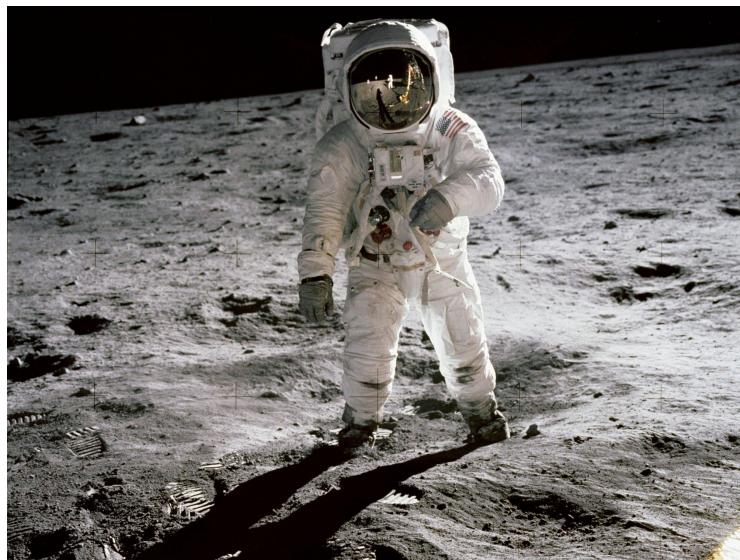


Labs for Foundations of Applied Mathematics

Volume III
Modeling with Uncertainty and Data

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

E. Evans

Brigham Young University

R. Evans

Brigham Young University

J. Grout

Drake University

J. Humpherys

Brigham Young University

T. Jarvis

Brigham Young University

J. Whitehead

Brigham Young University

J. Adams

Brigham Young University

J. Bejarano

Brigham Young University

Z. Boyd

Brigham Young University

M. Brown

Brigham Young University

A. Carr

Brigham Young University

T. Christensen

Brigham Young University

M. Cook

Brigham Young University

R. Dorff

Brigham Young University

B. Ehlert

Brigham Young University

M. Fabiano

Brigham Young University

A. Frandsen

Brigham Young University

K. Finlinson

Brigham Young University

J. Fisher

Brigham Young University

R. Fuhriman

Brigham Young University

S. Giddens

Brigham Young University

C. Gigena

Brigham Young University

M. Graham

Brigham Young University

F. Glines

Brigham Young University

M. Goodwin

Brigham Young University

R. Grout

Brigham Young University

D. Grundvig

Brigham Young University

J. Hendricks

Brigham Young University

A. Henriksen

Brigham Young University

I. Henriksen

Brigham Young University

C. Hettinger

Brigham Young University

S. Horst

Brigham Young University

K. Jacobson

Brigham Young University

J. Leete

Brigham Young University

J. Lytle	C. Robertson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. McMurray	M. Russell
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. McQuarrie	R. Sandberg
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Miller	M. Stauffer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Morrise	J. Stewart
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Morrise	S. Suggs
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Morrow	A. Tate
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Murray	T. Thompson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Nelson	M. Victors
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Parkinson	J. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Probst	R. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Proudfoot	J. West
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Reber	A. Zaitzeff
<i>Brigham Young University</i>	<i>Brigham Young University</i>

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Humpherys and Jarvis.

©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	iii
I Data Science Technologies	1
1 Introduction to SQL	3
2 Advanced SQL	15
3 Introduction to the Unix Shell	23
4 Unix Shell 2	35
5 Regular Expressions	49
6 Web Technologies	63
7 MongoDB	73
8 Introduction to Beautiful Soup	83
9 Advanced Web Scraping Techniques	93
10 Pandas I: Introduction	101
11 Pandas 2: Plotting	115
12 Pandas III: Grouping	127
13 Pandas IV: Time Series	139
14 Intro to Parallel Computing	151
15 Parallel Programming with MPI	167
II Machine Learning Algorithms	175
16 Kalman Filter	177

17	ARMA Models	187
18	Discrete Hidden Markov Models	195
19	Gaussian Mixture Models	207
20	Speech Recognition using CDHMMs	213
21	Gibbs Sampling and LDA	219
22	Metropolis Algorithm	227
23	PCA and LSI	237
24	Naive Bayes	249
25	Logistic Regression	257
26	Classification Trees	263
27	Random Forests	267
28	K-Nearest Neighbors and Support Vector Machines	269
29	Crime Mapping	273
30	Image Recognition Tasks	279
31	K-Means Clustering	283
32	Bayesian Search	293

Part I

Data Science Technologies

1

SQL I: Introduction

Lab Objective: *Being able to store and manipulate large data sets quickly is a fundamental part of data science. The SQL language is the classic database management system for working with tabular data. In this lab we introduce the basics of SQL, including creating, reading, updating, and deleting SQL tables, all via Python's standard SQL interaction modules.*

Relational Databases

A *relational database* is a collection of tables called *relations*. A single row in a table, called a *tuple*, corresponds to an individual instance of data. The columns, called *attributes* or *features*, are data values of a particular category. The collection of column headings is called the *schema* of the table, which describes the kind of information stored in each entry of the tuples.

For example, suppose a database contains demographic information for M individuals. If a table had the schema `(Name, Gender, Age)`, then each row of the table would be a 3-tuple corresponding to a single individual, such as `(Jane Doe, F, 20)` or `(Samuel Clemens, M, 74.4)`. The table would therefore be $M \times 3$ in shape. Another table with the schema `(Name, Income)` would be $M \times 2$ if it included all M individuals.

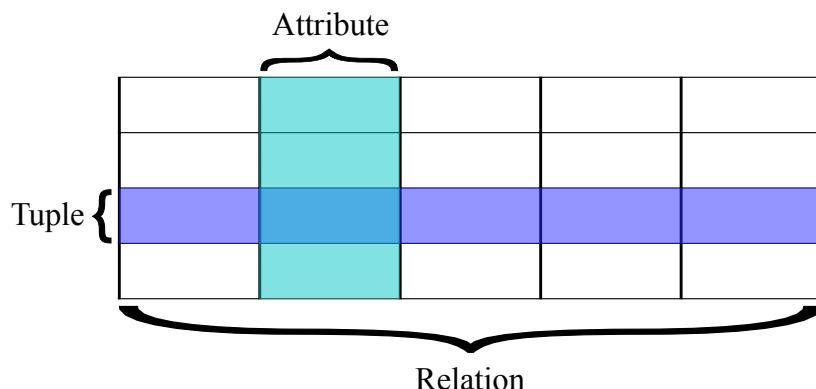


Figure 1.1: See https://en.wikipedia.org/wiki/Relational_database.

SQLite

The most common database management systems (DBMS) for relational databases are based on *Structured Query Language*, commonly called *SQL* (pronounced¹ “sequel”). Though SQL is a language in and of itself, most programming languages have tools for executing SQL routines. In Python, the most common variant of SQL is *SQLite*, implemented as the `sqlite3` module in the standard library.

A SQL database is stored in an external file, usually marked with the file extension `db` or `mdf`. These files should **not** be opened in Python with `open()` like text files; instead, any interactions with the database—creating, reading, updating, or deleting data—should occur as follows.

1. Create a connection to the database with `sqlite3.connect()`. This creates a database file if one does not already exist.
2. Get a *cursor*, an object that manages the actual traversal of the database, with the connection’s `cursor()` method.
3. Alter or read data with the cursor’s `execute()` method, which accepts an actual SQL command as a string.
4. Save any changes with the cursor’s `commit()` method, or revert changes with `rollback()`.
5. Close the connection.

```
>>> import sqlite3 as sql

# Establish a connection to a database file or create one if it doesn't exist.
>>> conn = sql.connect("my_database.db")
>>> try:
...     cur = conn.cursor()                      # Get a cursor object.
...     cur.execute("SELECT * FROM MyTable")      # Execute a SQL command.
... except sql.Error:                          # If there is an error,
...     conn.rollback()                         # revert the changes
...     raise                                 # and raise the error.
... else:                                    # If there are no errors,
...     conn.commit()                           # save the changes.
... finally:                                # Close the connection.
...     conn.close()
```

ACHTUNG!

Some changes, such as creating and deleting tables, are automatically committed to the database as part of the cursor’s `execute()` method. Be **extremely cautious** when deleting tables, as the action is immediate and permanent. Most changes, however, do not take effect in the database file until the connection’s `commit()` method is called. Be careful not to close the connection before committing desired changes, or those changes will not be recorded.

¹See <https://english.stackexchange.com/questions/7231/how-is-sql-pronounced> for a brief history of the somewhat controversial pronunciation of SQL.

The `with` statement can be used with `open()` so that file streams are automatically closed, even in the event of an error. Likewise, combining the `with` statement with `sql.connect()` automatically rolls back changes if there is an error and commits them otherwise. However, the actual database connection is **not** closed automatically. With this strategy, the previous code block can be reduced to the following.

```
>>> try:
...     with sql.connect("my_database.db") as conn:
...         cur = conn.cursor()                      # Get the cursor.
...         cur.execute("SELECT * FROM MyTable")      # Execute a SQL command.
...     finally:                                    # Commit or revert, then
...         conn.close()                           # close the connection.
```

Managing Database Tables

SQLite uses five native data types (relatively few compared to other SQL systems) that correspond neatly to native Python data types.

Python Type	SQLite Type
<code>None</code>	<code>NULL</code>
<code>int</code>	<code>INTEGER</code>
<code>float</code>	<code>REAL</code>
<code>str</code>	<code>TEXT</code>
<code>bytes</code>	<code>BLOB</code>

The `CREATE TABLE` command, together with a table name and a schema, adds a new table to a database. The schema is a comma-separated list where each entry specifies the column name, the column data type,² and other optional parameters. For example, the following code adds a table called `MyTable` with the schema (`Name`, `ID`, `Age`) with appropriate data types.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("CREATE TABLE MyTable (Name TEXT, ID INTEGER, Age REAL)")
...
>>> conn.close()
```

The `DROP TABLE` command deletes a table. However, using `CREATE TABLE` to try to create a table that already exists or using `DROP TABLE` to remove a nonexistent table raises an error. Use `DROP TABLE IF EXISTS` to remove a table without raising an error if the table doesn't exist. See Table 1.1 for more table management commands.

²Though SQLite does not force the data in a single column to be of the same type, most other SQL systems enforce uniform column types, so it is good practice to specify data types in the schema.

Operation	SQLite Command
Create a new table	<code>CREATE TABLE <table> (<schema>);</code>
Delete a table	<code>DROP TABLE <table>;</code>
Delete a table if it exists	<code>DROP TABLE IF EXISTS <table>;</code>
Add a new column to a table	<code>ALTER TABLE <table> ADD <column> <dtype></code>
Remove an existing column	<code>ALTER TABLE <table> DROP COLUMN <column>;</code>
Rename an existing column	<code>ALTER TABLE <table> ALTER COLUMN <column> <dtype>;</code>

Table 1.1: SQLite commands for managing tables and columns.

NOTE

SQL commands like `CREATE TABLE` are often written in all caps to distinguish them from other parts of the query, like the table name. This is only a matter of style: SQLite, along with most other versions of SQL, is case insensitive. In Python's SQLite interface, the trailing semicolon is also unnecessary. However, most other database systems require it, so it's good practice to include the semicolon in Python.

Problem 1. Write a function that accepts the name of a database file. Connect to the database (and create it if it doesn't exist). Drop the tables `MajorInfo`, `CourseInfo`, `StudentInfo`, and `StudentGrades` from the database if they exist. Next, add the following tables to the database with the specified column names and types.

- `MajorInfo`: `MajorID` (integers) and `MajorName` (strings).
- `CourseInfo`: `CourseID` (integers) and `CourseName` (strings).
- `StudentInfo`: `StudentID` (integers), `StudentName` (strings), and `MajorID` (integers).
- `StudentGrades`: `StudentID` (integers), `CourseID` (integers), and `Grade` (strings).

Remember to commit and close the database. You should be able to execute your function more than once with the same input without raising an error.

To check the database, use the following commands to get the column names of a specified table. Assume here that the database file is called `students.db`.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM StudentInfo;")
...     print([d[0] for d in cur.description])
...
['StudentID', 'StudentName', 'MajorID']
```

Inserting, Removing, and Altering Data

Tuples are added to SQLite database tables with the `INSERT INTO` command.

```
# Add the tuple (Samuel Clemens, 1910421, 74.4) to MyTable in my_database.db.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

With this syntax, SQLite assumes that values match sequentially with the schema of the table. The schema of the table can also be written explicitly for clarity.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable(Name, ID, Age) "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

ACHTUNG!

Never use Python's string operations to construct a SQL query from variables. Doing so makes the program susceptible to a *SQL injection attack*.^a Instead, use parameter substitution to construct dynamic commands: use a ? character within the command, then provide the sequence of values as a second argument to `execute()`.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     values = ('Samuel Clemens', 1910421, 74.4)
...     # Don't piece the command together with string operations!
...     # cur.execute("INSERT INTO MyTable VALUES " + str(values)) # BAD!
...     # Instead, use parameter substitution.
...     cur.execute("INSERT INTO MyTable VALUES(?, ?, ?);", values) # Good.
```

^aSee <https://xkcd.com/327/> for an example.

To insert several rows at a time to the same table, use the cursor object's `executemany()` method and parameter substitution with a list of tuples. This is typically much faster than using `execute()` repeatedly.

```
# Insert (Samuel Clemens, 1910421, 74.4) and (Jane Doe, 123, 20) to MyTable.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     rows = [('John Smith', 456, 40.5), ('Jane Doe', 123, 20)]
...     cur.executemany("INSERT INTO MyTable VALUES(?, ?, ?);", rows)
```

Problem 2. Expand your function from Problem 1 so that it populates the tables with the data given in Tables 2.1a–2.1d.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

(d) StudentGrades

Table 1.2: Student database.

The `StudentInfo` and `StudentGrades` tables are also recorded in `student_info.csv` and `student_grades.csv`, respectively, with `NULL` values represented as `-1`. A CSV (comma-separated values) file can be read like a normal text file or with the `csv` module.

```
>>> import csv
>>> with open("student_info.csv", 'r') as infile:
...     rows = list(csv.reader(infile))
```

To validate your database, use the following command to retrieve the rows from a table.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     for row in cur.execute("SELECT * FROM MajorInfo;"):
...         print(row)
(1, 'Math')
(2, 'Science')
(3, 'Writing')
(4, 'Art')
```

Problem 3. The data file `us_earthquakes.csv`^a contains data from about 3,500 earthquakes in the United States since the 1769. Each row records the year, month, day, hour, minute, second, latitude, longitude, and magnitude of a single earthquake (in that order). Note that latitude, longitude, and magnitude are floats, while the remaining columns are integers.

Write a function that accepts the name of a database file. Drop the table `USEarthquakes` if it already exists, then create a new `USEarthquakes` table with schema (`Year`, `Month`, `Day`, `Hour`, `Minute`, `Second`, `Latitude`, `Longitude`, `Magnitude`). Populate the table with the data from `us_earthquakes.csv`. Remember to commit the changes and close the connection. (Hint: using `executemany()` is much faster than using `execute()` in a loop.)

^aRetrieved from <https://datarepository.wolframcloud.com/resources/Sample-Data-US-Earthquakes>.

The WHERE Clause

Deleting or altering existing data in a database requires some searching for the desired row or rows. The `WHERE` clause is a *predicate* that filters the rows based on a boolean condition. The operators `==`, `!=`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, and `NOT` all work as expected to create search conditions.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     # Delete any rows where the Age column has a value less than 30.
...     cur.execute("DELETE FROM MyTable WHERE Age < 30;")
...     # Change the Name of "Samuel Clemens" to "Mark Twain".
...     cur.execute("UPDATE MyTable SET Name='Mark Twain' WHERE ID==1910421;")
```

If the `WHERE` clause were omitted from either of the previous commands, every record in `MyTable` would be affected. **Always** use a very specific `WHERE` clause when removing or updating data.

Operation	SQLite Command
Add a new row to a table	<code>INSERT INTO table VALUES(<values>);</code>
Remove rows from a table	<code>DELETE FROM <table> WHERE <condition>;</code>
Change values in existing rows	<code>UPDATE <table> SET <column1>=<value1>, ... WHERE <condition>;</code>

Table 1.3: SQLite commands for inserting, removing, and updating rows.

Problem 4. Modify your function from Problems 1 and 2 so that in the `StudentInfo` table, values of `-1` in the `MajorID` column are replaced with `NULL` values.

Also modify your function from Problem 3 in the following ways.

1. Remove rows from `USEarthquakes` that have a value of 0 for the `Magnitude`.
2. Replace 0 values in the `Day`, `Hour`, `Minute`, and `Second` columns with `NULL` values.

Reading and Analyzing Data

Constructing and managing databases is fundamental, but most time in SQL is spent analyzing existing data. A *query* is a SQL command that reads all or part of a database without actually modifying the data. Queries start with the `SELECT` command, followed by column and table names and additional (optional) conditions. The results of a query, called the *result set*, are accessed through the cursor object. After calling `execute()` with a SQL query, use `fetchall()` or another cursor method from Table 1.4 to get the list of matching tuples.

Method	Description
<code>execute()</code>	Execute a single SQL command
<code>executemany()</code>	Execute a single SQL command over different values
<code>executescript()</code>	Execute a SQL script (multiple SQL commands)
<code>fetchone()</code>	Return a single tuple from the result set
<code>fetchmany(n)</code>	Return the next n rows from the result set as a list of tuples
<code>fetchall()</code>	Return the entire result set as a list of tuples

Table 1.4: Methods of database cursor objects.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get tuples of the form (StudentID, StudentName) from the StudentInfo table.
>>> cur.execute("SELECT StudentID, StudentName FROM StudentInfo;")
>>> cur.fetchone()          # List the first match (a tuple).
(401767594, 'Michelle Fernandez')

>>> cur.fetchmany(3)       # List the next three matches (a list of tuples).
[(678665086, 'Gilbert Chapman'),
 (553725811, 'Roberta Cook'),
 (886308195, 'Rene Cross')]

>>> cur.fetchall()         # List the remaining matches.
[(103066521, 'Cameron Kim'),
 (821568627, 'Mercedes Hall'),
 (206208438, 'Kristopher Tran'),
 (341324754, 'Cassandra Holland'),
 (262019426, 'Alfonso Phelps'),
 (622665098, 'Sammy Burke')]

# Use * in place of column names to get all of the columns.
>>> cur.execute("SELECT * FROM MajorInfo;").fetchall()
[(1, 'Math'), (2, 'Science'), (3, 'Writing'), (4, 'Art')]

>>> conn.close()
```

The `WHERE` predicate can also refine a `SELECT` command. If the condition depends on a column in a different table from the data that is being selected, create a *table alias* with the `AS` command to specify columns in the form `table.column`.

```

>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the names of all math majors.
>>> cur.execute("SELECT SI.StudentName "
...             "FROM StudentInfo AS SI, MajorInfo AS MI "
...             "WHERE SI.MajorID == MI.MajorID AND MI.MajorName == 'Math'")
# The result set is a list of 1-tuples; extract the entry from each tuple.
>>> [t[0] for t in cur.fetchall()]
['Cassandra Holland', 'Michelle Fernandez']

# Get the names and grades of everyone in English class.
>>> cur.execute("SELECT SI.StudentName, SG.Grade "
...             "FROM StudentInfo AS SI, StudentGrades AS SG "
...             "WHERE SI.StudentID == SG.StudentID AND CourseID == 2;")
>>> cur.fetchall()
[('Roberta Cook', 'C'),
 ('Cameron Kim', 'C'),
 ('Mercedes Hall', 'A+'),
 ('Kristopher Tran', 'A'),
 ('Cassandra Holland', 'D-'),
 ('Alfonso Phelps', 'B'),
 ('Sammy Burke', 'A-')]

>>> conn.close()

```

Problem 5. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problems 1 and 2, query the database for all tuples of the form `(StudentName, CourseName)` where that student has an “A” or “A+” grade in that course. Return the list of tuples.

Aggregate Functions

A result set can be analyzed in Python using tools like NumPy, but SQL itself provides a few tools for computing a few very basic statistics: `AVG()`, `MIN()`, `MAX()`, `SUM()`, and `COUNT()` are *aggregate functions* that compress the columns of a result set into the desired quantity.

```

>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the number of students and the lowest ID number in StudentInfo.
>>> cur.execute("SELECT COUNT(StudentName), MIN(StudentID) FROM StudentInfo;")
>>> cur.fetchall()
[(10, 103066521)]

```

Problem 6. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problem 3, query the `USEarthquakes` table for the following information.

- The magnitudes of the earthquakes during the 19th century (1800–1899).
- The magnitudes of the earthquakes during the 20th century (1900–1999).
- The average magnitude of all earthquakes in the database.

Create a single figure with two subplots: a histogram of the magnitudes of the earthquakes in the 19th century, and a histogram of the magnitudes of the earthquakes in the 20th century. Show the figure, then return the average magnitude of all of the earthquakes in the database. Be sure to return an actual number, not a list or a tuple.

(Hint: use `np.ravel()` to convert a result set of 1-tuples to a 1-D array.)

NOTE

Problem 6 raises an interesting question: are the number of earthquakes in the United States increasing with time, and if so, how drastically? A closer look shows that only 3 earthquakes were recorded (in this data set) from 1700–1799, 208 from 1800–1899, and a whopping 3049 from 1900–1999. Is the increase in earthquakes due to there actually being more earthquakes, or to the improvement of earthquake detection technology? The best answer without conducting additional research is “probably both.” Be careful to question the nature of your data—how it was gathered, what it may be lacking, what biases or lurking variables might be present—before jumping to strong conclusions.

See the following for more info on the `sqlite3` and SQL in general.

- <https://docs.python.org/3/library/sqlite3.html>
- <https://www.w3schools.com/sql/>
- https://en.wikipedia.org/wiki/SQL_injection

Additional Material

Shortcuts for WHERE Conditions

Complicated **WHERE** conditions can be simplified with the following commands.

- **IN**: check for equality to one of several values quickly, similar to Python's `in` operator. In other words, the following SQL commands are equivalent.

```
SELECT * FROM StudentInfo WHERE MajorID == 1 OR MajorID == 2;  
SELECT * FROM StudentInfo WHERE MajorID IN (1,2);
```

- **BETWEEN**: check two (inclusive) inequalities quickly. The following are equivalent.

```
SELECT * FROM MyTable WHERE AGE >= 20 AND AGE <= 60;  
SELECT * FROM MyTable WHERE AGE BETWEEN 20 AND 60;
```


2

SQL II (The Sequel)

Lab Objective: Since SQL databases contain multiple tables, retrieving information about the data can be complicated. In this lab we discuss joins, grouping, and other advanced SQL query concepts to facilitate rapid data retrieval.

We will use the following database as an example throughout this lab, found in `students.db`.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

(d) StudentGrades

Table 2.1: Student database.

Joining Tables

A *join* combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
SELECT <alias.column, ...
    FROM <table> AS <alias> JOIN <table> AS <alias>, ...
    ON <alias.column> == <alias.column>, ...
    WHERE <condition>
```

The `ON` clause tells the query how to join tables together. Typically if there are N tables being joined together, there should be $N - 1$ conditions in the `ON` clause.

Inner Joins

An *inner join* creates a temporary table with the rows that have exact matches on the attribute(s) specified in the `ON` clause. Inner joins **intersect** two or more tables, as in Figure 2.1a.

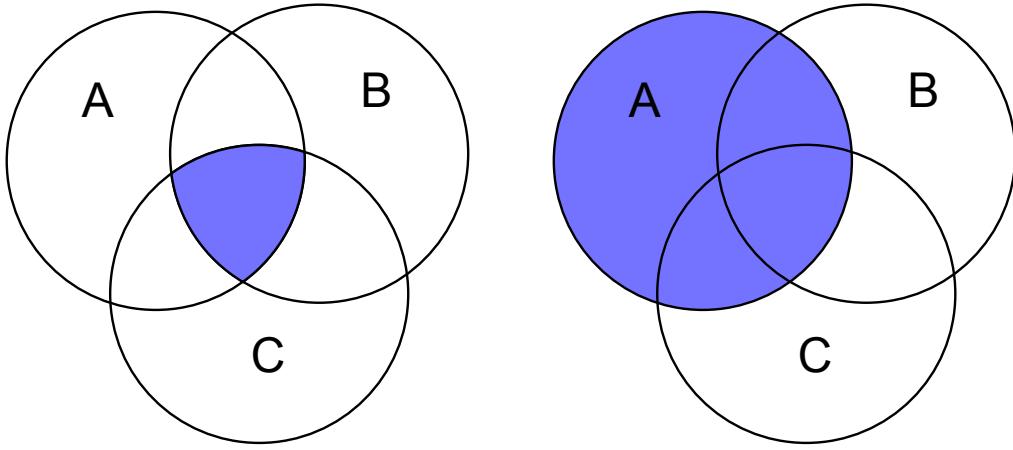


Figure 2.1

For example, Table 2.1c (`StudentInfo`) and Table 2.1a (`MajorInfo`) both have a `MajorID` column, so the tables can be joined by pairing rows that have the same `MajorID`. Such a join temporarily creates the following table.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
622665098	Sammy Burke	2	2	Science

Table 2.2: An inner join of `StudentInfo` and `MajorInfo` on `MajorID`.

Notice that this table is missing the rows where `MajorID` was `NULL` in the `StudentInfo` table. This is because there was no match for `NULL` in the `MajorID` column of the `MajorInfo` table, so the inner join throws those rows away.

Because joins deal with multiple tables at once, it is important to assign table aliases with the `AS` command. Join statements can also be supplemented with `WHERE` clauses like regular queries.

```
>>> import sqlite3 as sql
>>> cur = sql.connect("district.db")
>>> cur = conn.cursor()

>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID = MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]

# Select the names and ID numbers of the math majors.
>>> cur.execute("SELECT SI.StudentName, SI.StudentID "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID = MI.MajorID "
...             "WHERE MI.MajorName == 'Math';").fetchall()
[('Cassandra Holland', 341324754), ('Michelle Fernandez', 401767594)]
```

Problem 1. Write a function that accepts the name of a database file. Assuming the database to be in the format of Tables 2.1a–2.1d, query the database for the list of the names of students who have a B grade in any course (not a B– or a B+).

