/stable/api/matplotlib\\_configuration\\_api.html#matplotlib.RcParams](https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams).

### Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

### Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at <https://matplotlib.org/stable/tutorials/index.html>. It also has a large gallery of examples, found at <https://matplotlib.org/stable/gallery/index.html>. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.



# Bibliography

- [ADH<sup>+</sup>01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.
- [GKB17] Ian J. Goodfellow, Alexey Kurakin, and Samy Bengio. Adversarial examples in the physical world. 2017.
- [GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. 2015.
- [Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
- [Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.
- [Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.
- [VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.