Outer Joins

A *left outer join*, sometimes called a *left join*, creates a temporary table with **all** of the rows from the first (left-most) table, and all the “matched” rows on the given attribute(s) from the other relations. Rows from the left table that don’t match up with the columns from the other tables are supplemented with `NULL` values to fill extra columns. Compare the following table and code to Table 2.2.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
678665086	Gilbert Chapman	NULL	NULL	NULL
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
821568627	Mercedes Hall	NULL	NULL	NULL
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
262019426	Alfonso Phelps	NULL	NULL	NULL
622665098	Sammy Burke	2	2	Science

Table 2.3: A left outer join of `StudentInfo` and `MajorInfo` on `MajorID`.

```
>>> cur.execute("SELECT * "
...                 "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...                 "ON SI.MajorID = MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
(678665086, 'Gilbert Chapman', None, None, None),
(553725811, 'Roberta Cook', 2, 2, 'Science'),
(886308195, 'Rene Cross', 3, 3, 'Writing'),
(103066521, 'Cameron Kim', 4, 4, 'Art'),
(821568627, 'Mercedes Hall', None, None, None),
(206208438, 'Kristopher Tran', 2, 2, 'Science'),
(341324754, 'Cassandra Holland', 1, 1, 'Math'),
(262019426, 'Alfonso Phelps', None, None, None),
(622665098, 'Sammy Burke', 2, 2, 'Science')]
```

Some flavors of SQL also support the `RIGHT OUTER JOIN` command, but `sqlite3` does not recognize the command since T1 `RIGHT OUTER JOIN` T2 is equivalent to T2 `LEFT OUTER JOIN` T1.

Joining Multiple Tables

Complicated queries often join several different relations. If the same kind join is being used, the relations and conditional statements can be put in list form. For example, the following code selects courses that Kristopher Tran has taken, and the grades that he got in those courses, by joining three tables together. Note that 2 conditions are required in the `ON` clause in this case.

```
>>> cur.execute("SELECT CI.CourseName, SG.Grade "
...                 "FROM StudentInfo AS SI" # Join 3 tables.
...                 "INNER JOIN CourseInfo AS CI, StudentGrades SG "
...                 "ON SI.StudentID==SG.StudentID AND CI.CourseID==SG.CourseID "
...                 "WHERE SI.StudentName == 'Kristopher Tran';").fetchall()
[('Calculus', 'C+'), ('English', 'A')]
```

To use different kinds of joins in a single query, append one join statement after another. The join closest to the beginning of the statement is executed first, creating a temporary table, and the next join attempts to operate on that table. The following example performs an additional join on Table 2.3 to find the name and major of every student who got a C in a class.

```
# Do an inner join on the results of the left outer join.
>>> cur.execute("SELECT SI.StudentName, MI.MajorName "
...                 "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...                 "ON SI.MajorID == MI.MajorID "
...                 "INNER JOIN StudentGrades AS SG "
...                 "ON SI.StudentID = SG.StudentID "
...                 "WHERE SG.Grade = 'C';").fetchall()
[('Michelle Fernandez', 'Math'),
('Roberta Cook', 'Science'),
('Cameron Kim', 'Art'),
('Alfonso Phelps', None)]
```

In this last example, note carefully that Alfonso Phelps would have been excluded from the result set if an inner join was performed first instead of an outer join (since he lacks a major).

Problem 2. Write a function that accepts the name of a database file. Query the database for all tuples of the form `(Name, MajorName, Grade)` where `Name` is a student's name and `Grade` is their grade in Calculus. Only include results for students that are actually taking Calculus, but be careful not to exclude students who haven't declared a major.

Grouping Data

Many data sets can be naturally sorted into groups. The `GROUP BY` command gathers rows from a table and groups them by a certain attribute. The groups must be then combined by one of the *aggregate functions* `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`. The following code groups the rows in Table 2.1d by `studentID` and counts the number of entries in each group.

```
>>> cur.execute("SELECT StudentID, COUNT(*) "    # * means "all of the rows".
...           "FROM StudentGrades "
...           "GROUP BY StudentID").fetchall()
[(103066521, 3),
 (206208438, 2),
 (262019426, 2),
 (341324754, 2),
 (401767594, 2),
 (553725811, 1),
 (622665098, 2),
 (678665086, 3),
 (821568627, 3),
 (886308195, 1)]
```

`GROUP BY` can also be used in conjunction with joins. The join creates a temporary table like Tables 2.2 or 2.3, the results of which can then be grouped.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) "
...           "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...           "ON SG.StudentID == SI.StudentID "
...           "GROUP BY SG.StudentID").fetchall()
[('Cameron Kim', 3),
 ('Kristopher Tran', 2),
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Michelle Fernandez', 2),
 ('Roberta Cook', 1),
 ('Sammy Burke', 2),
 ('Gilbert Chapman', 3),
 ('Mercedes Hall', 3),
 ('Rene Cross', 1)]
```

Just like the `WHERE` clause chooses rows in a relation, the `HAVING` clause chooses groups from the result of a `GROUP BY` based on some criteria related to the groupings. For this particular command, it is often useful (but not always necessary) to create an alias for the columns of the result set with the `AS` operator. For instance, the result set of the previous example can be filtered down to only contain students who are taking 3 courses.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) as num_courses " # Alias.
...           "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...           "ON SG.StudentID == SI.StudentID "
...           "GROUP BY SG.StudentID "
...           "HAVING num_courses == 3").fetchall() # Refer to alias later.
[('Cameron Kim', 3), ('Gilbert Chapman', 3), ('Mercedes Hall', 3)]
```

Alternatively, get just the student names.

```
>>> cur.execute("SELECT SI.StudentName " # No alias.
...           "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...           "ON SG.StudentID == SI.StudentID "
...           "GROUP BY SG.StudentID "
...           "HAVING COUNT(*) == 3").fetchall()
[('Cameron Kim',), ('Gilbert Chapman',), ('Mercedes Hall',)]
```

Problem 3. Write a function that accepts a database file. Query the database for the list of the names of courses that have at least 5 student enrolled in them.

Other Miscellaneous Commands

Ordering Result Sets

The `ORDER BY` command sorts a result set by one or more attributes. Sorting can be done in ascending or descending order with `ASC` or `DESC`, respectively. This is always the very last statement in a query.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) AS num_courses " # Alias.
...           "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...           "ON SG.StudentID == SI.StudentID "
...           "GROUP BY SG.StudentID "
...           "ORDER BY num_courses DESC, SI.StudentName ASC").fetchall()
[('Cameron Kim', 3), # The results are now ordered by the
 ('Gilbert Chapman', 3), # number of courses each student is in,
 ('Mercedes Hall', 3), # then alphabetically by student name.
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Kristopher Tran', 2),
 ('Michelle Fernandez', 2),
 ('Sammy Burke', 2),
 ('Rene Cross', 1),
 ('Roberta Cook', 1)]
```

Problem 4. Write a function that accepts a database file. Query the given database for tuples of the form (MajorName, N) where N is the number of students in the specified major. Sort the results in ascending order by the count N.

Searching Text with Wildcards

The `LIKE` operator within a `WHERE` clause matches patterns in a `TEXT` column. The special characters % and _ and called *wildcards* that match any number of characters or a single character, respectively. For instance, `%Z_` matches any string of characters ending in a Z then another character, and `%i%` matches any string containing the letter i.

```
>>> results = cur.execute("SELECT StudentName FROM StudentInfo "
...                         "WHERE StudentName LIKE '%i%'").fetchall()
>>> [r[0] for r in results]
['Michelle Fernandez', 'Gilbert Chapman', 'Cameron Kim', 'Kristopher Tran']
```

Problem 5. Write a function that accepts a database file. Query the database for tuples of the form (StudentName, MajorName) where the last name of the specified student begins with the letter C.

Case Expressions

A case expression maps the values in a column using boolean logic. There are two forms of a case expression: simple and searched. A *simple case expression* matches and replaces specified attributes.

```
# Replace the values MajorID with new custom values.
>>> cur.execute("SELECT StudentName, CASE MajorID "
...                  "WHEN 1 THEN 'Mathematics' "
...                  "WHEN 2 THEN 'Soft Science' "
...                  "WHEN 3 THEN 'Writing and Editing' "
...                  "WHEN 4 THEN 'Fine Arts' "
...                  "ELSE 'Undeclared' END "
...                  "FROM StudentInfo "
...                  "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Fine Arts'),
 ('Cassandra Holland', 'Mathematics'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Soft Science'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Mathematics'),
 ('Rene Cross', 'Writing and Editing'),
 ('Roberta Cook', 'Soft Science'),
 ('Sammy Burke', 'Soft Science')]
```

A *searched case expression* involves using a boolean expression at each step, instead of listing all of the possible values for an attribute.

```
# Change NULL values in MajorID to 'Undeclared' and non-NULL to 'Declared'.
>>> cur.execute("SELECT StudentName, CASE "
...             "WHEN MajorID IS NULL THEN 'Undeclared' "
...             "ELSE 'Declared' END "
...             "FROM StudentInfo "
...             "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Declared'),
 ('Cassandra Holland', 'Declared'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Declared'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Declared'),
 ('Rene Cross', 'Declared'),
 ('Roberta Cook', 'Declared'),
 ('Sammy Burke', 'Declared')]
```

Chaining Queries

The result set of any SQL query is really just another table with data from the original database. Separate queries can be made from result sets by enclosing the entire query in parentheses. For these sorts of operations, it is very important to carefully label the columns resulting from a subquery.

```
>>> cur.execute("SELECT majorstatus, COUNT(*) AS majorcount "
...               "FROM ( "
...                   "# Begin subquery.
...                   "SELECT StudentName, CASE "
...                   "WHEN MajorID IS NULL THEN 'Undeclared' "
...                   "ELSE 'Declared' END AS majorstatus "
...                   "FROM StudentInfo) "
...               "# End subquery.
...               "GROUP BY majorstatus "
...               "ORDER BY majorcount DESC;").fetchall()
[('Declared', 7), ('Undeclared', 3)]
```

Problem 6. Write a function that accepts the name of a database file. Query the database for tuples of the form (StudentName, N, GPA) where N is the number of courses that the specified student is enrolled in and GPA is their grade point average based on the following point system.

$$\begin{array}{llll}
 A+, A = 4.0 & B = 3.0 & C = 2.0 & D = 1.0 \\
 A- = 3.7 & B- = 2.7 & C- = 1.7 & D- = 0.7 \\
 B+ = 3.4 & C+ = 2.4 & D+ = 1.4
 \end{array}$$

Order the results from greatest GPA to least.

3

Unix Shell 1: Introduction

Lab Objective: *Explore the basics of the Unix Shell. Understand how to navigate and manipulate file directories. Introduce the Vim text editor for easy writing and editing of text or other similar documents.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of Linux and MacOSX. The Unix shell is an interface for executing commands to the operating system. The majority of servers are Linux based, so having a knowledge of Unix shell commands allows us to interact with these servers.

As you get into Unix, you will find it is easy to learn but difficult to master. We will build a foundation of simple file system management and a basic introduction to the Vim text editor. We will address some of the basics in detail and also include lists of commands that interested learners are encouraged to research further.

NOTE

Windows is not built off of Unix, but it does come with a command line tool. We will not cover the equivalent commands in Windows command line, but you could download a Unix-based shell such as Git Bash or Cygwin to complete this lab (you will still lose out on certain commands).

File System

ACHTUNG!

In this lab you will work with files on your computer. Be careful as you go through each problem and as you experiment on your own. Be sure you are in the right directories and subfolders before you start creating and deleting files; some actions are irreversible.

Navigation

Typically you have probably navigated your computer by clicking on icons to open directories and programs. In the terminal, instead of point and click we use typed commands to move from directory to directory.

Begin by opening the Terminal. The text you see in the upper left of the Terminal is called the *prompt*. As you navigate through the file system you will want to know *where* you are so that you know you aren't creating or deleting files in the wrong locations.

To see what directory you are currently working in, type `pwd` into the prompt. This command stands for **p**rint **w**orking **d**irectory, and as the name suggests it prints out the string of your current location.

Once you know where you are, you'll want to know where you can move. The `ls`, or **l**ist **s**egments, command will list all the files and directories in your current folder location. Try typing it in.

When you know what's around you, you'll want to navigate directories. The `cd`, or **c**hange **d**irectory, command allows you to move through directories. To change to a new directory, type the `cd` command followed by the name of the directory to which you want to move (if you `cd` into a file, you will get an error). You can move up one directory by typing `cd ...`

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing `cd ~` or just `cd`. You can navigate to root by typing `cd /`.

Problem 1. Using these commands, navigate to the `Shell1/` directory provided with this lab. We will use this directory for the remainder of the lab. Use the `ls` command to list the contents of this directory. NOTE: You will find a directory within this directory called `Test/` that is available for you to experiment with the concepts and commands found in this lab. The other files and directories are necessary for the exercises we will be doing, so take care not to modify them.

Getting Help

As you go through this lab, you will come across many commands with functionality beyond what is taught here. The Terminal has two nice commands to help you with these commands. The first is `man <command>`, which opens the manual page for the command following `man`. Try typing in `man ls`; you will see a list of the name and description of the `ls` command, among other things. If you forget how to use a command the manual page is the first place you should check to remember.

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags	Description
-a	Do not ignore hidden files and folders
-l	List files and folders in long format
-r	Reverse order while sorting
-R	Print files and subdirectories recursively
-s	Print item name and size
-S	Sort by size
-t	Sort output by date modified

Table 3.1: Common flags of the ls command.

Flags

When you typed in `man ls` up above, you may have noticed several options listed in the description, such as `-a`, `-A`, `--author`. These are called flags and change the functionality of commands. Most commands will have flags that change their behavior. Table 3.1 contains some of the most common flags for the `ls` command.

Multiple flags can be combined as one flag. For example, if we wanted to list all the files in a directory in long format sorted by date modified, we would use `ls -a -l -t` or `ls -alt`.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. Before you begin, `cd` into the `Test/` directory in `Shell1/`.

To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, the command is similar but must use the `-r` flag. This flag stands for recursively copying files in subdirectories. If you try to copy a file without the `-r` the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second. If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

When deleting files, use `rm <filename>`, or `rm -r <dir_name>` when deleting a directory. Again, the `-r` flag tells the Terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to have the Terminal print strings of what it is doing. When your Terminal gets too cluttered, use `clear` to clean it up.

Below is an example of all these commands in action.

```
$ cd Test
$ touch data.txt          # create new empty file data.txt
$ mkdir New               # create directory New
$ ls                      # list items in test directory
New      data.txt
$ cp data.txt New/        # copy data.txt to New directory
$ cd New/
$ ls                      # list items in New directory
```

Commands	Description
<code>clear</code>	Clear the terminal screen
<code>cp file1 dir1</code>	Create a copy of <code>file1</code> and move it to <code>dir1/</code>
<code>cp file1 file2</code>	Create a copy of <code>file1</code> and name it <code>file2</code>
<code>cp -r dir1 dir2</code>	Create a copy of <code>dir1/</code> and all its contents into <code>dir2/</code>
<code>mkdir dir1</code>	Create a new directory named <code>dir1/</code>
<code>mkdir -p path/to/new/dir1</code>	Create <code>dir1/</code> and all intermediate directories
<code>mv file1 dir1</code>	Move <code>file1</code> to <code>dir1/</code>
<code>mv file1 file2</code>	Rename <code>file1</code> as <code>file2</code>
<code>rm file1</code>	Delete <code>file1</code> [-i, -v]
<code>rm -r dir1</code>	Delete <code>dir1/</code> and all items within <code>dir1/</code> [-i, -v]
<code>touch file1</code>	Create an empty file named <code>file1</code>

Table 3.2: The commands discussed in this section.

```

data.txt
$ mv data.txt new_data.txt          # rename data.txt new_data.txt
$ ls                                # list items in New directory
new_data.txt
$ cd ..                             # Return to test directory
$ rm -rv New/                         # Remove New directory and its contents
removed 'New/data.txt'
removed directory: 'New/'
$ clear                            # Clear terminal screen

```

Table 3.2 contains all the commands we have discussed so far. Notice the common flags are contained in square brackets; use `man` to see what these mean.

Problem 2. Inside the `Shell1/` directory, delete the `Audio/` folder along with all its contents. Create `Documents/`, `Photos/`, and `Python/` directories.

Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, if you needed to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using *wildcards*. We will use the * and ? wildcards. The * wildcard represents any string and the ? wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```

$ ls
File1.txt  File2.txt  File3.jpg   text_files
$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt
$ ls
File3.jpg   text_files

```

Command	Description
<code>*.txt</code>	All files that end with <code>.txt</code> .
<code>image*</code>	All files that have <code>image</code> as the first 5 characters.
<code>*py*</code>	All files that contain <code>py</code> in the name.
<code>doc*.txt</code>	All files of the form <code>doc1.txt</code> , <code>doc2.txt</code> , <code>docA.txt</code> , etc.

Table 3.3: Common uses for wildcards.

Command	Description
<code>cat</code>	Print the contents of a file in its entirety
<code>more</code>	Print the contents of a file one page at a time
<code>less</code>	Like more, but you can navigate forward and backward
<code>head</code>	Print the first 10 lines of a file
<code>head -nK</code>	Print the first K lines of a file
<code>tail</code>	Print just the last 10 lines of a file
<code>tail -nK</code>	Print the last K lines of a file

Table 3.4: Commands for printing contents of a file

See Table 3.3 for examples of common wildcard usage.

Problem 3. Within the `Shell1/` directory, there are many files. We will organize these files into directories. Using wildcards, move all the `.jpg` files to the `Photos/` directory, all the `.txt` files to the `Documents/` directory, and all the `.py` files to the `Python/` directory. You will see a few other folders in the `Shell1/` directory. Do not move any of the files within these folders at this point.

Displaying File Contents

When using the file system, you may be interested in checking file content to be sure you're looking at the right file. Several commands are made available for ease in reading file content.

The `cat` command, followed by the filename will display all the contents of a file on the screen. If you are dealing with a large file, you may only want to view a certain number of lines at a time. Use `less <filename>` to restrict the number of lines that show up at a time. Use the arrow keys to navigate up and down. Press `q` to exit.

For other similar commands, look at table 3.4.

Searching the File System

There are two commands we use for searching through our directories. The `find` command is used to find files or directories in a directory hierarchy. The `grep` command is used to find lines matching a string. More specifically, we can use `grep` to find words inside files. We will provide a basic template in Table 3.5 for using these two commands and leave it to you to explore the uses of the other flags. The `man` command can help you learn about them.

Command	Description
<code>find dir1 -type f -name "word"</code>	Find all files in <code>dir1/</code> (and its subdirectories) called <code>word</code> (-type <code>f</code> is for files; -type <code>d</code> is for directories)
<code>grep "word" filename</code>	Find all occurrences of <code>word</code> within <code>filename</code>
<code>grep -nr "word" dir1</code>	Find all occurrences of <code>word</code> within the files inside <code>dir1/</code> (-n lists the line number; -r performs a recursive search)

Table 3.5: Commands using `find` and `grep`.

Problem 4. In addition to the `.jpg` files you have already moved into the `Photot/` folder, there are a few other `.jpg` files in a few other folders within the `Shell1/` directory. Find where these files are using the `find` command and move them to the `Photos/` folder.

Pipes and Redirects

Terminal commands can be combined using *pipes*. When combined, or *piped*, the output of one command is passed to the another. Two commands are piped together using the `|` operator. To demonstrate pipes we will first introduce commands that allow us to view the contents of a file in Table 3.4.

In the first example below, the `cat` command output is piped to `wc -l`. The `wc` command stands for `word count`. This command can be used to count words or lines. The `-l` flag tells the `wc` command to count lines. Therefore, this first example counts the number of lines in `assignments.txt`. In the second example below, the command lists the files in the current directory sorted by size in descending order. For details on what the flags in this command do, consult `man sort`.

```
$ cd Shell1/Files/Feb
$ cat assignments.txt | wc -l
9

$ ls -s | sort -nr
12 project3.py
12 project2.py
12 assignments.txt
 4 pics
total 40
```

In the previous example, we pipe the contents of `assignments.txt` to `wc -l` using `cat`. When working with files specifically, you can also use *redirects*. The `<` operator gives a file to a Terminal command. The same output from the first example above can be achieved by running the following command:

```
$ wc -l < assignments.txt
9
```

If you are wanting to save the resulting output of a command to a file, use `>` or `>>`. The `>` operator will overwrite anything that may exist in the output file whereas `>>` will append the output to the end of the output file. For example, if we want to append the number of lines in `assignments.txt` to `word_count.txt`, we would run the following command:

```
$ wc -l < assignements.txt >> word_count.txt
```

Since `grep` is used to print lines matching a pattern, it is also very useful to use in conjunction with piping. For example, `ls -l | grep root` prints all files associated with the root user.

Problem 5. The words.txt file in the Documents/ directory contains a list of words that are not in alphabetical order. Write the number of words in words.txt and an alphabetically sorted list of words to sortedwords.txt using pipes and redirects. Save this file in the Documents/ directory. Try to accomplish this with a total of two commands or fewer.

Archiving and Compression

In file management, the terms archiving and compressing are commonly used interchangeably. However, these are quite different. To archive is to combine a certain number of files into one file. The resulting file will be the same size as the group of files that were archived. To compress is to take a file or group of files and shrink the file size as much as possible. The resulting compressed file will need to be extracted before being used.

The ZIP file format is the most popular for archiving and compressing files. If the `zip` Unix command is not installed on your system, you can download it by running `sudo apt-get install zip`. Note that you will need to have administrative rights to download this package. To unzip a file, use `unzip`.

```
$ cd Shell1/Documents  
$ zip zipfile.zip doc?.txt  
adding: doc1.txt (deflated 87%)  
adding: doc2.txt (deflated 90%)  
adding: doc3.txt (deflated 85%)  
adding: doc4.txt (deflated 97%)  
  
# use -l to view contents of zip file  
$ unzip -l zipfile.zip  
Archive: zipfile.zip  
Length      Date    Time     Name  
-----  
      5234  2015-08-26 21:21  doc1.txt  
      7213  2015-08-26 21:21  doc2.txt  
      3634  2015-08-26 21:21  doc3.txt  
      4516  2015-08-26 21:21  doc4.txt  
-----  
     16081                               3 files
```

```
inflating: doc1.txt
inflating: doc2.txt
inflating: doc3.txt
inflating: doc4.txt
```

While the zip file format is more popular on the Windows platform, the `tar` utility is more common in the Unix environment. The following commands use `tar` to archive the files and `gzip` to compress the archive.

Notice that all the commands below have the `-z`, `-v`, and `-f` flags. The `-z` flag calls for the `gzip` compression tool, the `-v` flag calls for a verbose output, and `-f` indicates the next parameter will be the name of the archive file.

```
$ ls
doc1.txt    doc2.txt    doc3.txt    doc4.txt

# use -c to create a new archive
$ tar -zcvf docs.tar.gz doc?.txt
doc1.txt
doc2.txt
doc3.txt
doc4.txt

$ ls
docs.tar.gz

# use -t to view contents
$ tar -ztvf <archive>
-rw-rw-r-- username/groupname 5119 2015-08-26 16:50 doc1.txt
-rw-rw-r-- username/groupname 7253 2015-08-26 16:50 doc2.txt
-rw-rw-r-- username/groupname 3524 2015-08-26 16:50 doc3.txt
-rw-rw-r-- username/groupname 4516 2015-08-26 16:50 doc4.txt

# use -x to extract
$ tar -zxvf <archive>
doc1.txt
doc2.txt
doc3.txt
doc4.txt
```

Problem 6. Archive and compress the files in the `Photos/` directory using `tar` and `gzip`. Name the archive `pics.tar.gz` and save it inside the `Photos/` directory. Use `ls -l` to see how much the files were compressed in the process.

Vim: A Terminal Text Editor

Today many have become accustomed to having GUIs (Graphic User Interfaces) for all their applications. Before modern text editors (i.e. Microsoft Word, Pages for Mac, Google Docs) there were terminal text editors. Vim is one of the most popular terminal text editors. While vim may be intimidating at first, as you become familiar with vim it may become one of your preferred text editors for writing code.

One of the major philosophies of vim is to be able to keep your fingers on the keyboard at all times. Thus, vim has many keyboard shortcuts that allow you to navigate the file and execute commands without relying on a mouse, toolbars, or arrow keys.

In this section, we will go over the basics of navigation and a few of the most common commands. We will also provide a list of commands that interested readers are encouraged to research.

It has been said that at no point does somebody finish learning Vim. You will find that you will constantly be able to add something new to your arsenal.

Getting Started

Start Vim with the following command:

```
$ vim my_file.txt
```

When executing this command, if `my_file.txt` already exists, vim will open the file and we may begin editing the existing file. If `my_file.txt` does not exist, it will be created and we may begin editing the file.

You may notice if you start typing the characters may or may not appear on your screen. This is because vim has multiple modes. When vim starts, we are placed in *command mode*. We want to be in *insert mode* to begin entering text. To enter insert mode from command mode, hit the `i` key. You should see `-- INSERT --` at the bottom of your terminal window. In insert mode vim act like a typical word processor. Letters will appear in the document as you type them. If you ever need to leave insert mode and return to command mode, hit the `Esc` key.

Saving/Quitting Vim

To save or quit the current document, first enter last line mode by pressing the `:` key. To just save, type `w` and hit enter. To save and quit, type `wq`. To quit without saving, run `q!`

Problem 7. Using vim, create a new file in the `Documents/` directory named `first_vim.txt`. Write least multiple lines to this file. Save and exit the file you have created.

Navigation

We are accustomed to navigating GUI text editors using a mouse and arrow keys. In vim, we navigate using keyboard shortcuts while in command mode.

Command	Description
a	append text after cursor
A	Append text to end of line
o	Begin a new line below the cursor
O	Begin a new line above the cursor
s	Substitute characters under cursor

Table 3.6: Commands for entering insert mode

Problem 8. Become accustomed to navigating in command mode using the following keys:

Command	Description
k	up
j	down
h	left
l	right
w	beginning of next word
e	end of next word
b	beginning of previous word
0	(zero) beginning of line
\$	end of line
gg	beginning of file
#gg	go to line #
G	end of file

Alternative Ways to Enter Insert Mode

Hitting the `i` key is not the only way to enter insert mode. Alternative methods are described in Table 3.6.

Visual Mode

Visual mode allows you to select multiple characters. Among other things, we can use this to replace words with the `s` command, and we can select text to cut or copy.

Problem 9. Open the document you created in the previous problem. While in command mode, enter visual mode by pressing the `v` key. Using the navigation keys discussed earlier, move the cursor to select a few words. Copy this text using the `y` key (stands for `yank`). Return to command mode by pressing `Esc`. Move the cursor to where you would like to paste the text and press the `p` key to paste. Similarly, select text in visual mode and hit `d` to delete the text and paste it somewhere else with the `p` key.

Command	Description
x	delete letter after cursor
X	delete letter before cursor
dd	delete line
d1	delete letter
d#l	delete # letters
dw	delete word
d#w	delete # words

Table 3.7: Commands for deleting in command mode

Command	Description
:map	customize
:help	view vim docs
cw	change word
u	undo
Ctrl-R	redo
.	Repeat the previous command
*	find next occurrence of word under cursor
#	find previous occurrence of word under cursor
/str	find str in file
n	find next match
N	find previous match

Table 3.8: Commands for entering insert mode

Deleting Text in Command Mode

Insert mode should only be used for inserting text. Try to get in the habit of leaving insert mode as soon as you are done adding the text you want to add. Deleting text is much more efficient and versatile in command mode. The x and X commands are used to delete single characters. The d command is always accompanied by another navigational command. See Table 3.7 for a few examples.

A Few Closing Remarks

In the next lab, we will introduce how to access another machine through the terminal. Vim will be essential in this situation since GUIs will not be an option.

If you are interested in continuing to use vim, you may be interested in checking out *gvim*. Gvim is a GUI that uses vim commands in a more traditional text editor window.

Also, in Table 3.8, we have listed a few more commands that are worth exploring. If you are interested in any of these features of vim, we encourage you to research these features further on the internet. Additionally, many people have published their *vimrc* file on the internet so other vim users can learn what options are worth exploring. It is also worth noting that we can use vim navigation commands in many other places in the shell. For example, try using the navigation commands when viewing the *man vim* page.

4

Unix Shell 2

Lab Objective: *Introduce system management, calling Unix Shell commands within Python, and other advanced topics. As in the last lab, the majority of learning will not be had in finishing the problems, but in following the examples.*

File Security

To begin, run the following command while inside the `Shell2/Python/` directory (`Shell2/` is the end product of `Shell1/` from the previous lab). Notice your output will differ from that printed below; this is for learning purposes.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
-rw-rw-r-- 1 username groupname 373 Aug  5 21:16 count_files.py
-rwxr-xr-x 1 username groupname   27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

Notice the first column of the output. The first character denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The remaining nine characters denote the permissions associated with that file. Specifically, these permissions deal with reading, writing, and executing files. There are three categories of people associated with permissions. These are the user (the owner), group, and others. For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx` tell us the owner can read, write, and execute the file. The next three characters `r-x` tell us members of the same group can read and execute the file. The final three characters `--x` tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are two different ways to specify permissions, *symbolic permissions* notation and *octal permissions* notation. Symbolic permissions notation is easier to use when we want to make small modifications to a file's permissions. See Table 4.1.

Octal permissions notation is easier to use when we want to set all the permissions at once. The number 4 corresponds to reading, 2 corresponds to writing, and 1 corresponds to executing. See Table 4.2.

The commands in Table 4.3 are also helpful when working with permissions.

Command	Description
chmod u+x file1	Add executing (x) permissions to user (u)
chmod g-w file1	Remove writing (w) permissions from group (g)
chmod o-r file1	Remove reading (r) permissions from other other users (o)
chmod a+w file1	Add writing permissions to everyone (a)

Table 4.1: Symbolic permissions notation

Command	Description
chmod 760 file1	Sets rwx to user, rw- to group, and --- to others
chmod 640 file1	Sets rw- to user, r-- to group, and --- to others
chmod 775 file1	Sets rwx to user, rwx to group, and r-x to others
chmod 500 file1	Sets r-x to user, --- to group, and --- to others

Table 4.2: Octal permissions notation

Scripts

A shell script is a series of shell commands saved in a file. Scripts are useful when we have a process that we do over and over again. The following is a very simple script.

```
#!/bin/bash
echo "Hello World"
```

The first line starts with `"#!"`. This is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the `bash` interpreter. If we were unsure where the `bash` interpreter is saved, we run `which bash`.

Problem 1. Create a file called `hello.sh` that contains the previous text and save it in the `Scripts/` directory. The file extension `.sh` is technically unnecessary, but it is good practice to always include an extension.

To execute a script, type the script name preceded by `./`

```
$ cd Scripts
$ ./hello.sh
bash: ./hello.sh: Permission denied

# Notice you do not have permission to execute this file. This is by default.
$ ls -l hello.sh
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh
```

Problem 2. Add executable permissions to `hello.sh`. Run the script to verify that it worked.

You can do this same thing with Python scripts. All you have to do is change the path following the shebang. To see where the Python interpreter is stored, run `which python`.

Command	Description
<code>chown</code>	change owner
<code>chgrp</code>	change group
<code>getfacl</code>	view all permissions of a file in a readable format.

Table 4.3: Other commands when working with permissions

Command	Description
<code>df dir1</code>	Display available disk space in file system containing <code>dir1</code>
<code>du dir1</code>	Display disk usage within <code>dir1</code> [-a, -h]
<code>free</code>	Display amount of free and used memory in the system
<code>ps</code>	Display a snapshot of current processes
<code>top</code>	Display interactive list of current processes

Table 4.4: Commands for resource management

Problem 3. The Python/ directory contains a file called `count_files.py`, a Python script that counts all the files within the `Shell2/` directory. Modify this file so it can be run as a script and change the permissions of this script so the user and group can execute the script.

NOTE

If you would like to learn how to run scripts on a set schedule, consider researching *cron jobs*.

Resource Management

To be able to optimize performance, it is valuable to always be aware of the resources we are using. Hard drive space and computer memory are two resources we must constantly keep in mind. The commands found in Table 4.4 are essential to managing resources.

Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In Table 4.5, we have listed some of the most common commands used in job control. We strongly encourage you to experiment with these commands.

Command	Description
COMMAND &	Adding an ampersand to the end of a command runs the command in the background
bg %N	Restarts the Nth interrupted job in the background
fg %N	Brings the Nth job into the foreground
jobs	Lists all the jobs currently running
kill %N	Terminates the Nth job
ps	Lists all the current processes
Ctrl-C	Terminates current job
Ctrl-Z	Interrupts current job
nohup	Run a command that will not be killed if the user logs out

Table 4.5: Job control commands

The `five_secs.sh` and `ten_secs.sh` scripts in the `Scripts/` directory take five seconds and ten seconds to execute respectively. These will be particularly useful as you are experimenting with these commands.

```
# Don't forget to change permissions if needed
$ ./ten_secs.sh &
$ ./five_secs.sh &
$ jobs
[1]+  Running          ./ten_secs.sh &
[2]-  Running          ./five_secs.sh &
$ kill %2
[2]-  Terminated      ./five_secs.sh &
$ jobs
[1]+  Running          ./ten_secs.sh &
```

Problem 4. In addition to the `five_secs.sh` and `ten_secs.sh` scripts, the `Scripts/` folder contains three scripts that each take about forty-five seconds to execute. Execute each of these commands in the background so all three are running at the same time. While they are all running, write the output of `jobs` to a new file `log.txt` saved in the `Scripts/` directory.

Using Python for File Management

Bash itself has control flow tools like if-else blocks and loops, but most of the syntax is highly unintuitive. Python, on the other hand, has extremely intuitive syntax for these control flow tools, so using Python to do shell-like tasks can result in some powerful but specific file management programs. Table 4.6 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 4.7. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`.

Shell Command	Python Function
ls	os.listdir()
cd	os.chdir()
pwd	os.getcwd()
mkdir	os.mkdir(), os.mkdirs()
cp	shutil.copy()
mv	os.rename(), os.replace()
rm	os.remove(), shutil.rmtree()
du	os.path.getsize()
chmod	os.chmod()

Table 4.6: Shell-Python compatibility

Function	Description
os.walk()	Iterate through the subfolders of a given directory.
os.path.isdir()	Return <code>True</code> if the input is a directory.
os.path.isfile()	Return <code>True</code> if the input is a file.
os.path.join()	Join several folder names or file names into one path.
glob.glob()	Return a list of file names that match a pattern.
subprocess.call()	Execute a shell command.
subprocess.check_output()	Execute a shell command and return its output as a string.

Table 4.7: Other useful Python functions for shell operations.

```
>>> import os
>>> from glob import glob

# Get the names of all Python files in the Python/ directory.
>>> glob("Python/*.py")
['Python/calc.py',
 'Python/count_files.py',
 'Python/mult.py',
 'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
>> glob("**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/img_1796.jpg',
 'Photos/img_1842.jpg',
 'Photos/img_1879.jpg',
 'Photos/img_1987.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2464.jpg',
```

```
'Photos/IMG_2679.jpg',
'Photos/IMG_2746.jpg']

# Walk through the directory, looking for .sh files.
>>> for directory, subdirectories, files in os.walk('.'):
...     for filename in files:
...         if filename.endswith(".sh"):
...             print(os.path.join(directory, filename))
...
./Scripts/five_secs.sh
./Scripts/script1.sh
./Scripts/script2.sh
./Scripts/script3.sh
./Scripts/ten_secs.sh
```

Problem 5. Write a Python function `grep()` that accepts the name of target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern, then determine which ones contain the target string. For example, `grep("*.py", "range()")` should search Python files for the command `range()`.

Validate your function by comparing it to `grep -lR` in the shell.

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Use `subprocess.call()` to run a Unix command, or `subprocess.check_output()` to run a Unix command and record its output.

```
$ cd Shell2/Scripts
$ python
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r-- 1 username groupname 20 Aug 26 2016 five_secs.sh
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script1.sh
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script2.sh
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script3.sh
-rw-r--r-- 1 username groupname 21 Aug 26 2016 ten_secs.sh
0                                     # decode() translates the result to a string.
>>> file_info = subprocess.check_output(["ls", "-l"]).decode()
>>> file_info.split('\n')
['total 40',
 '-rw-r--r-- 1 username groupname 20 Aug 26 2016 five_secs.sh',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script1.sh',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script2.sh',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script3.sh',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 ten_secs.sh',
 '']
```

ACHTUNG!

Be extremely careful when creating a shell process from Python. If the commands depend on user input, the program is vulnerable to a *shell injection attack*. For example, consider the following function.

```
>>> def inspect_file(filename):
...     """Return information about the specified file from the shell."""
...     return subprocess.check_output(["ls", "-l", filename]).decode()
```

If `inspect_file()` is given the input `".; rm -rf /"`, then `ls -l .` is executed innocently, and then `rm -rf /` destroys the computer.^a Be careful not to execute a shell command from within Python in a way that a malicious user could potentially take advantage of.

^aSee https://en.wikipedia.org/wiki/Code_injection#Shell_injection for more example attacks.

Problem 6. Write a Python function that accepts an integer n . Search the current directory and all subdirectories for the n largest files. Return a list of filenames, in order from largest to smallest.

(Hint: the shell commands `ls -s` and `du` show the file size.)

Downloading Files

The Unix shell has tools for downloading files from the internet. The most popular are `wget` and `curl`. At its most basic, `curl` is the more robust of the two while `wget` can download recursively.

When we want to download a single file, we just need the URL for the file we want to download. This works for PDF files, HTML files, and other content simply by providing the right URL.

```
$ wget https://github.com/Foundations-of-Applied-Mathematics/Data/blob/master/←
Volume1/dream.png
```

The following are also useful commands using `wget`.

```
# Download files from URLs listed in urls.txt
$ wget -i list_of_urls.txt

# Download in the background
$ wget -b URL

# Download something recursively
$ wget -r --no-parent URL
```

Problem 7. The file `urls.txt` in the `Documents/` directory contains a list of URLs. Download the files in this list using `wget` and move them to the `Photos/` directory.

When you finish this problem, archive and compress your entire `Shell12/` directory and save it as `ShellFinal.tar.gz`. Include the `-p` flag on `tar` to preserve the file permissions.

(Hint: see the previous lab for a refresher on `tar`. See also <https://xkcd.com/1168/>.)

One Final Note

Though there are multiple Unix shells, one of the most popular is the *bash* shell. The bash shell is highly customizable. In your home directory, you will find a hidden file named `.bashrc`. All customization changes are saved in this file. If you are interested in customizing your shell, you can customize the prompt using the `PS1` environment variable. As you become more and more familiar with the Unix shell, you will come to find there are commands you run over and over again. You can save commands you use frequently using `alias`. If you would like more information on these and other ways to customize the shell, you can find many quality reference guides and tutorials on the internet.

Additional Material

sed and awk

`sed` and `awk` are two different scripting languages in their own right. Like Unix, these languages are easy to learn but difficult to master. It is very common to combine Unix commands and `sed` and `awk` commands. We will address the basics, but if you would like more information see <<http://www.theunixschool.com/p/awk-sed.html>>

Printing Specific Lines Using sed

We have already used the `head` and `tail` commands to print the beginning and end of a file respectively. What if we wanted to print lines 30 to 40, for example? We can accomplish this using `sed`. In the `Documents/` folder, you will find the `lines.txt` file. We will use this file for the following examples.

```
# Same output as head -n3
$ sed -n 1,3p lines.txt
line 1
line 2
line 3

# Same output as tail -n3
$ sed -n 3,5p lines.txt
line 3
line 4
line 5

# Print lines 2-4
$ sed -n 3,5p lines.txt
line 2
line 3
line 4

# Print lines 1,3,5
$ sed -n -e 1p -e 3p -e 5p lines.txt
line 1
line 3
line 5
```

Find and Replace Using sed

Using `sed`, we can also perform find and replace. We can perform this function on the output of another command or we can perform this function in place on other files. The basic syntax of this `sed` command is the following.

```
sed s/str1/str2/g
```

This command will replace every instance of **str1** with **str2**. More specific examples follow.

```
$ sed s/line/LINE/g lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5

# Notice the file didn't change at all
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5

# To save the changes, add the -i flag
$ sed -i s/line/LINE/g lines.txt
$ cat lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
```

Formatting output using awk

Earlier in this lab we mentioned `ls -l` and as we have seen, this outputs lots of information. Using `awk`, we can select which fields we wish to print. Suppose we only cared about the file name and the permissions. We can get this output by running the following command.

```
$ ls -l | awk ' {print $1, $9}'
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we use quotation marks. Note it is a common mistake to forget to add these quotation marks. Inside these quotation marks, commands always take the same format.

```
awk ' <options> {<actions>} '
```

In the remaining examples we will not be using any of the options, but we will address various actions. For those interested in learning what options are available see <<http://www.theunixschool.com/p/awk-sed.html>>.

In the `Documents/` directory, you will find a `people.txt` file that we will use for the following examples. In our first example, we use the `print` action. The `$1` and `$9` mean that we are going to print the first and ninth fields. Beyond specifying which fields we wish to print, we can also choose how many characters to allocate for each field.

```

# contents of people.txt
$ cat people.txt
male,John,23
female,Mary,31
female,Sally,37
male,Ted,19
male,Jeff,41
female,Cindy,25

# Change the field separator (FS) to "," at the beginning of execution (BEGIN)
# By printing each field individually proves we have successfully separated the←
# fields
$ awk ' BEGIN{ FS = "," }; {print $1,$2,$3} ' < people.txt
male John 23
female Mary 31
female Sally 37
male Ted 19
male Jeff 41
female Cindy 25

# Format columns using printf so everything is in neat columns in order (gender←
# ,age,name)
$ awk ' BEGIN{ FS = ","}; {printf "%-6s %2s %s\n", $1,$3,$2} ' < people.txt
male    23 John
female  31 Mary
female  37 Sally
male    19 Ted
male    41 Jeff
female  25 Cindy

```

The statement `"%-6s %2s %s\n"` formats the columns of the output. This says to set aside six characters left justified, then two characters right justified, then print the last field to its full length.

We have barely scratched the surface of what `awk` can do. Performing an internet search for "awk one-liners" will give you many additional examples of useful commands you can run using `awk`.

System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in Table 4.8 are used to learn more about the computer system.

Command	Description
<code>passwd</code>	Change user password
<code>uname</code>	View operating system name
<code>uname -a</code>	Print all system information
<code>uname -m</code>	Print machine hardware
<code>w</code>	Show who is logged in and what they are doing
<code>whoami</code>	Print userID of current user

Table 4.8: Commands for system administration.

Secure Shell

Let's say you are working for a company with a file server. Hundreds of people need to be able to access the content of this machine, but how is that possible? Or say you have a script to run that requires some serious computing power. How are you going to be able to access your company's super computer to run your script? We do this through *Secure Shell* (SSH).

SSH is a network protocol encrypted using public-key cryptography. The system we are connecting *to* is commonly referred to as the *host* and the system we are connecting *from* is commonly referred to as the *client*. Once this connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type `exit`.

As a warning, you cannot normally SSH into a Windows machine. If you want to do this, search on the web for available options.

```
$ whoami      # use this to see what your current login is
client_username
$ ssh my_host_username@my_hostname

# You will then be prompted to enter the password for my_host_username

$ whoami      # use this to verify that you are logged into the host
my_host_username

$ hostname
my_hostname

$ exit
logout
Connection to my_host_name closed.
```

Now that you are logged in on the host computer, all the commands you execute are as though you were executing them on the host computer.

Secure Copy

When we want to copy files between the client and the host, we use the *secure copy* command, `scp`. The following commands are run when logged into the client computer.

```
# copy filename to the host's system at filepath
```

```
$ scp filename host_username@hostname:filepath  
  
#copy a file found at filepath to the client's system as filename  
$ scp host_username@hostname:filepath filename  
  
# you will be prompted to enter host_username's password in both these ←  
instances
```

Generating SSH Keys (Optional)

If there is a host that we access on a regular basis, typing in our password over and over again can get tedious. By setting up SSH keys, the host can identify if a client is a trusted user without needing to type in a password. If you are interested in experimenting with this setup, a Google search of "How to set up SSH keys" will lead you to many quality tutorials on how to do so.

5

Regular Expressions

Lab Objective: Cleaning and formatting data are fundamental problems in data science. Regular expressions are an important tool for working with text carefully and efficiently, and are useful for both gathering and cleaning data. This lab introduces regular expression syntax and common practices, including an application to a data cleaning problem.

A *regular expression* or *regex* is a string of characters that follows a certain syntax to specify a pattern. Strings that follow the pattern are said to *match* the expression (and vice versa). A single regular expression can match a large set of strings, such as the set all valid email addresses.

ACHTUNG!

There are some universal standards for regular expression syntax, but the exact syntax varies slightly depending on the program or language. However, the syntax presented in this lab (for Python) is sufficiently similar to any other regex system. Consider learning to use regular expressions in vim or your favorite text editor, keeping in mind that there are bound to be slight syntactic differences from what is presented here.

Regular Expression Syntax in Python

The `re` module implements regular expressions in Python. The function `re.compile()` takes in a regular expression string and returns a corresponding *pattern* object, which has methods for determining if and how other strings match the pattern. For example, the `search()` method returns `None` for a string that doesn't match, and a *match* object for a string that does (more on these later).

```
>>> import re
>>> pattern = re.compile("cat")      # Make a pattern for finding 'cat'.
>>> bool(pattern.search("cat"))     # 'cat' matches 'cat', of course.
True
>>> bool(pattern.search("catfish")) # 'catfish' also contains 'cat'.
True
>>> bool(pattern.search("hat"))    # 'hat' does not contain 'cat'.
False
```

ACHTUNG!

The poorly named `match()` method for pattern objects only matches strings that satisfy the pattern **at the beginning** of the string. To answer the question “does any part of my target string match this regular expression?” always use the `search()` method.

```
>>> pattern = re.compile("cat")
>>> bool(pattern.match("catfish"))
True
>>> bool(pattern.match("fishcat"))
False
>>> bool(pattern.search("fishcat"))
True
```

NOTE

Most of the functions in the `re` module are shortcuts for compiling a pattern object and calling one of its methods. For example, the following lines of code are equivalent.

```
>>> bool(re.compile("cat").search("catfish"))
True
>>> bool(re.search("cat", "catfish"))
True
```

Using `re.compile()` is good practice because the resulting object is reusable, while each call to `re.search()` compiles a new (but redundant) pattern object.

Problem 1. Write a function that compiles and returns a regular expression pattern object with the pattern string `"python"`.

Construct positive and negative test cases to test your object. Having good test cases will be important later, so be thorough. Your verification process might start as follows.

```
>>> pattern = re.compile("cat")
>>> positive = ["cat", "catfish", "fish cat", "your cat ran away"]
>>> assert all(pattern.search(p) for p in positive)
```

Literal Characters and Metacharacters

The following string characters (separated by spaces) are *metacharacters* in Python’s regular expressions, meaning they have special significance in a pattern string: `. ^ $ * + ? { } [] \ | ()`.

To construct a regular expression that matches strings with one or more metacharacters in them requires two things. First, use *raw strings* instead of regular Python strings by prefacing the string with an `r`, such as `r"cat"`. The resulting string interprets backslashes as actual backslash characters, rather than the start of an escape sequence like `\n` or `\t`. Second, preface any metacharacters with a backslash to indicate a literal character. For example, the following code constructs a regular expression to match the string `"$3.99? Thanks."`.

```
>>> dollar = re.compile(r"\$3\.\.99\?\? Thanks\.\.")
>>> bool(dollar.search("$3.99? Thanks."))
True
>>> bool(dollar.search("$3\.\.99? Thanks."))
False
>>> bool(dollar.search("$3.99?"))    # Doesn't contain the entire pattern.
False
```

Without raw strings, every backslash in has to be written as a double backslash, which makes many regular expression patterns hard to read (`"\\\$3\\\.99\\?\? Thanks\\\\. "`). Readability counts.

Problem 2. Write a function that compiles and returns a regular expression pattern object that matches the string `"^{@}{(?) [%]{.}{(*)}{[_]}{&}$"`.

The regular expressions of Problems 1 and 2 only match strings that are or include the exact pattern. The metacharacters allow regular expressions to have much more flexibility and control so that a single pattern could match a wide variety of strings, or a very specific set of strings.

To begin, the *line anchor* metacharacters `^` and `$` are used to match the `start` and the `end` of a line of text, respectively. This shrinks the matching set, even when using the `search()` method instead of the `match()` method. For example, the only single-line string that the expression `^x$` matches is `'x'`, whereas the expression `x` can match any string with an `x` in it.

```
>>> has_x, just_x = re.compile(r"x"), re.compile(r"^\x$")
>>> for test in ["x", "xabc", "abcx"]:
...     print(test + ':', bool(has_x.search(test)), bool(just_x.search(test)))
...
x: True True
xabc: True False          # Starts with 'x', but doesn't end with it.
abcx: True False          # Ends with 'x', but doesn't start with it.
```

The *pipe* character `|` is like a logical OR in a regular expression: `A|B` matches A or B.

```
>>> rb, rbg = re.compile(r"^\red$|^\blue$"), re.compile(r"^\red$|^\blue$|^\green$")
>>> for test in ["red", "blue", "green", "redblue"]:
...     print(test + ":", bool(rb.search(test)), bool(rbg.search(test)))
red: True True
blue: True True
green: False True
redblue: False False          # The line anchors prevent matching here.
```

The parentheses () create a *group* in a regular expression. Among other things, a group establishes an order of operations in an expression, much like how parentheses work in an arithmetic expression such as $3 \cdot (4 + 5)$.

```
>>> fish = re.compile(r"^(one|two) fish$")
>>> for test in ["one fish", "two fish", "red fish", "one two fish"]:
...     print(test + ':', bool(fish.search(test)))
...
one fish: True
two fish: True
red fish: False
one two fish: False
```

Problem 3. Write a function that compiles and returns a regular expression pattern object that matches the following strings (and no other strings, even with `re.search()`).

```
"Book store"      "Mattress store"      "Grocery store"
"Book supplier"   "Mattress supplier"   "Grocery supplier"
```

Character Classes

The hard bracket metacharacters [and] are used to create *character classes*, a part of a regular expression that can match a variety of characters. For example, the pattern [abc] matches any of the characters a, b, or c. This is different than a group delimited by parentheses: a group can match multiple characters, while a character class matches only one character. For instance, [abc] does not match ab or abc, and (abc) matches abc but not ab or even a.

Within character classes, there are two additional metacharacters. When ^ appears **as the first character** in a character class, right after the opening bracket [, the character class matches anything **not** specified instead. In other words, ^ is the set complement operation on the character class. Additionally, the dash - specifies a range of values. For instance, [0-9] matches any digit, and [a-z] matches any lowercase letter. Thus [^0-9] matches any character **except** for a digit, and [^a-z] matches any character **except** for a lowercase letters

```
>>> p1, p2 = re.compile(r"^[a-z][^0-9]$"), re.compile(r"^[^abcA-C][0-27-9]$")
>>> for test in ["d8", "aa", "E9", "EE", "d88"]:
...     print(test + ':', bool(p1.search(test)), bool(p2.search(test)))
...
d8: True True
aa: True False          # a is not in [^abcA-C] or [0-27-9].
E9: False True          # E is not in [a-z].
EE: False False          # E is not in [a-z] or [0-27-9].
d88: False False         # Too many characters.
```

Note that [0-27-9] acts like [(0-2)|(7-9)].

There are also a variety of shortcuts that represent common character classes, listed in Table 5.1. Familiarity with these shortcuts makes some regular expressions significantly more readable.

Character	Description
\b	Matches the empty string, but only at the start or end of a word.
\d	Matches any decimal digit; equivalent to [0-9].
\D	Matches any non-digit character; equivalent to [^\d].
\s	Matches any whitespace character; equivalent to [\t\n\r\f\v].
\S	Matches any non-whitespace character; equivalent to [^\s].
\w	Matches any alphanumeric character; equivalent to [a-zA-Z0-9_].
\W	Matches any non-alphanumeric character; equivalent to [^\w].

Table 5.1: Character class shortcuts.

Any of the character class shortcuts can be used within other custom character classes. For example, [_A-Z\s] matches an underscore, capital letter, or whitespace character.

Finally, a period . matches **any** character except for a line break, and is therefore equivalent to [^\n] on UNIX machines and [^\r\n] on Windows machines. This is a very powerful metacharacter; be careful to only use it when part of the regular expression really should match **any** character.

```
# Match any three-character string with a digit in the middle.
>>> pattern = re.compile(r"^.\\d.$")
>>> for test in ["a0b", "888", "n2%", "abc", "cat"]:
...     print(test + ':', bool(pattern.search(test)))
...
a0b: True
888: True
n2%: True
abc: False
cat: False

# Match two letters followed by a number and two non-newline characters.
>>> pattern = re.compile(r"^[a-zA-Z][a-zA-Z]\\d..$")
>>> for test in ["tk421", "bb8!?", "JB007", "Boba?"]:
...     print(test + ':', bool(pattern.search(test)))
...
tk421: True
bb8!?: True
JB007: True
Boba?: False
```

Character	Description
.	Matches any character except a newline.
^	Matches the start of the string.
\$	Matches the end of the string or just before the newline at the end of the string.
	A B creates an regular expression that will match either A or B.
[...]	Indicates a set of characters. A ^ as the first character indicates a complementing set.
(...)	Matches the regular expression inside the parentheses. The contents can be retrieved or matched later in the string.

Table 5.2: Standard regular expression metacharacters in Python.

Repetition

The remaining metacharacters are for matching a specified number of characters. This allows a single regular expression to match strings of varying lengths.

Character	Description
*	Matches 0 or more repetitions of the preceding regular expression.
+	Matches 1 or more repetitions of the preceding regular expression.
?	Matches 0 or 1 of the preceding regular expression.
{m,n}	Matches from m to n repetitions of the preceding regular expression.
*?, +?, ??, {m,n}?	Non-greedy versions of the previous four special characters.

Table 5.3: Repetition metacharacters for regular expressions in Python.

```
# Match 0 or more 'a' characters, ending in a 'b'.
>>> pattern = re.compile(r"^\w*a*b$")
>>> for test in ["b", "ab", "aaaaaaaaab", "aba"]:
...     print(test + ':', bool(pattern.search(test)))
...
b: True                                     # 0 'a' characters, then 1 'b'.
ab: True
aaaaaaaaab: True                           # Several 'a' characters, then 1 'b'.
aba: False                                  # 'b' must be the last character.

# Match an 'h' followed by at least one 'i' or 'a' characters.
>>> pattern = re.compile(r"^\w*h[ia]+\w$")
>>> for test in ["ha", "hii", "hiaiaa", "h", "hah"]:
...     print(test + ':', bool(pattern.search(test)))
...
ha: True
hii: True
hiaiaa: True                                # [ia] matches 'i' or 'a'.
h: False                                      # Need at least one 'i' or 'a'
hah: False                                    # 'i' or 'a' must be the last character.

# Match an 'a' followed by 'b' followed by 0 or 1 'c' characters.
>>> pattern = re.compile(r"^\w*abc?\w$")
>>> for test in ["ab", "abc", "abcc", "ac"]:
...     print(test + ':', bool(pattern.search(test)))
...
ab: True
abc: True
abcc: False                                   # Only up to one 'c' is allowed.
ac: False                                     # Missing the 'b'.
```

Each of the repetition operators acts on the expression immediately preceding it. This could be a single character, a group, or a character class. For instance, `(abc)+` matches `abc`, `abcabc`, `abcabcabc`, and so on, but not `aba` or `cba`. On the other hand, `[abc]*` matches any sequence of `a`, `b`, and `c`, including `abcabc` and `aabbcc`.

The curly braces {} specify a custom number of repetitions allowed. {,n} matches **up to** n instances, {m,} matches **at least** m instances, {k} matches **exactly** k instances, and {m,n} matches from m to n instances. Thus the ? operator is equivalent to {,1} and + is equivalent to {1,}.

```
# Match exactly 3 'a' characters.
>>> pattern = re.compile(r"^\w{3}\$")
>>> for test in ["aa", "aaa", "aaaa", "aba"]:
...     print(test + ':', bool(pattern.search(test)))
...
aa: False                                # Too few.
aaa: True
aaaa: False                               # Too many.
aba: False
```

ACHTUNG!

Line anchors are especially important when using repetition operators. Consider the following (bad) example and compare it to the previous example.

```
# Match exactly 3 'a' characters, hopefully.
>>> pattern = re.compile(r"\w{3}")
>>> for test in ["aaa", "aaaa", "aaaaaa", "aaaab"]:
...     print(test + ':', bool(pattern.search(test)))
...
aaa: True
aaaa: True                                # Should be too many!
aaaaaa: True                               # Should be too many!
aaaab: True                                # Too many, and even with the 'b'?
```

The unexpected matches occur because "aaa" is at the beginning of each of the test strings. With the line anchors ^ and \$, the search truly only matches the exact string "aaa".

Problem 4. A *valid Python identifier* (a valid variable name) is any string starting with an alphabetic character or an underscore, followed by any (possibly empty) sequence of alphanumeric characters and underscores.

Define a function that compiles and returns a regular expression pattern object that matches any valid Python identifier.

(Hint: Use the \w character class shortcut to keep your regular expression clean.)

Check your regular expression against the following words. These test cases are a good start, but are not exhaustive.

Matches:	"Mouse"	"compile"	"_123456789"	"__x__"	"while"
Non-matches:	"3rats"	"err*r"	"sq(x)"	"sleep()"	" x"

Manipulating Text with Regular Expressions

So far we have been solely concerned with whether or not a regular expression and a string match, but the power of regular expressions comes with what can be done with a match. In addition to the `search()` method, regular expression pattern objects have the following useful methods.

Method	Description
<code>match()</code>	Match a regular expression pattern to the beginning of a string.
<code>fullmatch()</code>	Match a regular expression pattern to all of a string.
<code>search()</code>	Search a string for the presence of a pattern.
<code>sub()</code>	Substitute occurrences of a pattern found in a string.
<code>subn()</code>	Same as <code>sub</code> , but also return the number of substitutions made.
<code>split()</code>	Split a string by the occurrences of a pattern.
<code>findall()</code>	Find all occurrences of a pattern in a string.
<code>finditer()</code>	Return an iterator yielding a match object for each match.

Table 5.4: Methods of regular expression pattern objects.

```
# Find words that start with 'cat'.
>>> expr = re.compile(r"\bcat\w*") # \b is the shortcut for a word boundary.

>>> target = "Let's catch some catfish for the cat"
>>> bool(expr.search(target))      # Check to see if there is a match.
True

>>> expr.findall(target)         # Get all matching substrings.
['catch' 'catfish', 'cat']

>>> expr.sub("DOG", target)     # Substitute 'DOG' for the matches.
"Let's DOG some DOG for the DOG"

>>> expr.split(target)          # Split the target by the matches.
["Let's ", ' some ', ' for the ', '']
```

Some substitutions require remembering part of the text that the regular expression matches. Groups are useful here: each group in the regular expression can be represented in the substitution string by `\n`, where n is an integer (starting at 1) specifying which group to use.

```
# Find words that start with 'cat', remembering what comes after the 'cat'.
>>> pig_latin = re.compile(r"\bcat(\w*)")
>>> target = "Let's catch some catfish for the cat"

>>> pig_latin.sub(r"at\1clay", target) # \1 = (\w*) from the expression.
"Let's atchclay some atfishclay for the atclay"
```

NOTE

The repetition operators ?, +, *, and {m,n} are *greedy*, meaning that they match the largest string possible. On the other hand, the operators ??, +?, *?, and {m,n}? are *non-greedy*, meaning they match the smallest strings possible. This is very often the desired behavior for a regular expression.

```
>>> target = "<abc> <def> <ghi>"  
  
# Match angle brackets and anything in between.  
>>> greedy = re.compile(r"^.+>$") # Greedy *  
>>> greedy.findall(target)  
['<abc> <def> <ghi>'] # The entire string matched!  
  
# Try again, using the non-greedy version.  
>>> nongreedy = re.compile(r"^.+?>") # Non-greedy *?  
>>> nongreedy.findall(target)  
['<abc>', '<def>', '<ghi>'] # Each <> set is an individual match.
```

Finally, there are a few customizations that make searching larger texts manageable. Each of these *flags* can be used as keyword arguments to `re.compile()`.

Flag	Description
<code>re.DOTALL</code>	. matches any character at all, including the newline.
<code>re.IGNORECASE</code>	Perform case-insensitive matching.
<code>re.MULTILINE</code>	^ matches the beginning of lines (after a newline) as well as the string; \$ matches the end of lines (before a newline) as well as the end of the string.

Table 5.5: Regular expression flags.

```
# Match any line with 3 consecutive 'a' characters somewhere.  
>>> pattern = re.compile("^.+a{3}.*$", re.MULTILINE) # Search each line.  
>>> pattern.findall("""  
This is aaan example.  
This is not an example.  
Actually, it's an example, but it doesn't match.  
This example does maaatch though."""")  
['This is aaan example.', 'This example does maaatch though.'][  
  
# Match anything instance of 'cat', ignoring case.  
>>> catfinder = re.compile("cat", re.IGNORECASE)  
>>> catfinder.findall("cat CAT cAt TAC ctacATT")  
['cat', 'CAT', 'cAt', 'cAT']
```

Problem 5. A Python *block* is composed of several lines of code with the same indentation level. Blocks are delimited by key words and expressions, followed by a colon. Possible key words are `if`, `elif`, `else`, `for`, `while`, `try`, `except`, `finally`, `with`, `def`, and `class`. Some of these keywords require an expression of some sort to follow before the colon (`if`, `elif`, `for`, etc.), some require no expressions to follow before the colon (`else`, `finally`), and `except` may or may not have an expression following before the colon.

Write a function that accepts a string of Python code and uses regular expressions to place colons in the appropriate spots. You may assume that every colon is missing in the input string. See the following for an example.

```
"""
k, i, p = 999, 1, 0
while k > i
    i *= 2
    p += 1
    if k != 999
        print("k should not have changed")
    else
        pass
print(p)
"""

# The string given above should become this string.
"""
k, i, p = 999, 1, 0
while k > i:
    i *= 2
    p += 1
    if k != 999:
        print("k should not have changed")
    else:
        pass
print(p)
"""
```

Problem 6. The file `fake_contacts.txt` contains poorly formatted contact data for 2000 fictitious individuals. Each line of the file contains data for one person, including their name and possibly their birthday, email address, and/or phone number. The formatting of the data is not consistent, and much of it is missing. For example, not everyone has their birthday listed, and those who do may have it listed in the form 1/1/11, 1/01/2011, or some other format.

Use regular expressions to parse the data and format it uniformly, writing birthdays as mm/dd/yyyy and phone numbers as (xxx)xxx-xxxx. Return a dictionary where the key is the name of an individual and the value is another dictionary containing their information. Each of these inner dictionaries should have the keys "birthday", "email", and "phone". In the case of missing data, map the key to `None`. The first two entries of the completed dictionary are given below.

```
{  
    "John Doe": {  
        "birthday": "01/01/1990",  
        "email": "john_doe90@hopefullynotarealaddress.com",  
        "phone": "(123)456-7890"  
    },  
    "Jane Smith": {  
        "birthday": None,  
        "email": None,  
        "phone": "(222)111-3333"  
    },  
    # ...  
}
```

Additional Material

Regular Expressions in the Unix Shell

As we have seen thusfar, regular expressions are very useful when we want to match patterns. Regular expressions can be used when matching patterns in the Unix Shell. Though there are many Unix commands that take advantage of regular expressions, we will focus on `grep` and `awk`.

Regular Expressions and grep

Recall from Lab 1 that `grep` is used to match patterns in files or output. It turns out we can use regular expressions to define the pattern we wish to match.

In general, we use the following syntax:

```
$ grep 'regexp' filename
```

We can also use regular expressions when piping output to `grep`.

```
# List details of directories within current directory.
$ ls -l | grep ^d
```

Regular Expressions and awk

As in Lab 2, we will be using `awk` to format output. By incorporating regular expressions, `awk` becomes much more robust. Before GUI spreadsheet programs like Microsoft Excel, `awk` was commonly used to visualize and query data from a file.

Including `if` statements inside `awk` commands gives us the ability to perform actions on lines that match a given pattern. The following example prints the filenames of all files that are owned by `freddy`.

```
$ ls -l | awk ' {if ($3 ~ /freddy/) print $9} '
```

Because there is a lot going on in this command, we will break it down piece-by-piece. The output of `ls -l` is getting piped to `awk`. Then we have an `if` statement. The syntax here means if the condition inside the parenthesis holds, print field 9 (the field with the filename). The condition is where we use regular expressions. The `~` checks to see if the contents of field 3 (the field with the username) matches the regular expression found inside the forward slashes. To clarify, `freddy` is the regular expression in this example and the expression must be surrounded by forward slashes.

Consider a similar example. In this example, we will list the names of the directories inside the current directory. (This replicates the behavior of the Unix command `ls -d */`)

```
$ ls -l | awk ' {if ($1 ~ /^d/) print $9} '
```

Notice in this example, we printed the names of the directories, whereas in one of the example using `grep`, we printed all the details of the directories as well.

ACHTUNG!

Some of the definitions for character classes we used earlier in this lab will not work in the Unix Shell. For example, \w and \d are not defined. Instead of \w, use [:alnum:]]. Instead of \d, use [:digit:]]. For a complete list of similar character classes, search the internet for *POSIX Character Classes* or *Bracket Character Classes*.

Problem 7. You have been given a list of transactions from a fictional start-up company. In the `transactions.txt` file, each line represents a transaction. Transactions are represented as follows:

```
# Notice the semicolons delimiting the fields. Also, notice that in ←
    between the last and first name, that is a comma, not a semicolon.
<ORDER_ID>;<YEAR><MONTH><DAY>;<LAST>,<FIRST>;<ITEM_ID>
```

Using this set of transactions, produce the following information using regular expressions and the given command:

- Using `grep`, print all transactions by either Nicholas Ross or Zoey Ross.
- Using `awk`, print a sorted list of the names of individuals that bought item 3298.
- Using `awk`, print a sorted list of items purchased between June 13 and June 15 of 2014 (inclusive).

These queries can be produced using one command each.

We encourage the interested reader to research more about how regular expressions can be used with `sed`.

6

Web Technologies

Lab Objective: *The Internet is an umbrella term for the collective grouping of all publicly accessible computer networks in the world. This collective network can be traversed to access services such as social communication, maps, video streaming, and accessibility to large datasets, all of which are hosted on computers across the world. Using these technologies requires an understanding of data serialization, data transportation protocols, and how programs such as servers, clients, and APIs are created to facilitate this communication.*

Data Serialization

Serialization is the process of packaging data in a form that makes it easy to transmit the data and quickly reconstruct it on another computer or in a different programming language. One of the most prevalent serialization metalanguages is *JSON*, which stands for *JavaScript Object Notation*.¹ It stores information about objects as a specially formatted string that is easy for both humans and machines to read and write. *Deserialization* is the process of reconstructing an object from the string.

JSON is built on two types of data structures: a collection of key/value pairs and an ordered list of values, similar to Python's built-in `dict` and `list` structures, respectively.

```
{                                     # A family's info written in JSON format.  
  "lastname": "Smith",           # The outer dictionary has two keys:  
  "children": [                  # "lastname" and "children".  
    {                           # The "children" key maps to a list of  
      "name": "Timmy",          # two dictionaries, one for each of the  
      "age": 8                 # two children.  
    },  
    {  
      "name": "Missy",  
      "age": 5  
    }  
  ]  
}
```

¹Despite having “JavaScript” in its name, JSON is a language-independent format. In fact, JSON is frequently used for transmitting data between different programming languages.

NOTE

To see a longer example of what JSON looks like, try opening a Jupyter Notebook (a `.ipynb` file) in a plain text editor. The file lists the Notebook cells, each of which has attributes like `"cell_type"` (usually code or markdown) and `"source"` (the actual code in the cell).

The JSON libraries of various languages have a fairly standard interface. The Python standard library module for JSON is called `json`; if performance speed is critical, consider using the `ujson` or `simplejson` modules that are written in C.

A string written in JSON format that represents a piece of data is called a *JSON message*. To generate the JSON message for a single Python object, use `json.dumps()`. Alternatively, the function `json.dump()` generates the JSON message of an object and writes it to an open file. To load a JSON string or file, use `json.loads()` or `json.load()`, respectively.

```
>>> import json

# Store info about a car in a nested dictionary.
>>> my_car = {
...     "car": {
...         "make": "Ford",
...         "color": [255, 30, 30]  },
...     "owner": "me" }

# Get the JSON message corresponding to my_car.
>>> car_str = json.dumps(my_car)
>>> car_str
'{"car": {"make": "Ford", "color": [255, 30, 30]}, "owner": "me"}'

# Load the JSON message into a Python object, reconstructing my_car.
>>> car_object = json.loads(car_str)
>>> for key in car_object:          # The loaded object is a dictionary.
...     print(key + ':', car_object[key])
...
car: {'make': 'Ford', 'color': [255, 30, 30]}
owner: me

# Write the car info to an external file.
>>> with open("my_car.json", 'w') as outfile:
...     json.dump(my_car, outfile)
...

# Read the file to check that it saved correctly.
>>> with open("my_car.json", 'r') as infile:
...     new_car = json.load(infile)
...
>>> print(new_car.keys())           # This loaded object is also a dictionary.
dict_keys(['car', 'owner'])
```

Problem 1. The file `nyc_traffic.json` contains information about 1000 traffic accidents in New York City during the summer of 2017.^a Each entry lists one or more reasons for the accident, such as “Unsafe Speed” or “Fell Asleep.”

Write a function that loads the data from the JSON file. Look at the first few entries of the dataset and decide how to gather information about the cause(s) of each accident. Make a readable, sorted bar chart showing the total number of times that each of the 7 most common reasons for accidents are listed in the data set.

(Hint: the `collections.Counter` data structure may be useful here.)

To check your work, the 6th most common reason is “Backing Unsafely,” listed 59 times.

^aSee <https://opendata.cityofnewyork.us/>.

Custom Encoders and Decoders for JSON

The default JSON encoder and decoder do not support serialization for every kind of data structure. For example, a `set` cannot be serialized using only `json` functions. However, the default JSON encoder can be subclassed to handle sets or custom data structures. A custom encoder must organize the information in an object as nested lists and dictionaries. The corresponding custom decoder uses the way that the encoder organizes the information to reconstruct the original object.

For example, one way to serialize a `set` is to express it as a dictionary with one key that indicates its data type, and another key mapping to the actual data.

```
>>> class SetEncoder(json.JSONEncoder):
...     """A custom JSON encoder for Python sets."""
...     def default(self, obj):
...         if not isinstance(obj, set):
...             raise TypeError("expected a set for encoding")
...         return {"dtype": "set", "data": list(obj)}
...
# Use the custom encoder to convert a set to its custom JSON message.
>>> set_message = json.dumps(set('abca'), cls=SetEncoder)
>>> set_message
'{"dtype": "set", "data": ["a", "b", "c"]}'
...
# Define a custom decoder for JSON messages generated by the SetEncoder.
>>> def set_decoder(item):
...     if "dtype" in item:
...         if item["dtype"] != "set" or "data" not in item:
...             raise ValueError("expected a JSON message from SetEncoder")
...         return set(item["data"])
...     raise ValueError("expected a JSON message from SetEncoder")
...
# Use the custom decoder to convert a JSON message to the original object.
>>> json.loads(set_message, object_hook=set_decoder)
{'a', 'b', 'c'}
```

Checks for errors like in the previous example are good practice to ensure that custom encoders and decoders are only used when intended.

Problem 2. The following class facilitates a regular 3×3 game of tic-tac-toe, where the boxes in the board have the following coordinates.

(0, 0)	(0, 1)	(0, 2)
(1, 0)	(1, 1)	(1, 2)
(2, 0)	(2, 1)	(2, 2)

```
class TicTacToe:
    def __init__(self):
        """Initialize an empty board. The 0's go first."""
        self.board = [[' ']*3 for _ in range(3)]
        self.turn, self.winner = "0", None

    def move(self, i, j):
        """Mark an O or X in the (i,j)th box and check for a winner."""
        if self.winner is not None:
            raise ValueError("the game is over!")
        elif self.board[i][j] != ' ':
            raise ValueError("space ({}, {}) already taken".format(i, j))
        self.board[i][j] = self.turn

        # Determine if the game is over.
        b = self.board
        if any(sum(s == self.turn for s in r) == 3 for r in b):
            self.winner = self.turn      # 3 in a row.
        elif any(sum(r[i] == self.turn for r in b) == 3 for i in range(3)):
            self.winner = self.turn      # 3 in a column.
        elif b[0][0] == b[1][1] == b[2][2] == self.turn:
            self.winner = self.turn      # 3 in a diagonal.
        elif b[0][2] == b[1][1] == b[2][0] == self.turn:
            self.winner = self.turn      # 3 in a diagonal.
        else:
            self.turn = "0" if self.turn == "X" else "X"

    def empty_spaces(self):
        """Return the list of coordinates for the empty boxes."""
        return [(i, j) for i in range(3) for j in range(3)
                if self.board[i][j] == ' ']

    def __str__(self):
        return "\n-----\n".join(" | ".join(r) for r in self.board)
```

Write a custom encoder and decoder for the `TicTacToe` class. If the custom encoder receives anything other than a `TicTacToe` object, raise a `TypeError`.

NOTE

JSON is a good option for transferring data between two different languages. Python's `pickle` module is particularly good for serialization when the stored object will only be unpacked in Python. The main functions are almost identical to `json.dumps()` and its companions.

```
>>> import pickle

>>> item = pickle.dumps([1, 2, 3, 4, 5, 6])
>>> item
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05K\x06e.'

>>> pickle.loads(item)
[1, 2, 3, 4, 5, 6]
```

In addition, there are many other serialization formats such as YAML and XML. However, JSON is the dominant format for serialization in web applications.

Servers and Clients

The Internet is like a network of roads connecting the buildings of a city where each building represents a computer and each road represents the physical wires or wireless pathways that allow for intercommunication. Navigating the road properly, requires using the correct kinds of vehicles and following the established laws for road travel. There are also various kinds of vehicles for different purposes and with different capabilities that are used to transport items from building to building. In a similar fashion, the Internet has specific protocols that allow for standardized communication within and between computers.

The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these is *Transmission Control Protocol* (TCP), used to establish a connection between two computers, exchange bits of information called *packets*, and then close the connection. More specifically, TCP creates network *socket* objects that are used to send and receive data packets from a computer. A socket is basically an address within a computer on which a program can send or receive data, like a P.O. box within a post office. The post office is the computer that receives mail, but the mail is distributed to individual programs that check the P.O. box for their personal communications.

Creating a Server

A *server* is a program that interacts with and provides functionality to *client* programs. A client program contacts a server to receive some sort of response that assists it in fulfilling its function. Servers are fundamental to modern networks and provide services such as file sharing, authentication, webpage information, databases, etc.

One simple way to create a server in Python is via the `socket` module. The server socket must first be initialized by specifying the type of connection and the address that clients can find the server at. The server socket then listens and waits for a connection from a client, receives and processes data, then eventually sends a response back to the client. After exchanges between the server and the client are finished, the server closes the connection to the client.

```

def mirror_server(server_address=("0.0.0.0", 33333)):
    """A server for reflecting strings back to clients in reverse order."""
    print("Starting mirror server on {}".format(server_address))

    # Specify the socket type, which determines how clients will connect.
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.bind(server_address)      # Assign this socket to an address.
    server_sock.listen(1)                # Start listening for clients.

    while True:
        # Wait for a client to connect to the server.
        print("\nWaiting for a connection...")
        connection, client_address = server_sock.accept()
        try:
            # Receive data from the client.
            print("Connection accepted from {}".format(client_address))
            in_data = connection.recv(1024).decode()      # Receive data.
            print("Received '{}' from client".format(in_data))

            # Process the received data and send something back to the client.
            out_data = in_data[::-1]
            print("Sending '{}' back to the client".format(out_data))
            connection.sendall(out_data.encode())         # Send data.

        finally:      # Make sure the connection is closed securely.
            connection.close()
            print("Closing connection from {}".format(client_address))

```

The two parameters for `socket.socket()` specify the socket type.² The server address is a tuple (host, port). The host is the IP address, which in this case is `"localhost"` or `"0.0.0.0"`—the default address that specifies the local machine and allows connections on all interfaces. The port number is an integer from 0 to 65535. About 250 port numbers are commonly used, and certain ports have pre-defined uses.

ACHTUNG!

Only use port numbers greater than 1023 to avoid interrupting standard system services.

After setting up the server socket, it waits for a client to connect. The `accept()` method returns a new socket object (`connection`) and the client's address. Data is received through the connection socket's `recv()` method, which takes an integer specifying the number of bits of data to receive. The data is transferred as a raw byte stream (of type `bytes`), so the `decode()` method is necessary to translate the data into a string. Likewise, data that is sent back to the client through the connection socket's `sendall()` method must be encoded into a byte stream via the `encode()` method.

Finally, the `try-finally` blocks ensure that the connection is always closed securely. To stop a server, raise a `KeyboardInterrupt` (press `ctrl+c`) in the terminal where it is running.

²See <https://docs.python.org/3/library/socket.html> for details on these parameters.

NOTE

When running `mirror_server()`, the program hangs on the following line.

```
connection, client_address = server_sock.accept()
```

This is because the `accept()` method does not return until a connection is made with a client. Therefore, this server program cannot be executed in its entirety without a client. Client creation is addressed in the next section.

ACHTUNG!

It often takes some time for a computer to reopen a port after closing a server connection. This is due to the timeout functionality of specific protocols that check connections for errors and disruptions. While testing code, wait a few seconds before running the program again, or use different ports for each test.

Problem 3. Write a function that accepts a (host, port) tuple and starts up a tic-tac-toe server at the specified location. Wait to accept a connection, then while the connection is open, repeat the following operations.

1. Receive a JSON serialized `TicTacToe` object (serialized with your custom encoder from Problem 2) from the client.
2. Deserialize the `TicTacToe` object using your custom decoder from Problem 2.
3. If the client has just won the game, send "`WIN`" back to the client and close the connection.
4. If there is no winner but board is full, send "`DRAW`" to the client and close the connection.
5. If the game still isn't over, make a random move on the tic-tac-toe board and serialize the updated `TicTacToe` object. If this move wins the game, send "`LOSE`" to the client, then send the serialized object separately (as proof), and close the connection. Otherwise, send only the updated `TicTacToe` object back to the client but keep the connection open.

(Hint: print information at each step so you can see what the server is doing.)

Ensure that the connection closes securely even if an exception is raised. Note that you will not be able to fully test your server until you have written a client (see Problem 4).

Creating a Client

The `socket` module also has tools for writing client programs. First, create a socket object with the same settings as the server socket, then call the `connect()` method with the server address as a parameter. Once the client socket is connected to the server socket, the two sockets can transfer information between themselves.

```

def mirror_client(server_address=("0.0.0.0", 33333)):
    """A client program for mirror_server()."""
    print("Attempting to connect to server at {}...".format(server_address))

    # Set up the socket to be the same type as the server.
    client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_sock.connect(server_address)      # Attempt to connect to the server.

    # Send some data from the client user to the server.
    out_data = input("Type a message to send to the server: ")
    client_sock.sendall(out_data.encode())           # Send data.

    # Wait to receive a response back from the server.
    in_data = client_sock.recv(1024).decode()          # Receive data.
    print("Received '{}' from the server".format(in_data))

    # Close the client socket.
    client_sock.close()

```

Note that unlike the server socket, the client socket sends and reads the data itself instead of creating a new connection socket. When the client program is complete, close the client socket. The server will keep running, waiting for another client to serve.

To see a client and server communicate, open a terminal and run the server, then run the client in a separate terminal.

```

# TERMINAL 1
>>> mirror_server()                      # First start up the server.
Starting mirror server on ('0.0.0.0', 33333)

Waiting for a connection...      # At this point, start the client.
Connection accepted from ('127.0.0.1', 50679).
Received 'racecars and lollipops' from client
Sending 'spopillol dna sracecar' back to the client
Closing connection from ('127.0.0.1', 50679)

Waiting for a connection...
# The client program is over, but the server waits to keep serving clients.

```

```

# TERMINAL 2
>>> mirror_client()                      # Start the client after the server.
Attempting to connect to server at ('0.0.0.0', 33333)...
Connected!
Type a message to send: racecars and lollipops
Received 'spopillol dna sracecar' from the server

```

Problem 4. Write a function that accepts a (host, port) tuple and connects to the tic-tac-toe server at the specified location. Start by initializing a new `TicTacToe` object, then repeat the following steps until the game is over.

1. Print the board and prompt the player for a move. Continue prompting the player until they provide valid input.
2. Update the board with the player's move, then serialize it using your custom encoder from Problem 2, and send the serialized version to the server.
3. Receive a response from the server. If the game is over, congratulate or mock the player appropriately. If the player lost, receive a second response from the server (the final game board), deserialize it, and print it out.

Close the connection once the game ends.

APIs

An *Application Program Interface* (API) is a particular kind of server that listens for requests from authorized users and responds with data. For example, a list of twenty different locations can be sent with the proper request syntax to a Google Maps API, and it will respond with the calculated driving time from each location to every other. Every API has *endpoints* where clients send their requests. Though standards exist for creating and communicating with APIs, most APIs have a unique syntax for authentication and requests that is documented by the organization providing the service.

ACHTUNG!

Each website and API has a policy that specifies appropriate behavior for automated data retrieval and usage. If data is requested without complying with these requirements, there can be severe legal consequences. Most websites detail their policies in a file called `robots.txt` on their main page. See, for example, <https://www.google.com/robots.txt>.

Problem 5. The `requests` module is the standard way to send a download request to an API.

```
>>> import requests
>>> requests.get(endpoint).json()      # Download and extract the data.
```

Write a function that makes requests to download data from the following API endpoints managed by New York City.

- Recycling bin locations: <https://data.cityofnewyork.us/api/views/sxx4-xhzg/rows.json?accessType=DOWNLOAD>
- Residential addresses: <https://data.cityofnewyork.us/api/views/7823-25a9/rows.json?accessType=DOWNLOAD>

Save the recycling data as `nyc_recycling.json` and the address data as `nyc_addresses.json`.

Problem 6. Write a function that loads the data files generated in Problem 5 but **does not** call the actual function from Problem 5. Determine how close the residential addresses in New York City are to the nearest recycling bin.

1. Retrieve the latitude and longitude of each recycling bin and, separately, the latitude and longitude of each residential address (ignore entries without these coordinates). Note carefully that the coordinates for the recycling data are in (latitude, longitude) format, but the coordinates for the address data are in (longitude, latitude) format.
(Hint: Both datasets are, at the highest level, dictionaries with two keys: "`meta`", which has information about the data; and "`data`", which has the actual data. All of the information needed is contained in the "`data`" key value.)
2. Load the recycling bin data into a k-d tree.
3. For each address, query the tree to find the distance to the nearest recycling bin, in terms of the coordinates.
4. Plot a histogram of the distances.

For steps 2–3, recall the following syntax for using a k-d tree in SciPy.

```
from scipy.spatial import KDTree

tree = KDTree(data)                      # Initialize the tree.
min_distance, index = tree.query(target)  # Query for a point.
```

7

MongoDB

Lab Objective: *Relational databases, including those managed with SQL or pandas, require data to be organized into tables. However, many data sets have an inherently dynamic structure that cannot be efficiently represented as tables. MongoDB is a non-relational database management system that is well-suited to large, fast-changing datasets. In this lab we introduce the Python interface to MongoDB, including common commands and practices.*

Database Initialization

Suppose the manager of a general store has all sorts of inventory: food, clothing, tools, toys, etc. There are some common attributes shared by all items: name, price, and producer. However, other attributes are unique to certain items: sale price, number of wheels, or whether or not the product is gluten-free. A relational database housing this data would be full of mostly-blank rows, which is extremely inefficient. In addition, adding new items to the inventory requires adding new columns, causing the size of the database to rapidly increase. To efficiently store the data, the whole database would have to be restructured and rebuilt often.

To avoid this problem, NoSQL databases like MongoDB avoid using relational tables. Instead, each item is a JSON-like object, and thus can contain whatever attributes are relevant to the specific item, without including any meaningless attribute columns.

NOTE

MongoDB is a database management system (DBMS) that runs on a server, which should be running in its own dedicated terminal. Refer to the Additional Material section for installation instructions.

The Python interface to MongoDB is called `pymongo`. After installing `pymongo` and with the MongoDB server running, use the following code to connect to the server.

```
>>> from pymongo import MongoClient  
# Create an instance of a client connected to a database running  
# at the default host IP and port of the MongoDB service on your machine.  
>>> client = MongoClient()
```

Creating Collections and Documents

A MongoDB database stores *collections*, and a collection stores *documents*. The syntax for creating databases and collections is a little unorthodox, as it is done through attributes instead of methods.

```
# Create a new database.  
>>> db = client.db1  
  
# Create a new collection in the db database.  
>>> col = db.collection1
```

Documents in MongoDB are represented as JSON-like objects, and therefore do not adhere to a set schema. Each document can have its own *fields*, which are completely independent of the fields in other documents.

```
# Insert one document with fields 'name' and 'age' into the collection.  
>>> col.insert_one({'name': 'Jack', 'age': 23})  
  
# Insert another document. Notice that the value of a field can be a string,  
# integer, truth value, or even an array.  
>>> col.insert_one({'name': 'Jack', 'age': 22, 'student': True,  
...                 'classes': ['Math', 'Geography', 'English']})  
  
# Insert many documents simultaneously into the collection.  
>>> col.insert_many([  
...     {'name': 'Jill', 'age': 24, 'student': False},  
...     {'name': 'John', 'nickname': 'Johnny Boy', 'soldier': True},  
...     {'name': 'Jeremy', 'student': True, 'occupation': 'waiter'} ])
```

NOTE

Once information has been added to the database it will remain there, even if the python environment you are working with is shut down. It can be reaccessed anytime using the same commands as before.

```
>>> client = MongoClient()  
>>> db = client.db1  
>>> col = db.collection1
```

To delete a collection, use the database's `drop_collection()` method. To delete a database, use the client's `drop_database()` method.

Problem 1. The file `trump.json` contains posts from <http://www.twitter.com> (tweets) over the course of an hour that have the key word “trump”.^a Each line in the file is a single JSON message that can be loaded with `json.loads()`.

Create a MongoDB database and initialize a collection in the database. Use the collection’s `delete_many()` method with an empty set as input to clear existing contents of the collection, then fill the collection one line at a time with the data from `trump.json`. Check that your collection has 95,643 entries with its `count()` method.

^aSee the Additional Materials section for an example of using the Twitter API.

Querying a Collection

MongoDB uses a *query by example* pattern for querying. This means that to query a database, an example must be provided for the database to use in matching other documents.

```
# Find all the documents that have a 'name' field containing the value 'Jack'.
>>> data = col.find({'name': 'Jack'})

# Find the FIRST document with a 'name' field containing the value 'Jack'.
>>> data = col.find_one({'name': 'Jack'})
```

The `find_one()` method returns the first matching document as a dictionary. The `find()` query may find any number of objects, so it will return a `Cursor`, a Python object that is used to iterate over the query results. There are many useful functions that can be called on a `Cursor`, for more information see <http://api.mongodb.com/python/current/api/pymongo/cursor.html>.

```
# Search for documents containing True in the 'student' field.
>>> students = col.find({'student': True})
>>> students.count()                      # There are 2 matching documents.
2

# List the first student's data.
# Notice that each document is automatically assigned an ID number as '_id'.
>>> students[0]
{'_id': ObjectId('59260028617410748cc7b8c7'),
 'age': 22,
 'classes': ['Math', 'Geography', 'English'],
 'name': 'Jack',
 'student': True}

# Get the age of the first student.
>>> students[0]['age']
22

# List the data for every student.
>>> list(students)
[{'_id': ObjectId('59260028617410748cc7b8c7'),
```

```
'age': 22,
'classes': ['Math', 'Geography', 'English'],
'name': 'Jack',
'student': True},
{'_id': ObjectId('59260028617410748cc7b8ca'),
'name': 'Jeremy',
'occupation': 'waiter',
'student': True}]
```

The Logical operators listed in the following table can be used to do more complex queries.

Operator	Description
\$lt, \$gt	<, >
\$lte,\$gte	<=, >=
\$eq, \$ne	==, !=
\$in, \$nin	in, not in
\$or, \$and, \$not	or, and, not
\$exists	Match documents with a specific field
\$type	Match documents with values of a specific type
\$all	Match arrays that contain all queried elements
\$size	Match arrays with a specified number of elements
\$regex	Search documents with a regular expression

Table 7.1: MongoDB Query Operators

```
# Query for everyone that is either above the age of 23 or a soldier.
>>> results = col.find({'$or':[{'age':{'$gt': 23}},{'soldier': True}]})

# Query for everyone that is a student (those that have a 'student' attribute
# and haven't been expelled).
>>> results = col.find({'student': {'$not': {'$in': [False, 'Expelled']}}})

# Query for everyone that has a student attribute.
>>> results = col.find({'student': {'$exists': True}})

# Query for people whose name contains a the letter 'e'.
>>> import re
>>> results = col.find({'name': {'$regex': re.compile('e')}})
```

It is likely that a database will hold more complex JSON entries than these, with many nested attributes and arrays. For example, an entry in a database for a school might look like this.

```
{'name': 'Jason', 'age': 16,
'student': {'year':'senior', 'grades': ['A', 'C', 'A', 'B'], 'flunking': False},
'jobs':['waiter', 'custodian']}
```

To query the nested attributes and arrays use a dot, as in the following examples.

```
# Query for student that are seniors
>>> results = col.find({'student.year': 'senior'})

# Query for students that have an A in their first class.
>>> results = col.find({'student.grades.0': 'A'})
```

The Twitter JSON files are large and complex. To see what they look like, either look at the JSON file used to populate the `collection` or print any tweet from the database. The following website also contains useful information about the fields in the JSON file <https://dev.twitter.com/overview/api/tweets>.

The `distinct` function is also useful in seeing what the possible values are for a given field.

```
# Find all the values in the names field.
>>> col.distinct("name")
['Jack', 'Jill', 'John', 'Jeremy']
```

Problem 2. Query the Twitter collection from Problem 1 for the following information.

- How many tweets include the word Russia? Use `re.IGNORECASE`.
- How many tweets came from one of the main continental US time zones? These are listed as `"Central Time (US & Canada)"`, `"Pacific Time (US & Canada)"`, `"Eastern Time (US & Canada)"`, and `"Mountain Time (US & Canada)"`.
- How often did each language occur? Construct a dictionary with each language and its frequency count.
(Hint: use `distinct()` to get the language options.)

Deleting and Sorting Documents

Items can be deleted from a database using the same syntax that is used to find them. Use `delete_one` to delete just the first item that matches your search, or `delete_many` to delete all items that match your search.

```
# Delete the first person from the database whose name is Jack.
>>> col.delete_one({'name':'Jack'})

# Delete everyone from the database whose name is Jack.
>>> col.delete_many({'name':'Jack'})

# Clear the entire collection.
>>> col.delete_many({})
```

Another useful function is the `sort` function, which can sort the data by some attribute. It takes in the attribute by which the data will be sorted, and then the direction (1 for ascending and -1 for descending). Ascending is the default. The following code is an example of sorting.

```
# Sort the students by name in alphabetic order.  
>>> results = col.find().sort('name', 1)  
>>> for person in results:  
...     print(person['name'])  
...  
Jack  
Jack  
Jeremy  
Jill  
John  
  
# Sort the students oldest to youngest, ignoring those whose age is not listed.  
>>> results = col.find({'age': {'$exists': True}}).sort('age', -1)  
>>> for person in results:  
...     print(person['name'])  
...  
Jill  
Jack  
Jack
```

Problem 3. Query the Twitter collection from Problem 1 for the following information.

- What are the usernames of the 5 most popular (defined as having the most followers) tweeters? Don't include repeats.
- Of the tweets containing at least 5 hashtags, sort the tweets by how early the 5th hashtag appears in the text. What is the earliest spot (character count) it appears?
- What are the coordinates of the tweet that came from the northernmost location? Use the latitude and longitude point in "coordinates".

Updating Documents

Another useful attribute of MongoDB is that data in the database can be updated. It is possible to change values in existing fields, rename fields, delete fields, or create new fields with new values. This gives much more flexibility than a relational database, in which the structure of the database must stay the same. To update a database, use either `update_one` or `update_many`, depending on whether one or more documents should be changed (the same as with `delete`). Both of these take two parameters; a find query, which finds documents to change, and the update parameters, telling these things what to update. The syntax is `update_many({find query},{update parameters})`.

The update parameters must contain update operators. Each update operator is followed by the field it is changing and the value to change it. The syntax is the same as with query operators. The operators are shown in the table below.

Operator	Description
<code>\$inc</code> , <code>\$mul</code>	<code>+=</code> , <code>*=</code>
<code>\$min</code> , <code>\$max</code>	<code>min()</code> , <code>max()</code>
<code>\$rename</code>	Rename a specified field to the given new name
<code>\$set</code>	Assign a value to a specified field (creating the field if necessary)
<code>\$unset</code>	Remove a specified field
<code>\$currentDate</code>	Set the value of the field to the current date. With <code>"\$type": "date"</code> , use a <code>datetime</code> format; with <code>"\$type": "timestamp"</code> , use a timestamp.

Table 7.2: MongoDB Update Operators

```
# Update the first person from the database whose name is Jack to include a
# new field 'lastModified' containing the current date.
>>> col.update_one({'name': 'Jack'},
...                  {''$currentDate': {'lastModified': {'$type': 'date'}}})

# Increment everyones age by 1, if they already have an age field.
>>> col.update_many({'age': {'$exists': True}}, {'$inc': {'age': 1}})

# Give the first John a new field 'best_friend' that is set to True.
>>> col.update_one({'name': 'John'}, {'$set': {'best_friend': True}})
```

Problem 4. Clean the twitter collection in the following ways.

- Get rid of the `"retweeted_status"` field in each tweet.
- Update every tweet from someone with at least 1000 followers to include a `popular` field whose value is `True`. Report the number of popular tweets.
- (OPTIONAL) The geographical coordinates used before in `coordinates.coordinates` are turned off for most tweets. But many more have a bounding box around the coordinates in the `place` field. Update every tweet without coordinates that contains a bounding box so that the coordinates contains the average value of the points that form the bounding box. Make the structure of `coordinates` the same as the others, so it contains `coordinates` with a longitude, latitude array and a `type`, the value of which should be 'Point'.

(Hint: Iterate through each tweet in with a bounding box but no coordinates. Then for each tweet, grab it's id and the bounding box coordinates. Find the average, and then update the tweet. To update it search for it's id and then give the needed update parameters. First unset coordinates, and then set `coordinates.coordinates` and `coordinates.type` to the needed values.)

Additional Material

Installation of MongoDB

MongoDB runs as an isolated program with a path directed to its database storage. To run a practice MongoDB server on your machine, complete the following steps:

Create Database Directory

To begin, navigate to an appropriate directory on your machine and create a folder called `data`. Within that folder, create another folder called `db`. Make sure that you have read, write, and execute permissions for both folders.

Retrieve Shell Files

To run a server on your machine, you will need the proper executable files from MongoDB. The following instructions are individualized by operating system. For all of them, download your binary files from <https://www.mongodb.com/download-center?jmp=nav#community>.

1. For Linux/Mac:

Extract the necessary files from the downloaded package. In the terminal, navigate into the `bin` directory of the extracted folder. You may then start a Mongo server by running in a terminal: `./mongod --dbpath /path to your data folder`.

2. For Windows:

Go into your Downloads folder and run the Mongo `.msi` file. Follow the installation instructions. You may install the program at any location on your machine, but do not forget where you have installed it. You may then start a Mongo server by running in command prompt: `C:\locationofmongoprogram\mongod.exe --dbpath C:\path to data folder\data\db`.

MongoDB servers are set by default to run at address:port `127.0.0.1:27107` on your machine.

You can also run Mongo commands through a mongo terminal shell. More information on this can be found at <https://docs.mongodb.com/getting-started/shell/introduction/>.

Twitter API

Pulling information from the Twitter API is simple. First you must get a Twitter account and register your app with them on apps.twitter.com. This will enable you to have a consumer key, consumer secret, access token, and access secret, all required by the Twitter API.

You will also need to install `tweepy`, an open source library that allows python to easily work with the Twitter API. This can be installed with pip by running from the command line

```
$pip install tweepy
```

The data for this lab was then pulled using the following code on May 26, 2017.

```
import tweepy  
from tweepy import OAuthHandler  
from tweepy import Stream
```

```

from tweepy.streaming import StreamListener
from pymongo import MongoClient
import json

#Set up the database
client = MongoClient()
mydb = client.db1
twitter = mydb.collection1

f = open('trump.txt','w') #If you want to write to a file

consumer_key = #Your Consumer Key
consumer_secret = #Your Consumer Secret
access_token = #Your Access Token
access_secret = #Your Access Secret

my_auth = OAuthHandler(consumer_key, consumer_secret)
my_auth.set_access_token(access_token, access_secret)

class StreamListener(tweepy.StreamListener):
    def on_status(self, status):
        print(status.text)

    def on_data(self, data):
        try:
            twitter.insert_one(json.loads(data)) #Puts the data into your ←
            MongoDB
            f.write(str(data)) #Writes the data to an output file
            return True
        except BaseException as e:
            print(str(e))
            print("Error")
        return True

    def on_error(self, status):
        print(status)
        if status_code == 420: #This means twitter has blocked us temporarily, ←
            so we want to stop or they will get mad. Wait 30 minutes or so and ←
            try again. Running this code often in a short period of time will ←
            cause twitter to block you. But you can stream tweets for as long ←
            as you want without any problems.
        return False
    else:
        return True

stream_listener = StreamListener()
stream = tweepy.Stream(auth=my_auth, listener=stream_listener)
stream.filter(track=["trump"]) #This pulls all tweets that include the keyword ←
    "trump". Any number of keywords can be searched for.

```



8

Web Scraping I: Introduction to BeautifulSoup

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 BeautifulSoup, Python's canonical tool for efficiently and cleanly navigating and parsing 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 provides a document with structure and is composed of pairs of *tags* to surround and define various types of 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 even be written equivalently 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 ``, the tag for representing an image.

Problem 1. The HTML of a website is easy to view in most browsers. In Google Chrome, go to `http://www.example.com`, right click anywhere on the page that isn't a picture or a link, and select **View Page Source**. This will open the HTML source code that defines the page. Examine the source code. What tags are used? What is the value of the `type` attribute associated with the `style` tag?

Write a function that returns the set of names of tags used in the website, and the value of the `type` attribute of the `style` tag (as a string).
 (Hint: there are ten unique tag names.)

BeautifulSoup

BeautifulSoup (`bs4`) is a package¹ 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.

¹BeautifulSoup is not part of the standard library; install it with `conda install beautifulsoup4` or with `pip install beautifulsoup4`.

```
>>> 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>
"""

>>> 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 8.1: 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. Use your function and the source code from <http://www.example.com> (see `example.html`) to check your answers from Problem 1.

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 to 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 8.2: Navigation attributes of the `bs4.element.Tag` class.

```
>>> print(pig_soup.prettify())
<html>
  <head>                                # <head> is the parent of the <title>
    <title>
      Three Little Pigs
    </title>
  </head>
  <body>                                 # <body> is the sibling of <head>
    <p class="title">                      # and the parent of two <p> tags (title and story).
      <b>
        The Three Little Pigs
      </b>
    </p>
    <p class="story">
      Once upon a time, there were three little pigs named
      <a class="pig" href="http://example.com/larry" id="link1">
        Larry,
      </a>
      <a class="pig" href="http://example.com/mo" id="link2">
        Mo
      </a>
      , and
      <a class="pig" href="http://example.com/curly" id="link3">
        Curly.                               # The preceding <a> tags are siblings with each
                                              # other and the following two <p> tags.
    </p>
    <p>
      The three pigs had an odd fascination with experimental construction.
    </p>
    <p>
      ...
    </p>
  </body>
</html>
```

```
# 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>

# Alternatively, get all siblings past <a> at once.
>>> list(a_tag.next_siblings)
['\n',
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 ', and\n',
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>,
 '\n',
 <p>The three pigs had an odd fascination with experimental construction.</p>,
 '\n',
 <p>...</p>,
 '\n']
```

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.

```
# Get to the <p> tag that has class="story".
>>> 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:
...     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'`.

NOTE

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. The file `example.html` contains the HTML source for `http://www.example.com`. Write a function that reads the file and loads the code into BeautifulSoup. Find the only `<a>` tag with a hyperlink and return its text.

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,
>>> 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.^a. 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.”
3. The tag which contains the number associated with the Actual Max Temperature.

This HTML tree is significantly larger than the previous examples. To get started, consider opening the file in a web browser. Find the element that you are searching for on the page, right click it, and select `Inspect`. This opens the HTML source at the element that the mouse clicked on.

^aSee 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

Consider the problem of finding the tag that is a link 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 compile 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>]

# Final 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']
```

Problem 5. The file `large_banks_index.html` is an index of data about large banks, as recorded by the Federal Reserve.^a Write a function that reads the file and loads the source into BeautifulSoup. Return a list of the tags containing the links to bank data from September 30, 2003 to December 31, 2014, where the dates are in reverse chronological order.

^aSee <https://www.federalreserve.gov/releases/lbr/>.

Problem 6. The file `large_banks_data.html` is one of the pages from the index in Problem 5.^a Write a function that reads the file and loads the source into BeautifulSoup. 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.

In the case of a tie, sort the banks alphabetically by name.

^aSee <http://www.federalreserve.gov/releases/lbr/20030930/default.htm>.

9

Web Scraping II: Advanced Web Scraping Techniques

Lab Objective: *Gathering data from the internet often requires information from several web pages. In this lab, we present two methods for crawling through multiple web pages without violating copyright laws or straining the load a server. We also demonstrate how to scrape data from asynchronously loaded web pages, and how to interact programmatically with web pages when needed.*

Scraping Etiquette

There are two main ways that web scraping can be problematic for a website owner. First, if the scraper doesn't respect the website's terms and conditions or gathers private or proprietary data. Second, if the scraper imposes too much extra server load by making requests too often or in quick succession. These are extremely important considerations in any web scraping program.

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*.^a

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`.^b

^aSee www.robotstxt.org/orig.html and en.wikipedia.org/wiki/Robots_exclusion_standard.

^bPython 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>.

The standard way to get the HTML source code of a website using Python is via the `requests` library.¹ Calling `requests.get()` sends an HTTP GET request to a specified website; the result is an object with a response code to indicate whether or not the request was responded to, and access to the website source code.

¹Though `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/>.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.example.com")
>>> print(response.status_code, response.ok, response.reason)
200 True OK

# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
# ...
```

ACHTUNG!

Consecutive GET requests without pauses can strain a website's server and provoke retaliation. Most servers are designed to identify such scrapers, block their access, and sometimes even blacklist the user. This is especially common in smaller websites that aren't built to handle enormous amounts of traffic. To briefly pause the program between requests, use `time.sleep()`.

```
>>> import time
>>> time.sleep(3)                                # Pause execution for 3 seconds.
```

The amount of necessary wait time depends on the website. Sometimes, `robots.txt` contains a `Request-rate` directive which gives a ratio of the form `<requests>/<seconds>`. If this doesn't exist, pausing for a half-second to a second between requests is typically sufficient. An email to the site's webmaster is always the safest approach, and may be necessary for large scraping operations.

Crawling Through Multiple Pages

While web *scraping* refers to the actual gathering of web-based data, web *crawling* refers to the navigation of a program between webpages. Web crawling allows a program to gather related data from multiple web pages and websites.

Consider `www.wunderground.com`, a site that records weather data in various cities. The page <https://www.wunderground.com/history/airport/KSAN/2012/7/1/DailyHistory.html> records the weather in San Diego on July 1, 2012. Data for previous or subsequent days can be accessed by clicking on the `Previous Day` and `Next Day` links, so gathering data for an entire month or year requires crawling through several pages. The following example gathers temperature data for July 1 through July 4 of 2012.

```

import re
import time
import requests
from bs4 import BeautifulSoup

def wunder_temp(day="/history/airport/KSAN/2012/7/1/DailyHistory.html"):
    """Crawl through Weather Underground and extract temperature data."""
    # Initialize variables, including a regex for finding the 'Next Day' link.
    actual_mean_temp = []
    next_day_finder = re.compile(r"Next Day")
    base_url = "https://www.wunderground.com"           # Web page base URL.
    page = base_url + day                                # Complete page URL.
    current = None

    for _ in range(4):
        while current is None: # Try downloading until it works.
            # Download the page source and PAUSE before continuing.
            page_source = requests.get(page).text
            time.sleep(1)          # PAUSE before continuing.
            soup = BeautifulSoup(page_source, "html.parser")
            current = soup.find(string="Mean Temperature")

        # Navigate to the relevant tag, then extract the temperature data.
        temp_tag = current.parent.parent.next_sibling.next_sibling.span.span
        actual_mean_temp.append(int(temp_tag.string))

        # Find the URL for the page with the next day's data.
        new_day = soup.find(string=next_day_finder).parent["href"]
        page = base_url + new_day                          # New complete page URL.
        current = None

    return actual_mean_temp

```

In this example, the `for` loop cycles through the days, and the `while` loop ensures that each website page loads properly: if the downloaded `page_source` doesn't have a tag whose string is "Mean Temperature", the request is sent again. Later, after locating and recording the Actual Mean Temperature, the function locates the link to the next day's page. This link is stored in the HTML as a relative website path (`/history/airport/...`); the complete URL to the next day's page is the concatenation of the base URL `https://www.wunderground.com` with this relative link.

Problem 1. Modify `wunder_temp()` so that it gathers the Actual Mean Temperature, Actual Max Temperature, and Actual Min Temperature for every day in July of 2012. Plot these three measurements against time on the same plot.

Consider printing information at each iteration of the outer loop to keep track of the program's progress.

An alternative approach that is often useful is to first identify the links to relevant pages, then scrape each of these page in succession. For example, the Federal Reserve releases quarterly data on large banks in the United States at <http://www.federalreserve.gov/releases/lbr>. The following function extracts the four measurements of total consolidated assets for JPMorgan Chase during 2004.

```
def bank_data():
    """Crawl through the Federal Reserve site and extract bank data."""
    # Compile regular expressions for finding certain tags.
    link_finder = re.compile(r"2004$")
    chase_bank_finder = re.compile(r"^JPMORGAN CHASE BK")

    # Get the base page and find the URLs to all other relevant pages.
    base_url="https://www.federalreserve.gov/releases/lbr/"
    base_page_source = requests.get(base_url).text
    base_soup = BeautifulSoup(base_page_source, "html.parser")
    link_tags = base_soup.find_all(name='a', href=True, string=link_finder)
    pages = [base_url + tag.attrs["href"] for tag in link_tags]

    # Crawl through the individual pages and record the data.
    chase_assets = []
    for page in pages:
        time.sleep(1)                      # PAUSE, then request the page.
        soup = BeautifulSoup(requests.get(page).text, "html.parser")

        # Find the tag corresponding to Chase Banks's consolidated assets.
        temp_tag = soup.find(name="td", string=chase_bank_finder)
        for _ in range(10):
            temp_tag = temp_tag.next_sibling
        # Extract the data, removing commas.
        chase_assets.append(int(temp_tag.string.replace(',', '')))

    return chase_assets
```

Problem 2. Modify `bank_data()` so that it extracts the total consolidated assets (“Consol Assets”) for JPMorgan Chase, Bank of America, and Wells Fargo recorded each December from 2004 to the present. In a single figure, plot each bank’s assets against time. Be careful to keep the data sorted by date.

Problem 3. ESPN hosts data on NBA athletes at <http://www.espn.go.com/nba/statistics>. Each player has their own page with detailed performance statistics. For each of the five offensive leaders in points and each of the five defensive leaders in rebounds, extract the player’s career minutes per game (MPG) and career points per game (PPG). Make a scatter plot of MPG against PPG for these ten players.

Asynchronously Loaded Content and User Interaction

Web crawling with the methods presented in the previous section fails under a few circumstances. First, many webpages use *JavaScript*, the standard client-side scripting language for the web, to load portions of their content *asynchronously*. This means that at least some of the content isn't initially accessible through the page's source code. Second, some pages require user interaction, such as clicking buttons which aren't links (`<a>` tags which contain a URL that can be loaded) or entering text into form fields (like search bars).

The *Selenium* framework provides a solution to both of these problems. Originally developed for writing unit tests for web applications, Selenium allows a program to open a web browser and interact with it in the same way that a human user would, including clicking and typing. It also has BeautifulSoup-esque tools for searching the HTML source of the current page.

NOTE

Selenium requires an executable *driver* file for each kind of browser. The following examples use Google Chrome, but Selenium supports Firefox, Internet Explorer, Safari, Opera, and PhantomJS (a special browser without a user interface). See <https://seleniumhq.github.io/selenium/docs/api/py> or <http://selenium-python.readthedocs.io/installation.html> for installation instructions.

To use Selenium, start up a browser using one of the drivers in `selenium.webdriver`. The browser has a `get()` method for going to different web pages, a `page_source` attribute containing the HTML source of the current page, and a `close()` method to exit the browser.

```
>>> from selenium import webdriver

# Start up a browser and go to example.com.
>>> browser = webdriver.Chrome()
>>> browser.get("https://www.example.com")

# Feed the HTML source code for the page into BeautifulSoup for processing.
>>> soup = BeautifulSoup(browser.page_source, "html.parser")
>>> print(soup.prettify())
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>
    Example Domain
  </title>
  <meta charset="utf-8"/>
  <meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
# ...

>>> browser.close()                                # Close the browser.
```

Selenium can deliver the HTML page source to BeautifulSoup, but it also has its own tools for finding tags in the HTML.

Method	Returns
<code>find_element_by_tag_name()</code>	The first tag with the given name
<code>find_element_by_name()</code>	The first tag with the specified <code>name</code> attribute
<code>find_element_by_class_name()</code>	The first tag with the given <code>class</code> attribute
<code>find_element_by_id()</code>	The first tag with the given <code>id</code> attribute
<code>find_element_by_link_text()</code>	The first tag with a matching <code>href</code> attribute
<code>find_element_by_partial_link_text()</code>	The first tag with a partially matching <code>href</code> attribute

Table 9.1: Methods of the `selenium.webdriver.Chrome` class.

Each of the `find_element_by_*`() methods returns a single object representing a *web element* (of type `selenium.webdriver.remote.webelement.WebElement`), much like a BeautifulSoup tag (of type `bs4.element.Tag`). If no such element can be found, a Selenium `NoSuchElementException` is raised. Each webdriver also has several `find_elements_by_*`() methods (elements, plural) that return a list of all matching elements, or an empty list if there are no matches.

Web element objects have methods that allow the program to interact with them: `click()` sends a click, `send_keys()` enters in text, and `clear()` deletes existing text. This functionality makes it possible for Selenium to interact with a website in the same way that a human would. For example, the following code opens up <https://www.google.com>, types “Python Selenium Docs” into the search bar, and hits enter.

```
>>> from selenium.webdriver.common.keys import Keys
>>> from selenium.common.exceptions import NoSuchElementException

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://www.google.com")
...     try:
...         # Get the search bar, type in some text, and press Enter.
...         search_bar = browser.find_element_by_name('q')
...         search_bar.clear()                      # Clear any pre-set text.
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN)       # Press Enter.
...     except NoSuchElementException:
...         print("Could not find the search bar!")
...         raise
... finally:
...     browser.close()
...
```

Problem 4. The arXiv (pronounced “archive”) is an online repository of scientific publications, hosted by Cornell University. Write a function that accepts a string to serve as a search query. Use Selenium to enter the query into the search bar of <https://arxiv.org> and press Enter. The resulting page has up to 25 links to the PDFs of technical papers that match the query. Gather these URLs, then continue to the next page (if there are more results) and continue gathering links until obtaining at most 100 URLs. Return the list of URLs.

NOTE

Using Selenium to access a page's source code is typically much safer, though slower, than using `requests.get()`, since Selenium waits for each web page to load before proceeding. For instance, the arXiv is a somewhat defensive about scrapers (<https://arxiv.org/help/robots>), but Selenium makes it possible to gather info from the website without offending the administrators.

Problem 5. *Project Euler* (<https://projecteuler.net>) is a collection of mathematical computing problems. Each problem is listed with an ID, a description/title, and the number of users that have solved the problem.

Using Selenium, BeautifulSoup, or both, for each of the (at least) 600 problems in the archive at <https://projecteuler.net/archives>, record the problem ID and the number of people who have solved it. Return a list of IDs, sorted from largest to smallest by the number of people who have solved them. That is, the first entry in the list should be the ID of the **most solved** problem, and the last entry in the list should be the ID of the **least solved** problem. (Hint: start by identifying the URLs to each archive page.)

10

Pandas I: 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, syntax, and explore its capabilities for quickly analyzing and presenting data.

Series

A pandas `Series` is generalization of a one-dimensional NumPy array. Like a NumPy array, every `Series` has a data type (`dtype`), and the entries of the `Series` are all of that type. Unlike a NumPy array, every `Series` has an `index` that labels each entry, and a `Series` object can also be given a name to label the entire data set.

```
>>> import numpy as np
>>> import pandas as pd

# Initialize a Series of random entries with an index of letters.
>>> pd.Series(np.random.random(4), index=['a', 'b', 'c', 'd'])
a    0.474170
b    0.106878
c    0.420631
d    0.279713
dtype: float64

# The default index is integers from 0 to the length of the data.
>>> pd.Series(np.random.random(4), name="uniform draws")
0    0.767501
1    0.614208
2    0.470877
3    0.335885
Name: uniform draws, dtype: float64
```

The index in a `Series` is a pandas object of type `Index` and is stored as the `index` attribute of the `Series`. The plain entries in the `Series` are stored as a NumPy array and can be accessed as such via the `values` attribute.

```
>>> s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'], name="some ints")

>>> s1.values                                # Get the entries as a NumPy array.
array([1, 2, 3, 4])

>>> print(s1.name, s1.dtype, sep=", ")    # Get the name and dtype.
some ints, int64

>>> s1.index                                  # Get the pd.Index object.
Index(['a', 'b', 'c', 'd'], dtype='object')
```

The elements of a `Series` can be accessed by either the regular position-based integer index, or by the corresponding label in the index. New entries can be added dynamically as long as a valid index label is provided, similar to adding a new key-value pair to a dictionary. A `Series` can also be initialized from a dictionary: the keys become the index labels, and the values become the entries.

```
>>> s2 = pd.Series([10, 20, 30], index=["apple", "banana", "carrot"])
>>> s2
apple    10
banana   20
carrot   30
dtype: int64

# s2[0] and s2["apple"] refer to the same entry.
>>> print(s2[0], s2["apple"], s2["carrot"])
10 10 30

>>> s2[0] += 5                               # Change the value of the first entry.
>>> s2["dewberry"] = 0                      # Add a new value with label 'dewberry'.
>>> s2
apple    15
banana   20
carrot   30
dewberry   0
dtype: int64

# Initialize a Series from a dictionary.
>>> pd.Series({"eggplant":3, "fig":5, "grape":7}, name="more foods")
eggplant    3
fig         5
grape       7
Name: more foods, dtype: int64
```

Slicing and fancy indexing also work the same way in `Series` as in NumPy arrays. In addition, multiple entries of a `Series` can be selected by indexing a list of labels in the index.

```
>>> s3 = pd.Series({"lions":2, "tigers":1, "bears":3}, name="oh my")
>>> s3
bears    3
lions    2
tigers   1
Name: oh my, dtype: int64

# Get a subset of the data by regular slicing.
>>> s3[1:]
lions    2
tigers   1
Name: oh my, dtype: int64

# Get a subset of the data with fancy indexing.
>>> s3[np.array([len(i) == 5 for i in s3.index])]
bears    3
lions    2
Name: oh my, dtype: int64

# Get a subset of the data by providing several index labels.
>>> s3[ ["tigers", "bears"] ]
tigers    1
bears     3
# Note that the entries are reordered,
# and the name stays the same.
Name: oh my, dtype: int64
```

Problem 1. Create a pandas `Series` where the index labels are the even integers $0, 2, \dots, 50$, and the entries are $n^2 - 1$, where n is the entry's label. Set all of the entries equal to zero whose labels are divisible by 3.

Operations with Series

A `Series` object has all of the advantages of a NumPy array, including entry-wise arithmetic, plus a few additional features (see Table 10.1). Operations between a `Series` S_1 with index I_1 and a `Series` S_2 with index I_2 results in a new `Series` with index $I_1 \cup I_2$. In other words, the index dictates how two `Series` can interact with each other.

```
>>> s4 = pd.Series([1, 2, 4], index=['a', 'c', 'd'])
>>> s5 = pd.Series([10, 20, 40], index=['a', 'b', 'd'])
>>> 2*s4 + s5
a    12.0
b    NaN
c    NaN
d    48.0
dtype: float64
# s4 doesn't have an entry for b, and
# s5 doesn't have an entry for c, so
# the combination is Nan (np.nan / None).
```

Method	Returns
<code>abs()</code>	Object with absolute values taken (of numerical data)
<code>argmax()</code>	The index label of the maximum value
<code>argmin()</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 10.1: Numerical methods of the `Series` and `DataFrame` pandas classes.

Many `Series` are more useful than NumPy arrays primarily because of their index. For example, a `Series` can be indexed by time with a pandas `DatetimeIndex`, an index with date and/or time values. The usual way to create this kind of index is with `pd.date_range()`.

```
# Make an index of the first three days in July 2000.
>>> pd.date_range("7/1/2000", "7/3/2000", freq='D')
DatetimeIndex(['2000-07-01', '2000-07-02', '2000-07-03'],
               dtype='datetime64[ns]', freq='D')
```

Problem 2. Suppose you make an investment of d dollars in a particularly volatile stock. Every day the value of your stock goes up by \$1 with probability p , or down by \$1 with probability $1 - p$ (this is an example of a *random walk*).

Write a function that accepts a probability parameter p and an initial amount of money d , defaulting to 100. Use `pd.date_range()` to create an index of the days from 1 January 2000 to 31 December 2000. Simulate the daily change of the stock by making one draw from a Bernoulli distribution with parameter p (a binomial distribution with one draw) for each day. Store the draws in a pandas `Series` with the date index and set the first draw to the initial amount d . Sum the entries cumulatively to get the stock value by day. Set any negative values to 0, then plot the series using the `plot()` method of the `Series` object.

Call your function with a few different values of p and d to observe the different possible kinds of behavior.

NOTE

The `Series` in Problem 2 is an example of a *time series*, since it is indexed by time. Time series show up often in data science; we will explore them in more depth in another lab.

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

Data Frames

A `DataFrame` is a collection of `Series` that share the same index, and is therefore a two-dimensional generalization of a NumPy array. The row labels are collectively called the *index*, and the column labels are collectively called the *columns*. An individual column in a `DataFrame` object is one `Series`.

There are many ways to initialize a `DataFrame`. In the following code, we build a `DataFrame` out of a dictionary of `Series`.

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> df1 = pd.DataFrame({'series 1': x, "series 2": y})
>>> df1
   series 1  series 2
a -0.365542  1.227960
b  0.080133  0.683523
c  0.737970      NaN
d  0.097878 -1.102835
e        NaN  1.345004
f        NaN  0.217523
```

Note that the index of this `DataFrame` is the union of the index of `Series` `x` and that of `Series` `y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column `1` is `NaN`. This same reasoning explains the other missing values as well. Note that if we take the first column of the `DataFrame` and drop the missing values, we recover the `Series` `x`:

```
>>> df1["series1"].dropna()
a    -0.365542
b     0.080133
c     0.737970
d     0.097878
Name: series 1, dtype: float64
```

ACHTUNG!

A pandas `DataFrame` cannot be sliced in exactly the same way as a NumPy array. Notice how we just used `df1["series 1"]` to access a *column* of the the `DataFrame` `df1`. We will discuss this in more detail later on.

We can also initialize a `DataFrame` using a NumPy array, creating custom row and column labels.

```
>>> data = np.random.random((3, 4))
>>> pd.DataFrame(data, index=['A', 'B', 'C'], columns=np.arange(1, 5))

      1         2         3         4
A  0.065646  0.968593  0.593394  0.750110
B  0.803829  0.662237  0.200592  0.137713
C  0.288801  0.956662  0.817915  0.951016
3 rows    4 columns
```

As with `Series`, if we don't specify the index or columns, the default is `np.arange(n)`, where `n` is either the number of rows or columns.

Viewing and Accessing Data

In this section we will explore some elementary accessing and querying techniques that enable us to maneuver through and gain insight into our data. Try using the `describe()` and `head()` methods for quick data summaries.

Basic Data Access

We can slice the rows of a `DataFrame` much as with a NumPy array.

```
>>> df = pd.DataFrame(np.random.randn(4, 2), index=['a', 'b', 'c', 'd'],
                      columns = ['I', 'II'])
>>> df[:2]

      I         II
a  0.758867  1.231330
b  0.402484 -0.955039

[2 rows x 2 columns]
```

More generally, we can select subsets of the data using the `iloc` and `loc` indexers. The `loc` index selects rows and columns based on their *labels*, while the `iloc` method selects them based on their integer *position*. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than using bracket indexing, because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc/iloc` explicitly, bypasses the extra checks.

```
>>> # select rows a and c, column II
>>> df.loc[['a','c'], 'II']

a    1.231330
c    0.556121
Name: II, dtype: float64

>>> # select last two rows, first column
>>> df.iloc[-2:, 0]

c   -0.475952
d   -0.518989
Name: I, dtype: float64
```

Finally, a column of a `DataFrame` may be accessed using simple square brackets and the name of the column, or alternatively by treating the label as an object:

```
>>> # get second column of df
>>> df['II']          # or, equivalently, df.II

a    1.231330
b   -0.955039
c    0.556121
d    0.173165
Name: II, dtype: float64
```

All of these techniques for getting subsets of the data may also be used to set subsets of the data:

```
>>> # set second columns to zeros
>>> df['II'] = 0
>>> df['II']
a    0
b    0
c    0
d    0
Name: II, dtype: int64

>>> # add additional column of ones
>>> df['III'] = 1
>>> df
      I  II  III
a  -0.460457    0    1
b   0.973422    0    1
c   -0.475952   0    1
d   -0.518989   0    1
```

SQL Operations in pandas

The `DataFrame`, being a tabular data structure, bears an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases, and in this section we will explore how pandas accomplishes some of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, since it can eliminate the need to switch between programming languages for different tasks. Within pandas we can handle both the querying *and* data analysis.

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

```
>>> #build toy data for SQL operations
>>> 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 important 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'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade      5 non-null float64
ID         5 non-null int64
Math_Major 5 non-null object
dtypes: float64(1), int64(1), object(1)
```

We can also get some basic information about the structure of the `DataFrame` using the `head()` or `tail()` methods.

```
>>> mathInfo.head()
   Grade  ID Math_Major  ID  Age  GPA
0    4.0    0        y    0   20  3.8
1    3.0    1        n    2   18  3.0
2    3.5    5        y    4   19  2.8
3    3.0    6        n    6   20  3.8
```

4	4.0	3	n	7	19	3.4
---	-----	---	---	---	----	-----

The method `isin()` is a useful way to find certain values in a `DataFrame`. It compares the input (a list, dictionary, or `Series`) to the `DataFrame` and returns a boolean `Series` showing whether or not the values match. You can then use this boolean array to select appropriate locations. Now let's look at the pandas equivalent of some SQL SELECT statements.

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]

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

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>> studentInfo[studentInfo['Class'].isin(['J', 'Sp'])]['Name']
```

Problem 3. The example above shows how to implement a simple WHERE condition, and it is easy to have a more complex expression. Simply enclose each condition by parentheses, and use the standard boolean operators & (AND), | (OR), and ~ (NOT) to connect the conditions appropriately. Use pandas to execute the following query:

```
SELECT ID, Name from studentInfo WHERE Age > 19 AND Sex = 'M'
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function, which takes as arguments the two `DataFrame` objects to join, 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') # INNER JOIN is the default
      Age Class ID     Name Sex  Grade Math_Major
0    20    Sp   0   Mylan   M    4.0        y
1    21    Se   1   Regan   F    3.0        n
2    22    Se   3    Jess   F    4.0        n
3    20     J   5   Remi   F    3.5        y
4    20     J   6   Matt   M    3.0        n
[5 rows x 7 columns]

>>> # 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
[9 rows x 2 columns]
```

Problem 4. Using a join operation, create a `DataFrame` containing the ID, age, and GPA of all male individuals. You ought to be able to accomplish this in one line of code.

Be aware that other types of SQL-like operations are also possible, such as UNION. When you find yourself unsure of how to carry out a more involved SQL-like operation, the online pandas documentation will be of great service.

Analyzing Data

Although pandas does not provide built-in support for heavy-duty statistical analysis of data, there are nevertheless many features and functions that facilitate basic data manipulation and computation, even when the data is in a somewhat messy state. We will now explore some of these features.

Basic Data Manipulation

Because the primary pandas data structures are subclasses of the `ndarray`, they are valid input to most NumPy functions, and can often be treated simply as NumPy arrays. For example, basic vectorized operations work just fine:

```
>>> x = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), index=['a', 'b', 'd', 'e', 'f'])
>>> x**2
a    1.710289
b    0.157482
c    0.540136
d    0.202580
dtype: float64
>>> z = x + y
>>> z
a    0.123877
b    0.278435
c        NaN
d   -1.318713
e        NaN
f        NaN
dtype: float64
>>> np.log(z)
a   -2.088469
b   -1.278570
```

```
c      NaN
d      NaN
e      NaN
f      NaN
dtype: float64
```

Notice that pandas automatically aligns the indexes when adding two `Series` (or `DataFrames`), so that the the index of the output is simply the union of the indexes of the two inputs. The default missing value `NaN` is given for labels that are not shared by both inputs.

It may also be useful to transpose `DataFrames`, re-order the columns or rows, or sort according to a given column. Here we demonstrate these capabilities:

```
>>> df = pd.DataFrame(np.random.randn(4,2), index=['a', 'b', 'c', 'd'], columns=['I', 'II'])
>>> df
   I          II
a -0.154878 -1.097156
b -0.948226  0.585780
c  0.433197 -0.493048
d -0.168612  0.999194

[4 rows x 2 columns]

>>> df.transpose()
           a          b          c          d
I -0.154878 -0.948226  0.433197 -0.168612
II -1.097156  0.585780 -0.493048  0.999194

[2 rows x 4 columns]

>>> # switch order of columns, keep only rows 'a' and 'c'
>>> df.reindex(index=['a', 'c'], columns=['II', 'I'])
           II          I
a -1.097156 -0.154878
c -0.493048  0.433197

[2 rows x 2 columns]

>>> # sort descending according to column 'II'
>>> df.sort_values('II', ascending=False)
   I          II
d -0.168612  0.999194
b -0.948226  0.585780
c  0.433197 -0.493048
a -0.154878 -1.097156

[4 rows x 2 columns]
```

Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and 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`. The following example illustrates this concept:

```
>>> x = pd.Series(np.arange(5))
>>> y = pd.Series(np.random.randn(5))
>>> x.iloc[3] = np.nan
>>> x + y
0    0.731521
1    0.623651
2    2.396344
3        NaN
4    3.351182
dtype: float64
```

If we are not interested in the missing values, we can simply drop them from the data altogether:

```
>>> (x + y).dropna()
0    0.731521
1    0.623651
2    2.396344
4    3.351182
dtype: float64
```

This is not always the desired behavior, however. It may well be the case that missing data actually corresponds to some default value, such as zero. In this case, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 0, add
>>> x.fillna(0) + y
0    0.731521
1    0.623651
2    2.396344
3    1.829400
4    3.351182
dtype: float64
```

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

Data I/O

Being able to import and export data is a fundamental skill in data science. Unfortunately, with the multitude of data formats and conventions, importing data can often be a painful task. The pandas library seeks to reduce some of the difficulty by providing file readers for various types of formats, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
<code>describe()</code>	Return a <code>Series</code> describing the data structure
<code>head()</code>	Return the first n rows, defaulting to 5
<code>tail()</code>	Return the last n rows, defaulting to 5
<code>to_csv()</code>	Write the index and entries to a CSV file
<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

Table 10.3: Methods for viewing or 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. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a `DataFrame`, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- **delimiter:** This argument specifies the character that separates data fields, often a comma or a whitespace character.
- **header:** The row number (starting at 0) in the CSV file that contains the column names.
- **index_col:** If you want to use one of the columns in the CSV file as the index for the `DataFrame`, set this argument to the desired column number.
- **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 assigned to this argument.

There are several other keyword arguments, but this should be enough to get you started.

When you need to save your data, pandas allows you to write to several different file formats. A typical example is the `to_csv()` function method attached to `Series` and `DataFrame` objects, which writes the data to a CSV file. Keyword arguments allow you to specify the separator character, omit writing the columns names or index, and specify many other options. The code below demonstrates its typical usage:

```
>>> df.to_csv("my_df.csv")
```

Problem 5. The file `crime_data.csv` contains data on types of crimes committed in the United States from 1960 to 2016.

- Load the data into a pandas `DataFrame`, using the column names in the file and the column titled “Year” as the index. Make sure to skip lines that don’t contain data.
- Insert a new column into the data frame that contains the crime rate by year (the ratio of “Total” column to the “Population” column).

- Plot the crime rate as a function of the year.
- List the 5 years with the highest crime rate in descending order.
- Calculate the average number of total crimes as well as burglary crimes between 1960 and 2012.
- Find the years for which the total number of crimes was below average, but the number of burglaries was above average.
- Plot the number of murders as a function of the population.
- Select the Population, Violent, and Robbery columns for all years in the 1980s, and save this smaller data frame to a CSV file `crime_subset.csv`.

Problem 6. In 1912 the RMS *Titanic* sank after colliding with an iceberg. The file `titanic.csv` contains data on the incident. Each row represents a different passenger, and the columns describe various features of the passengers (age, sex, whether or not they survived, etc.)

Start by cleaning the data.

- Read the data into a `DataFrame`. Use the first row of the file as the column labels, but do not use any of the columns as the index.
- Drop the columns "`Sibsp`", "`Parch`", "`Cabin`", "`Boat`", "`Body`", and "`home.dest`".
- Drop any entries without data in the "`Survived`" column, then change the remaining entries to `True` or `False` (they start as 1 or 0).
- Replace null entries in the "`Age`" column with the average age.
- Save the new `DataFrame` as `titanic_clean.csv`.

Next, answer the following questions.

- How many people survived? What percentage of passengers survived?
- What was the average price of a ticket? How much did the most expensive ticket cost?
- How old was the oldest survivor? How young was the youngest survivor? What about non-survivors?

11

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 and explore the data as a whole.*

Overview of Plotting Tools

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

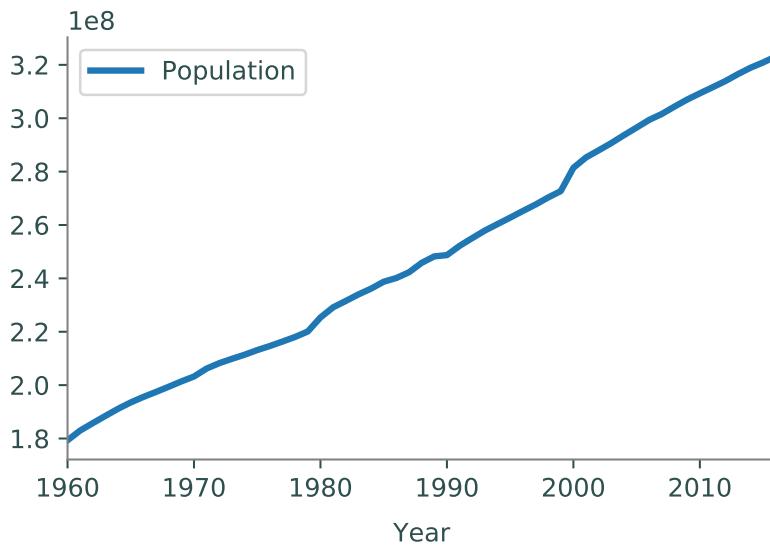
Plot Type	plot() ID	Uses and Advantages
Line plot	<code>"line"</code>	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	<code>"scatter"</code>	Compare exactly two data sets, independent of ordering
Bar plot	<code>"bar", "barch"</code>	Compare categorical or sequential data
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 11.1: Uses for the `plot()` method of the pandas `Series` and `DataFrame`. 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()`, or another matplotlib plotting function, 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

>>> crime = pd.read_csv("crime_data.csv", index_col="Year")
>>> crime.plot(y="Population") # Plot population against the index (years).
```



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

```
>>> plt.plot(crime.index, crime["Population"], label="Population")
>>> plt.xlabel(crime.index.name)
>>> plt.xlim(min(crime.index), max(crime.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>.

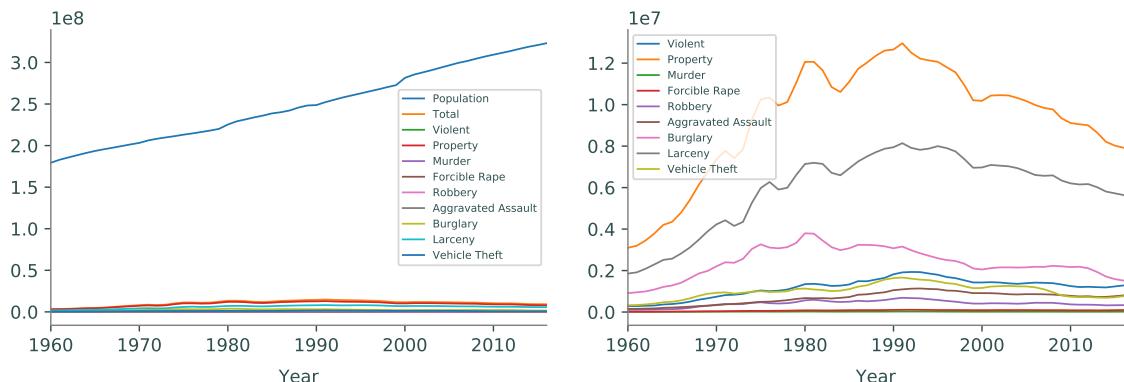
Visualizing an Entire Data Set

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. In fact, the `plot()` method attempts by default to plot **every Series** (column) in the `DataFrame`. This is especially useful with sequential data, like the crime data set.

The crime data set has 11 columns, so the resulting figure, Figure 11.1a, is fairly cluttered. However, it does show that the "`Population`" column is on a completely different scale than the others. Dropping a few columns gives a better overview of the data, shown in Figure 11.1b.

```
# Plot all columns together against the index.
>>> crime.plot(linewidth=1)

# Plot all columns together except for 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(linewidth=1)
```



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

(b) All columns of the crime data set except "Population" and "Total".

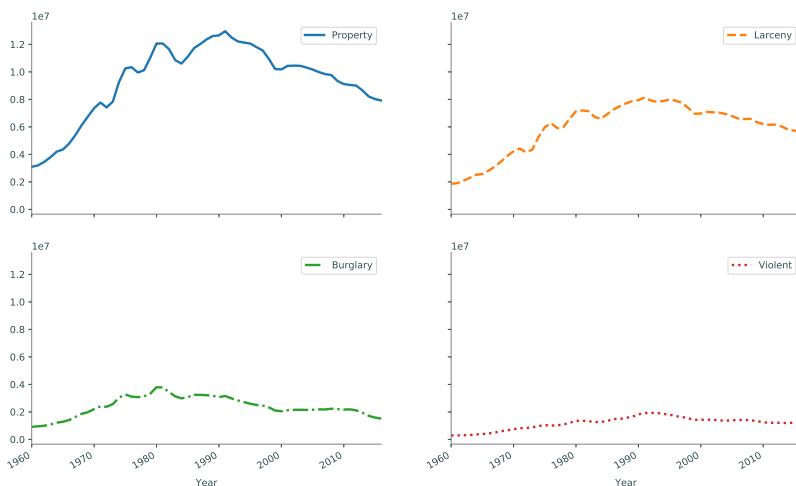
Figure 11.1

ACHTUNG!

The "Population" column differs from the other columns because it is has **different units of measure**: population is measured by "number of people," but all other columns are measured in "number of crimes." Be careful not to plot parts of a data set together if those parts don't have the same units or are otherwise incomparable.

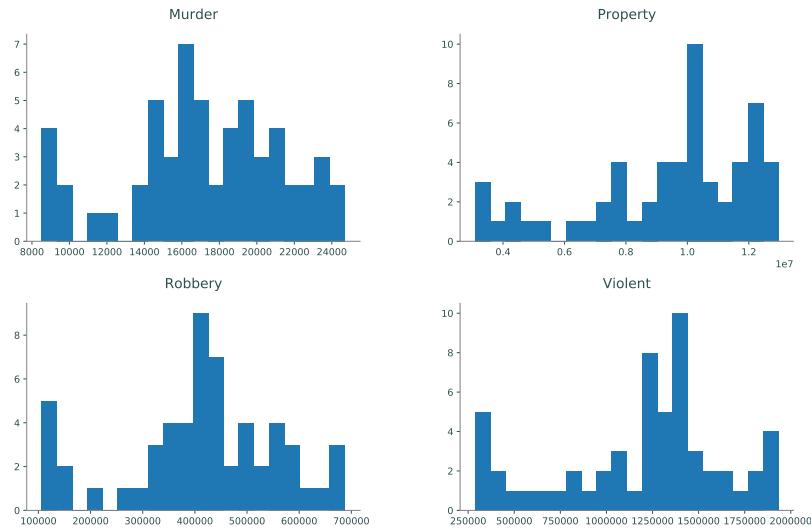
To quickly plot several columns in separate subplots, use `subplots=True` and specify a shape tuple as the `layout` for the plots. Subplots automatically share the same x -axis; set `sharey=True` to force them to share the same y -axis as well.

```
>>> crime.plot(y=["Property", "Larceny", "Burglary", "Violent"],
...               subplots=True, layout=(2,2), sharey=True,
...               style=['-', '--', '-.', ':'])
```



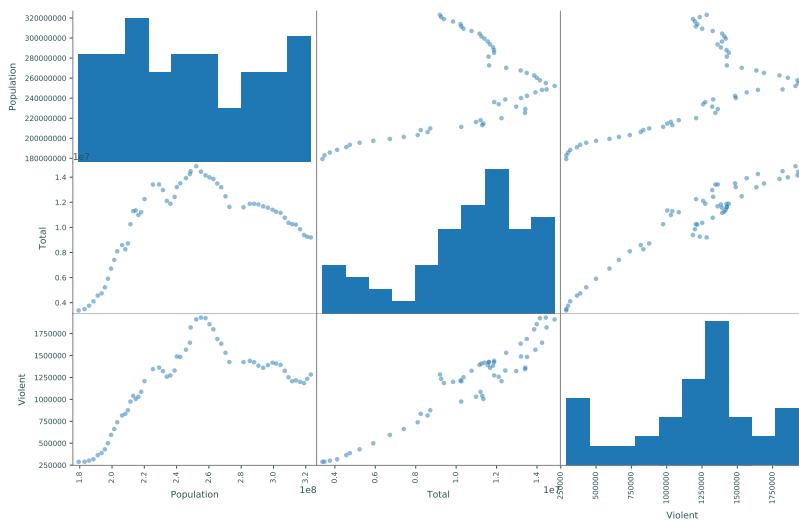
The `plot()` method can generate subplots of any kind of plot. However, since subplots share an *x*-axis by default, histograms turn out poorly whenever there are columns with very different data ranges. For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter.

```
>>> crime[["Violent", "Murder", "Robbery", "Property"]].hist(grid=False, bins=20)
```



Finally, 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 way to very quickly do an initial analysis of the correlation between different columns.

```
>>> pd.plotting.scatter_matrix(crime[["Population", "Total", "Violent"]])
```

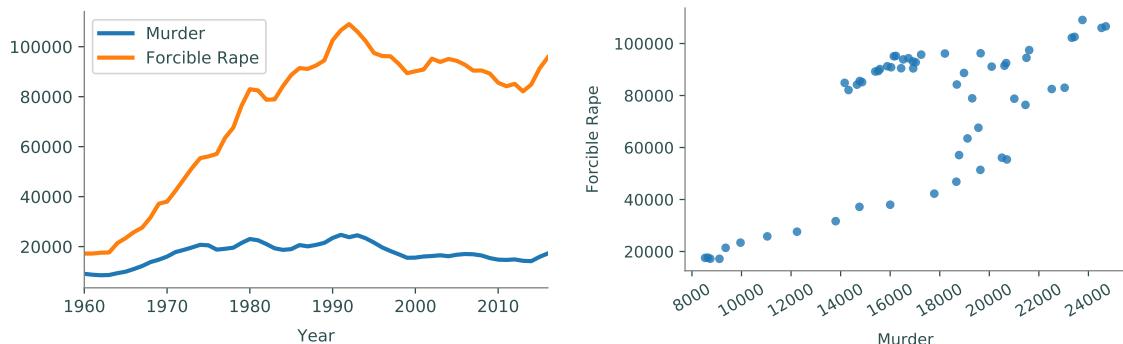


Patterns and Correlations

After visualizing the entire data set initially, the next step is usually to closely compare related parts of the data. For example, Figure 11.1b suggests that the "Murder" and "Forcible Rape" 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 an x and a y column as arguments.

```
# Plot 'Murder' and 'Forcible Rape' as lines against the index.
>>> crime.plot(y=["Murder", "Forcible Rape"])

# Make a scatter plot of 'Murder' against 'Forcible Rape', ignoring the index.
>>> crime.plot(kind="scatter", x="Murder", y="Forcible Rape", alpha=.8, rot=30)
```



What does these graphs show about the data? First of all, rape is more common than murder. Second, rates of rape appear to be steadily increased from the mid 1960's to the mid 1990's before leveling out, while murder rates stay relatively constant. The disparity between rape and murder is confirmed in the scatter plot: the clump of data points at about 15,000 murders and 90,000 rapes shows that there have been many years where rape was relatively high while murder was somewhat low.

ACHTUNG!

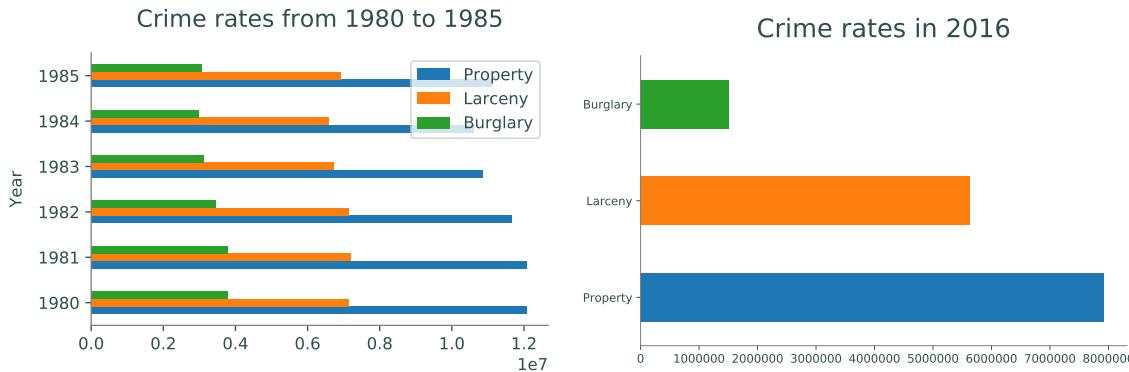
While analyzing data, especially while searching for patterns and correlations, **always** ask yourself if the data makes sense and is trustworthy. What lurking variables could have influenced the data measurements as they were being gathered?

The crime data set is somewhat suspect in this regard. The murder rate is likely accurate, since murder is conspicuous and highly reported, but what about the rape rate? Are the number of rapes increasing, or is the percentage of rapes being reported increasing? (It's probably both!) Be careful about drawing conclusions for sensitive or questionable data.

Figure 11.1b also reveals some general patterns relative to time. For instance, there seems to be a small bump in each type of crime in the early 1980's. Slicing the entries from 1980 to 1985 provides a closer look. Since there are only a few entries, we can treat the data as categorical and make a bar chart.

```
# Plot 'Property' and 'Larceny' rates from 1980 to 1985.
>>> crime.loc[1980:1985, ["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                                         title="Crime rates from 1980 to 1985")

# Plot the most recent year's crime rates for comparison.
>>> crime.iloc[-1][["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                                         title="Crime rates in 2016", color=["C0", "C1", "C2"])
>>> plt.tight_layout()
```



NOTE

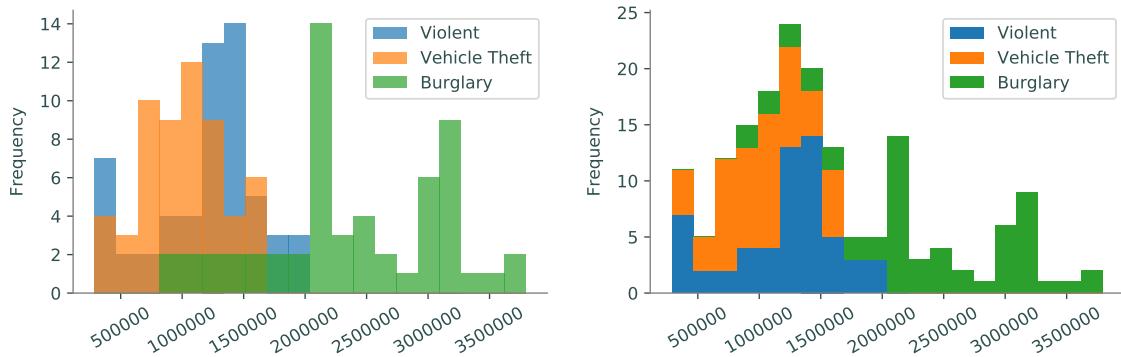
As a general rule, horizontal bar charts (`kind="hbar"`) are better than the default vertical bar charts (`kind="bar"`) because most humans can detect horizontal differences more easily than vertical differences. If the labels are too long to fit on a normal figure, use `plt.tight_layout()` to adjust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

Distributional Visualizations

Histograms are good for examining the distribution of a **single** column in a data set. While pandas is capable of plotting several histograms on the same plot, the results are usually hard to read.

```
# Plot three histograms together.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...               bins=20, alpha=.7, rot=30)

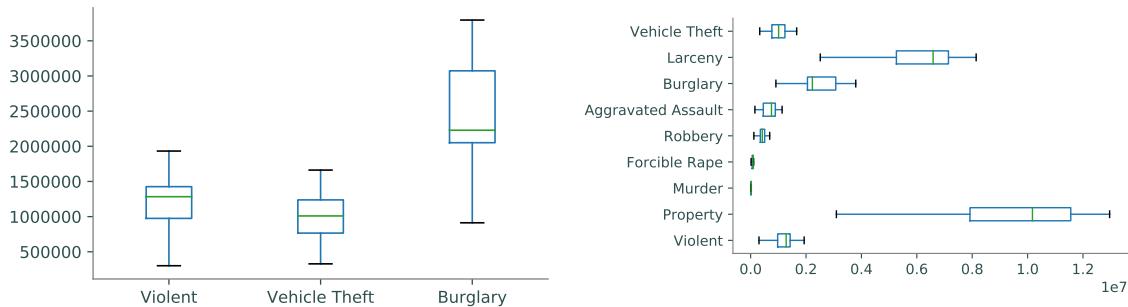
# Plot three histograms, stacking one on top of the other.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...               bins=20, stacked=True, rot=30)
```



Instead of using histograms to compare distributions of data, use box plots. 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. While not quite the same as a histogram, box plots are much better suited to quickly compare relatable distributions.

```
# Compare the distributions of three columns.
>>> crime.plot(kind="box", y=["Violent", "Vehicle Theft", "Burglary"])

# Compare the distributions of all columns but 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(kind="box", vert=False)
```



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 `sat_act.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 11.6a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 11.6b, reveals the **frequency** of points in binned regions.

```
>>> satact = pd.read_csv("sat_act.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)
```

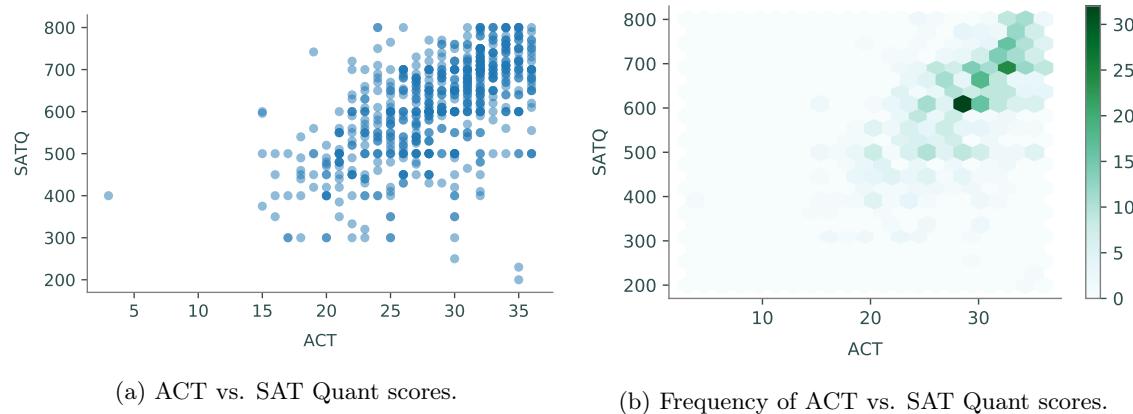


Figure 11.6

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.

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

Principles of Good Data Visualization

Visualization is much more than a set of pretty pictures scattered throughout a paper for the sole purpose of providing contrast to the text. When properly implemented, 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 11.7. What does it depict? We can tell from a simple glance that 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. We have not specified the units for the *x* or the *y* axis, we have no idea what `cons` is, there is no title, and we don't even know where the data came from in the first place.

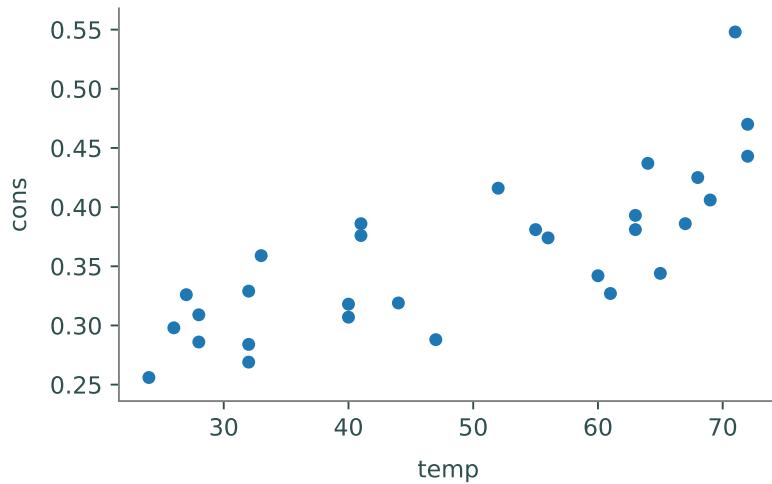


Figure 11.7: Non-specific data.

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. Clearly, we need to explain our data in a useful manner that includes all of the vital information.

Consider again Figure 11.7. 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")
```

We have at this point reproduced the rather substandard plot in Figure 11.7. Using `data('Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.
2. `cons` corresponds to “consumption of ice cream per head” and is measured in pints.
3. `temp` corresponds to temperature, degrees Fahrenheit.
4. The listed source is: “Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.”

We add these important details using the following code. As we have seen in previous examples, pandas automatically generates legends when appropriate. However, although pandas also automatically labels the x and y axes, 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 head*, 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 labelling using xlabel and ylabel
>>> plt.xlabel("Temp (Farenheit)")
>>> plt.ylabel("Consumption per head (pints)")
```

To arbitrarily add the necessary text to the figure, use either `plt.annotate()` or `plt.text()`. Alternatively, add text immediately below wherever the figure is displayed.

```
>>> 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, however, and can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. We again 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 11.8.

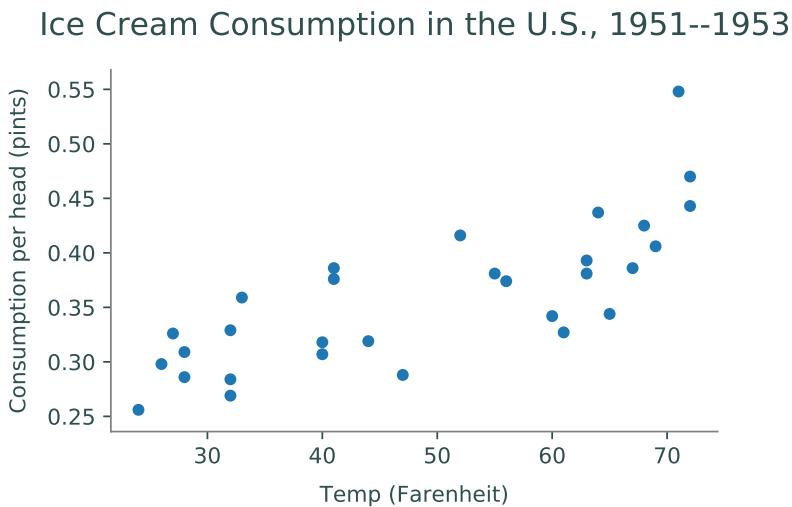


Figure 11.8: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

Problem 1. The `pydataset` module^a contains numerous data sets, each stored as a pandas `DataFrame`.

```
>>> from pydataset import data
# Call data() to see the entire list of data sets.
```

```
# To load a particular data set, enter its ID as an argument to data().
>>> titanic = data("Titanic")
# To see the information about a data set, call data() with show_doc=True.
>>> data("Titanic", show_doc=True)
Titanic

PyDataset Documentation (adopted from R Documentation. The displayed
examples are in R)

## Survival of passengers on the Titanic
```

Visualize and describe at least 5 of the following data sets with 2 or 3 figures each. Comment on the implications and significance of each visualization and give a comprehensive summary of the data set.

- "[Arbuthnot](#)": Ratios of male to female births in London from 1629-1710
- "[trees](#)": Girth, height and volume for black cherry trees
- "[road](#)": Road accident deaths in the United States
- "[birthdeathrates](#)": Birth and death rates by country
- "[bfeed](#)": Child breast feeding records
- "[heart](#)": Survival of patients on the waiting list for the Stanford heart transplant program
- "[lung](#)": Survival in patients with advanced lung cancer from the North Central Cancer Treatment group
- "[birthwt](#)": Risk factors associated with low infant birth weight
- A data set of your choice

Include each of the following in each visualization.

- A clear title, with relevant information for the period or region the data was collected in.
- Axis labels that specify units.
- A legend (if appropriate).
- The source. You may include the source information in your plot or print it after the plot.

^aRun `pip install pydataset` if needed.

12

Pandas III: 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`¹ 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")
```

¹Proceedings 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`".

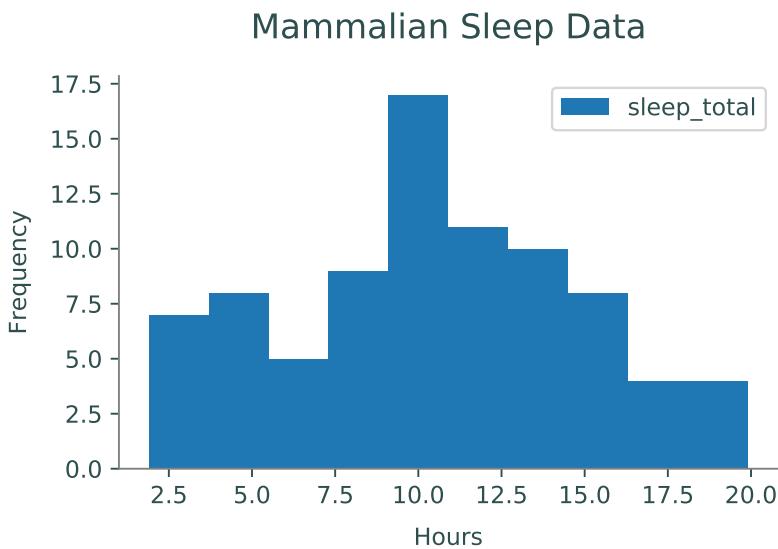


Figure 12.1: "`sleep_total`" frequencies from the mammalian sleep data set.

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
```

For starters, `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 easy it makes it to compare 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()
           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
```

Visualizing Groups

There are a few ways that `groupby()` or similar techniques can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time. The following visualization improve on Figure 12.1 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")
```

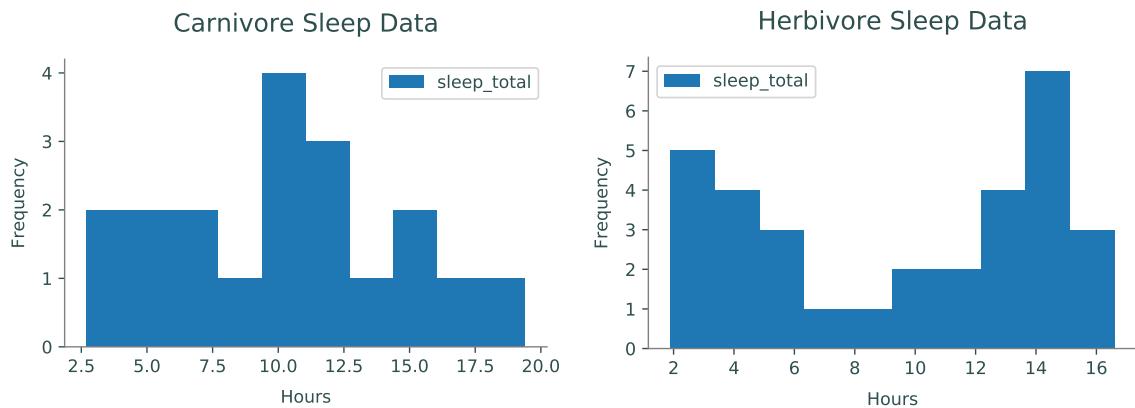
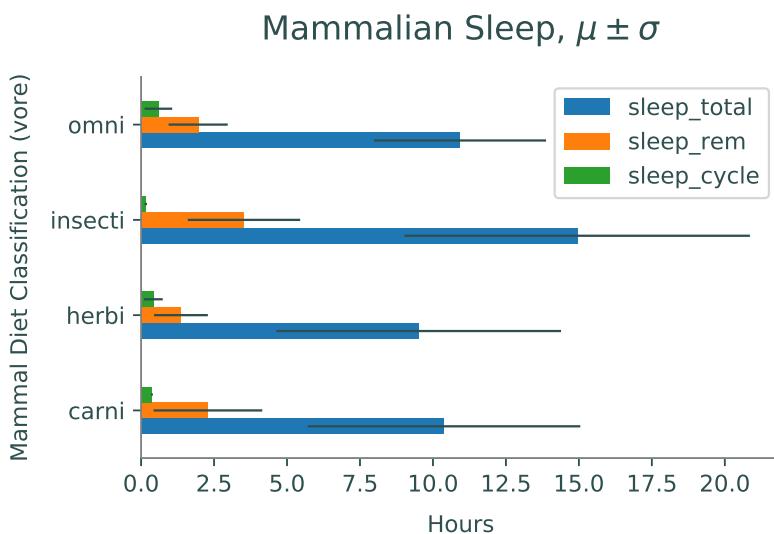


Figure 12.2: "`sleep_total`" histograms for two groups in the mammalian sleep data set.

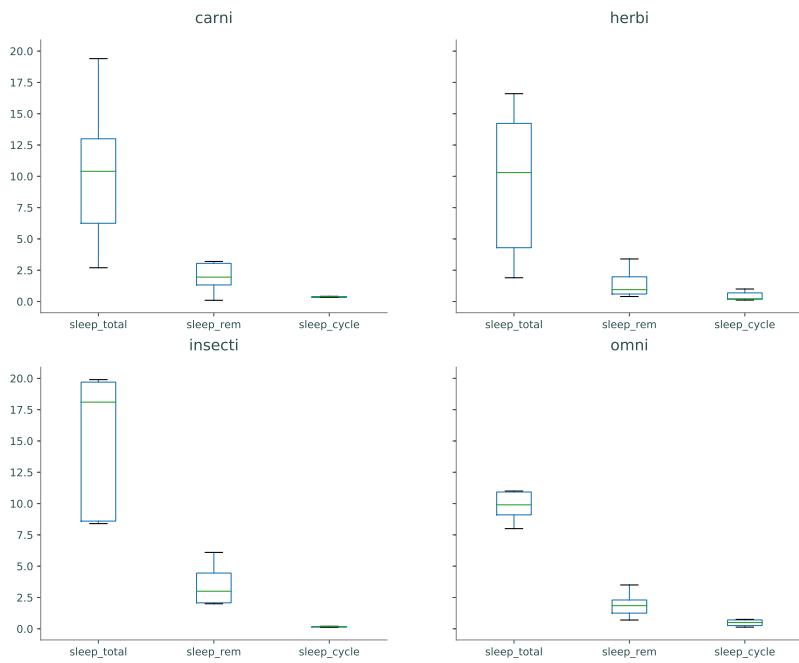
The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

```
>>> vores[["sleep_total", "sleep_rem", "sleep_cycle"]].mean().plot(kind="barh",
           xerr=vores.std(), title=r"Mammalian Sleep, $\mu\pm\sigma$")
>>> plt.xlabel("Hours")
>>> plt.ylabel("Mammal Diet Classification (vore)")
```



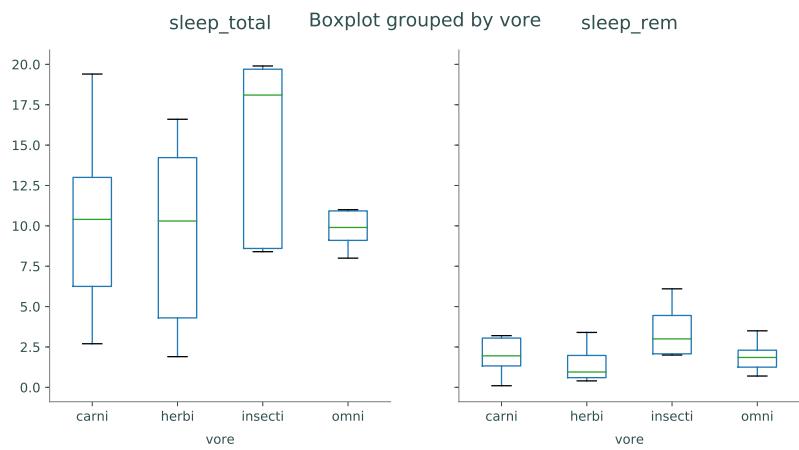
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

```
# 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 1. Examine the following data sets from pydataset and answer the corresponding questions. Use visualizations to support your conclusions.

- "`iris`", measurements of various species of iris flowers.
 1. Which species is easiest to distinguish from the others? How?
 2. Given iris data without a species label, what strategies could you use to identify the flower's species?
(Hint: group the data by poison, then group each subset by treatment.)
- "`poisons`", experimental results of three different poisons and four different treatments.
 1. In general, which poison is most deadly? Which treatment is most effective?
 2. If you were poisoned, how would you choose the treatment if you did not know which poison it was? What if you did know which poison it was?
- "`diamonds`", prices and characteristics of almost 54,000 round-cut diamonds.
 1. How does the color and cut of a diamond affect its price?
 2. Of the diamonds with color "`H`", those with a "`Fair`" cut sell, on average, for a **higher** price than those with an "`Ideal`" (superior) cut. What other factors could explain this unintuitive statistic?

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("{}: {}".format(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    5    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 than males.

Unlike "`HairEyeColor`", many data sets have more than one entry in the data for each grouping (for example, 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. Other options include taking the min, max, standard deviation, or just counting the number of occurrences.

As an example, consider again the Titanic data set found in `titanic.csv`². 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")
>>> titanic = titanic[["Survived", "Pclass", "Sex", "Age", "Fare", "Embarked"]]
>>> titanic["Age"].fillna(titanic["Age"].mean(), inplace=True)
>>> titanic.dropna(inplace=True)

>>> 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
```

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

NOTE

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

```
>>> titanic.groupby(["Sex", "Pclass"])["Survived"].mean().unstack()
Pclass      1.0      2.0      3.0
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   137.0    94.0   106.0
male     61.0    25.0    75.0
```

Discretizing Continuous Data

So far we have 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 just added `"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.

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([6, 1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(0, 4], (0, 4], (0, 4], (0, 4], (4, 8], (4, 8], (4, 8], (0, 4]]
Categories (2, interval[int64]): [(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]   129    85   158
male   (0, 12]     4    11    35
          (12, 18]    4    10    37
          (18, 80]   171   150   420
```

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        1.0  2.0  3.0                1.0  2.0  3.0
Sex   Age
female (0, 12]      -  -    7            1  13  23
              (12, 18]     -  4   23            12  4   5
              (18, 80]     -  31  101           129 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 2. Suppose that someone claims that the city from which a passenger embarked had a strong influence on the passenger's survival rate. Investigate this claim.

1. Check the survival rates of the passengers based on where they embarked from (given in the `"Embarked"` column).
2. Create a pivot table to examine survival rates based on both place of embarkment and gender.
3. What do these tables suggest to you about the significance of where people embarked in influencing their survival rate? Examine the context of the problem, and explain what you think this really means.
4. Investigate the claim further with at least two more pivot tables, exploring other criteria (e.g., class, age, etc.). Carefully explain your conclusions.

Problem 3. Examine the following data sets from `pydataset` and answer the corresponding questions. Use visualizations and/or pivot tables as appropriate to support your conclusions.

- "`npk`", an experiment on the effects of nitrogen (N), phosphate (P), and potassium (K) on the growth of peas.
 1. Which element is most effective in general for simulating growth? Which is the least effective?
 2. What combination of N, P, and K is optimal? What combination is the worst?
- "`swiss`", standardized fertility measures and socio-economic indicators for French-speaking provinces of Switzerland at about 1888.
 1. What is the relationship in the data between fertility rates and infant mortality?
 2. How are provinces that are predominantly Catholic different from non-Catholic provinces, if at all?
 3. What factors in the data are the most important for predicting fertility?
- Examine a data set of your choice. Formulate simple questions about the data and hypothesize the answers to those questions. Demonstrate the correctness of incorrectness of each hypothesis. Explain your conclusions.

13

Pandas IV: Time Series

Lab Objective: Many real world data sets—stock market measurements, ocean tide levels, website traffic, seismograph data, audio signals, fluid simulations, quarterly dividends, and so on—are time series, meaning they come with time-based labels. There is no universal format for such labels, and indexing by time is often difficult with raw data. Fortunately, pandas has tools for cleaning and analyzing time series. In this lab we use pandas to clean and manipulate time-stamped data and introduce some basic tools for time series analysis.

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 (military time).

```
>>> 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 because 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 13.1 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 13.1: 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. An pandas index composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with such an index 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 dates.
>>> 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 1. 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 rows with missing values, cast the "`VALUES`" column to floats, then plot the data. (Hint: Use `lw=.5` to make the line thin enough for the data.)

Generating Time-based Indices

Some time series datasets come without explicit labels, but still 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>	Whether or not to trim the time to midnight

Table 13.2: Parameters for `pd.date_range()`.

Exactly two of the parameters `start`, `end`, and `periods` 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 13.3 for a sampling of some of the options. For a complete list of the options, see <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>.

Parameter	Description
<code>"D"</code>	calendar daily (default)
<code>"B"</code>	business daily
<code>"H"</code>	hourly
<code>"T"</code>	minutely
<code>"S"</code>	secondly
<code>"MS"</code>	first day of the month
<code>"BMS"</code>	first weekday of the month
<code>"W-MON"</code>	every Monday
<code>"WOM-3FRI"</code>	every 3rd Friday of the month

Table 13.3: Options for the `freq` parameter to `pd.date_range()`.

```
# 5 consecutive days starting with 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')

# 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')

# 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')

```

The `freq` parameter also supports more flexible string representations.

```

>>> 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 2. The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Plot the data. (Hint: use the `union()` method of `DatetimeIndex` class.)

Periods

The 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 constructor of the `Period` 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 also (in some cases) indicates the offset of the `Period`. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 13.3.

```

# The default value for 'freq' is "M" for months.
>>> 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('2007-01-01 00:00:00')
>>> p2.end_time
Timestamp('2007-12-31 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.

```
# 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]', freq='Q-DEC')
```

After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# 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]', freq='3M')

# Change frequency to be quarterly.
>>> p.asfreq("Q-DEC")
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

Say you have created a `PeriodIndex`, but the bounds are not exactly where you expected they would be. You can actually shift `PeriodIndex` objects by adding or subtracting an integer, n . The resulting `PeriodIndex` will be shifted by $n \times freq$.

```
# Shift index by 1
>>> p -= 1
>>> p
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so.

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

Similarly, you can switch from timestamps to periods.

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

Problem 3. The file `finances.csv` contains a list of simulated quarterly earnings and expense totals from a fictional company. Load the data into a `Series` or `DataFrame` with a `PeriodIndex` with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the data begins in September 1978. Plot the data.

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

Imagine you have a dataset that does not have datapoints at a fixed frequency. For example, a dataset of website traffic would take on this form. Because the datapoints occur at irregular intervals, it may be more difficult to procure any meaningful insight. In situations like these, *resampling* your data is worth considering.

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, and requires some kind of aggregation to produce a new data set. The first parameter to `resample()` is an offset string from Table 13.3: "`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.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

```

Problem 4. The file `website_traffic.csv` contains records for different visits to a fictitious website. Read in the data, calculate the duration of each visit (in seconds), and convert the index to a `DatetimeIndex`. Use downsampling to calculate the average visit duration by minute, and the average visit duration by hour. Plot both results on the same graph.

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 5. Compute the following information about the DJIA dataset from Problem 1.

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

For the monthly statistics, define the gain (or loss) to be the difference between the DJIA on the last and first days of the month.

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 some kind of calculation on just 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.

```
# Generate a time series using random walk from a uniform distribution.
N = 10000
bias = 0.01
s = np.zeros(N)
s[1:] = np.random.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=.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 α 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.

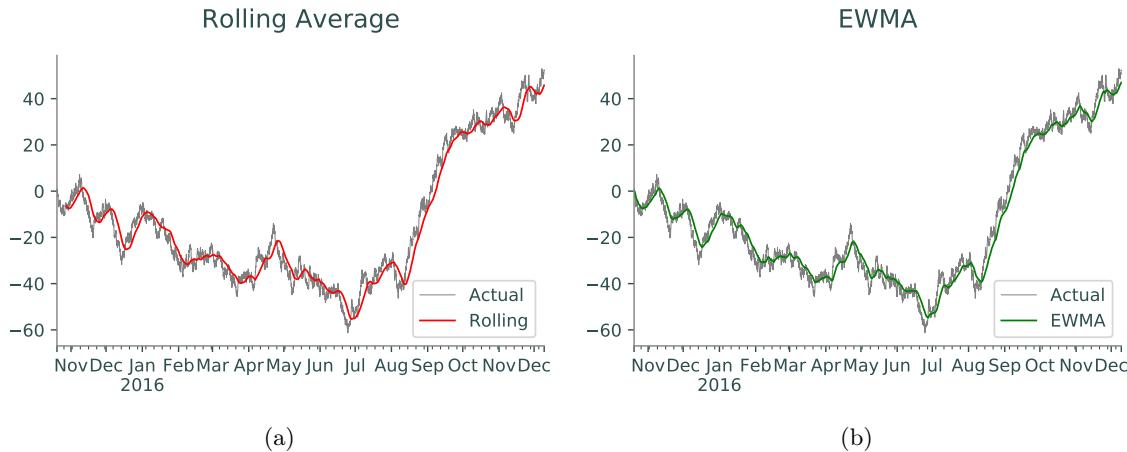


Figure 13.1: Rolling average and EWMA.

```
ax2 = plt.subplot(121)
s.plot(color="gray", lw=.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")
```

Problem 6. Plot the following from the DJIA dataset with a window or span of 30, 120, and 365.

- The original data points.
- Rolling average.
- Exponential average.
- Minimum rolling values.
- Maximum rolling values.

Describe how varying the length of the window changes the approximation to the data.

14

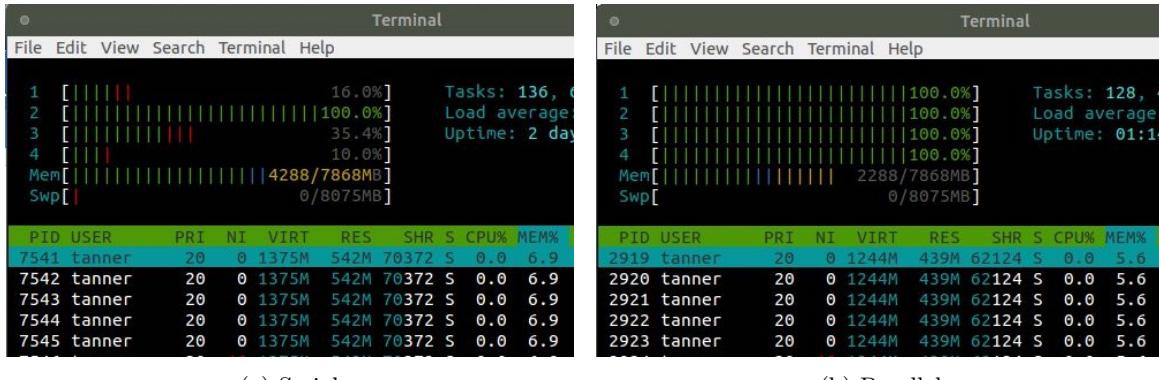
Introduction to Parallel Computing

Lab Objective: *A great deal of modern problems involve such a large number of computations that running them on a single processor becomes either impractical or impossible. However, most computers now have multiple processor cores which can allow various processes to run simultaneously. For even more massive computations, computers can be grouped into clusters that combine both memory and processing power. In this lab, we explore the basic principles of parallel computing by analyzing the cluster setup, the standard parallel commands, and code designs that fully utilize available resources. All concepts are taught using the iPyParallel Python package.*

Why Parallel Computing?

Many modern challenges in data science and mathematics involve enormous computational tasks. In order to overcome these obstacles, faster processors are constantly being developed to improve a computer's capacity. However, making faster processors involves making their transistors, and consequently the processors, smaller. This is a problem because it is difficult to dissipate the heat of a small processor. Thus, there has been a consistent push in the past few decades to *parallelize* computations using multiple processors for greater speed. Though there are many different architectures for parallel computing, essentially, a *supercomputer* or *computer cluster* is a group of normal computers that share their processors and memory for greater performance.

In the majority of circumstances, these processors communicate with each other and coordinate their tasks with a message passing system. The details of the most common message passing interface, or MPI, will be the topic of the next lab. However, there are basic commands used for parallel programming that are shared in both MPI and in the Python module `iPyParallel`. This lab focuses on exploring the basic principles of computing clusters as well as those shared basic commands.



(a) Serial

(b) Parallel

Figure 14.1: In the serial implementation, one core is running the program. In the parallel, it is split across all cores.

Serial Execution vs. Parallel Execution

Serial programs are those that are executed one piece at a time. This can be visualized as a program that runs all of its computations on a single processor or core. Figure 14.1 visualizes what this would look like by using the Linux command `htop`. This command shows which programs are running, how many resources they are using up, and how much each processor core is running.

When running a program in serial, only a fraction of the computer's resources are being used. This can be beneficial for smooth multitasking on a personal computer because programs can run uninterrupted on their own core. However, to reduce the runtime of large computations, it is beneficial to devote all of a computer's resources (or the resources of many computers) to a program. In theory, this parallelization can allow programs to run N times faster where N is the number of processors or processor cores that can be accessed. Due to the necessity for communication, coordination, and travel time on wires, that improvement is not realistic, but the difference is substantial.

Basic Parallel Architecture

There are several common architectures that combine computing resources for parallel processing. Each architecture has a different protocol for sharing memory and processors between computing *nodes*, which are the different simultaneous processing areas. Each also offers unique advantages for the type of processing being completed. However, the parallel commands used with each are very similar.

The following exercises will be based on the standard architecture in the `iPyParallel` module, which gives each node its own processing and memory space. Once constructed, the nodes are coordinated with a master process.

The `iPyParallel` Architecture

The `iPyParallel` architecture can be better understood as a *controller* that distributes messages (or commands) and *engines* that receive the messages and run the processes. The controller is comprised of a hub to manage communications and schedulers to assign processes to the engines. Engines are distributed across computers or processor cores and have the sole task of running their assigned commands. A diagram of the pieces of this architecture can be seen in Figure 14.2.

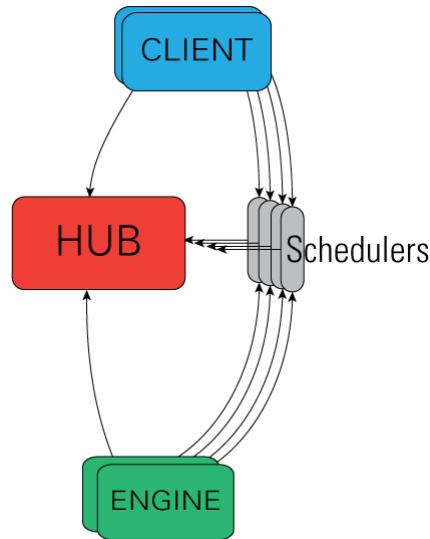


Figure 14.2: An outline of the pieces of the `iPyParallel` architecture. Retrieved from official documentation at <https://ipyparallel.readthedocs.io/en/latest/intro.html#architecture-overview>.

To initialize this architecture, `iPyParallel` must be installed on the machine. Further instructions for installation can be found in the Additional Material section at the end of the lab.

Setting up a Cluster

In order to allow for personal experimentation, the cluster setup for this lab will consist of a single machine with multiple processor cores. However, further information on how to connect multiple machines is contained in Additional Material.

The Controller

To initialize a controller on a machine, run the following command in a terminal window:

```
$ ipcontroller start
```

This command will start a controller process in the directory in which it was run and can be stopped by using `Ctrl+C`. By default, this controller uses JSON files contained in `UserDirectory/.ipython/profile-default/security/` to determine its settings. Once a controller is running, it acts like a server listening for client connections from engine processes.

The Engines

To start an engine process, run the following in a new terminal window:

```
$ ipengine start
```

This command starts an engine client program in the directory in which it was run and by default connects to a controller with the settings defined in the JSON files of `UserDirectory/.ipython/profile-default/security/`. There is no limit to the number of engines that can be started in their own terminal windows and connected to the controller, however, it is recommended that only as many engines are started as there are cores to maximize efficiency. Once started, each engine has its own ID number on the controller that is used for communication.

ACHTUNG!

Engines must be in the same directory as any additional Python files that will to be imported. For example, if `program.py` imports `function` from `important.py`, then `important.py` has to be in the same directory as the engine. This does not apply to installed Python modules.

The Cluster

Once a controller and its engines have been started and are connected, a cluster has successfully been established. The controller will now 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.

Instead of starting the controller and engines in separate terminal windows as previously shown, there is a method to start the controller and engines in one terminal window by running:

```
$ ipcluster start # By default assigns an engine to each processor core.  
$ ipcluster start --n 4 # Starts a cluster with 4 engines.
```

The terminal command `ctrl+c` can be used to stop the terminal window process and `ipcluster stop` can be used to entirely stop the cluster. Though the previous instructions to start a controller and engines in separate terminal windows are less convenient, having unique terminal windows for the engines allows a user to see individual errors in detail and is more convenient for starting a cluster of multiple computers.

NOTE

Jupyter notebooks also have a **Clusters tab** in which clusters can be initialized using an interactive GUI. To enable this tab, run:

```
$ ipcluster nbextension enable
```

The iPyParallel Interface

Since Python is a relatively slow scripting language, and since the main purpose of parallel computing is to speed up run time, most parallel computing is done in a language other than Python. However, it is still fairly easy to speed up run time and to test parallel code logic in the `iPyParallel` environment. Once a cluster has been started with a controller and engines, `iPyParallel` has a specific interface that allows for communication. This interface allows a user to specify what happens on each core and how those cores communicate, which is the heart of all parallel computing.

A controller is manipulated through a `Client` object, which can be initialized as follows:

```
>>> from ipyparallel import Client
>>> client = Client()
>>> client.ids # If you had four processors, the output would be as follows.
[0, 1, 2, 3]
```

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 to be equal on all engines. The remainder of the lab will be focused on the `DirectView` class which can be initialized as follows:

```
>>> dview = client[:] # Group all engines into a Direct View.
>>> dview2 = client[:2] # Group engines 0,1, and 2 into a Direct View.
```

There is also a `targets` object which can be set as a `DirectView` variable or as a parameter in functions. It allows for a subgroup of the Direct View to be specified for subsequent actions. It is accessed as follows.

```
# Target only engines 0 and 2 until changed.
>>> dview.targets = [0,2]
# To revert to all engines,
>>> dview.targets = None
```

Since each engine has its own namespace, we must ensure that the desired modules are imported in each engine. There is more than one way to do this.

For example, the following commands import NumPy to all engines simultaneously.

```
>>> with dview.sync_imports():
...     import numpy
>>> dview.execute(''':np = numpy''')
# Or simply,
>>> dview.execute(''':import numpy as np'''')
```

Problem 1. Write a function `initialize()` that initializes a `Client` object, creates a `DirectView` with all available engines, and imports `scipy.sparse as spar` on all engines.

Basic Parallel Commands in iPyParallel

The `DirectView` gives control of the available engines to the user and allows them to interact with each engine individually or as a group. Each engine can be likened to its own iPython terminal with a namespace in which variables, functions, and imports must be defined individually.

Blocking vs. Non-Blocking

Parallel commands can be implemented in a *blocking* or *non-blocking* format. They are defined as follows:

Blocking - The command is sent and the program halts until the answer is received. This format is usually ideal for problems in which each node is performing the same task.

Non-Blocking - The command is sent and an object is immediately returned with a flag that shows whether the response is ready or not. This allows for greater synchronization between nodes and for large blocks of code to be executed without needing to wait for responses.

Though difficult to grasp initially, this is an extremely important part of parallel processing. An example of the benefits of blocking and non-blocking is demonstrated below.

```
# The following is a blocking implementation of a for loop
>>> dview.execute(''import numpy as np''')
>>> results = []
>>> for i in range(1000):
... results.append(dview.apply_sync(lambda x: np.sum(np.random.random(x)),i))
...
The for loop waits until each answer is gathered, then appends them. This
blocking method takes 16.8495s with 4 engines.
...

# The following is a non-blocking implementation
>>> results2 = []
>>> for j in range(1000):
... results2.append(dview.apply_async(lambda x: np.sum(np.random.random(x)),j))
>>> results2 = [x.get() for x in results2]
...
In this example, the for loop appends an ASyncResult object to the list
and continues the loop. The answers are later retrieved after they have
finished processing. Though this method has two for loops, it takes only
12.9706s with 4 engines.
...
```

As was demonstrated above, 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 check points to retrieve answers and enough memory to store response objects.

There is a `DirectView` variable called `block` that controls the default format. Blocking can also be entered as a parameter in most functions. The variable is accessed as follows.

```
# Make blocking default
>>> dview.block = True
```

Variables on Different Engines

All variables must be initialized on each individual engine. There are a few methods to do this.

Push and Pull

When variables are sent to engines or retrieved from them, the methods `push` and `pull` are used. The following demonstrates these methods.

```
# To share the variables 'a' and 'b' across all engines
>>> dview['a'] = 10
>>> dview['b'] = 5
# These two commands are shorthand for
>>> dview.push({'a':10, 'b':5}, block=True)

# To ensure the variables are on engine 0
>>> client[0]['a']
10

# On all engines
>>> dview['a']
[10, 10, 10]
# Which is shorthand for
>>> dview.pull('a', block=True)
```

The above examples demonstrate simple blocking methods of sending variables to the engines. There are also non-blocking methods that return an `AsyncResult` object that has commands to access its content. This is demonstrated as follows:

```
>>> res = dview.pull(['a', 'b'], block=False)
>>> res.ready()
True
>>> res.get()
[[10, 5], [10, 5], [10, 5], [10, 5]]
```

A table of `AsyncResult` methods are included in Table 14.1.

Class Method	Description
<code>wait(timeout)</code>	Wait until the result is available or until <code>timeout</code> seconds pass. This method always returns <code>None</code> .
<code>ready()</code>	Return whether the call has completed.
<code>successful()</code>	Return whether the call completed without raising an exception. Will raise <code>AssertionError</code> if the result is not ready.
<code>get(timeout)</code>	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 14.1: All information from <https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult>.

Problem 2. Write a function `variables(dx)` that accepts a dictionary of variables. Create a `Client` object and a `Direct View` and distribute the variables. Then, pull the variables in both a blocking and non-blocking format. Return the resultant objects of both formats.

Scatter and Gather

There are also ways to take an array of elements and split them up between all of the Direct View engines. This is called *scattering*. A simple example is contained below.

```
>>> import numpy as np
>>> one = [1, 2, 3, 4]
>>> two = np.array([1, 2, 3, 4])
>>> dview.scatter("one_part", one, block=True)
>>> dview["one_part"]
[[1], [2], [3], [4]]
>>> dview.scatter("two_part", one, block=True, targets=[0,2])
>>> dview["two_part"]
[array([1, 2]), array([3, 4])]
```

Collecting scattered pieces into one object is called *gathering*. An example is contained below.

```
>>> print(dview.gather("one_part", block=True))
[1, 2, 3, 4]
>>> print(dview.gather("two_part", block=True, targets=[0,2]))
array([1, 2, 3, 4])
```

This method of distributing and collecting a dataset is the foundation of the MapReduce algorithm that is prominent in modern Data Science.

Methods for Execution

To execute functions on each of the engines, we can use a few different functions. Though not syntactically the same as commercial parallel programs, the `iPyParallel` functions perform similarly and are as follows.

Execute

The `execute` method is the simplest method for executing commands on parallel engines. It accepts a string with exact syntax for a method to be complete. Examples are as follows. The `htop` command should be used to compare against serial methods.

```
>>> dview.execute('''
... import numpy as np
... rand = np.random.random
... c = np.sum(rand(a+b))
... ''')
>>> dview['c']
```

```
[7.9619371, 7.3431609, 7.4818468, 8.2783728]

# More involved example
>>> dview.scatter("p1", list(range(4)), block=True)
>>> dview.scatter("p2", list(range(10,14)), block=True)
>>> dview["p1"]
[[0], [1], [2], [3]]
>>> dview["p2"]
[[10], [11], [12], [13]]
>>> dview.execute('''
... def adding(a,b):
...     return a+b
... result = adding(p1[0],p2[0])
... ''')
<AsyncResult: execute:finished>
>>> dview['result']
[10, 12, 14, 16]
```

Though primitive, this method functions well for simple procedures. More intricate methods follow.

Apply

The `apply` method is the simplest function distributor for the `iPyParallel` interface. It accepts a function and the arguments that correspond with the function and distributes it to the engines. It has two children, `apply_sync` which blocks and `apply_async` which doesn't block. There is no blocking parameter for these functions. An example follows.

```
# apply_sync always blocks
>>> dview.apply_sync(lambda x: a+b+x, 20)
[35, 35, 35, 35]

# apply_async never blocks
>>> def double_add(x,y):
...     return 2*x + 2*y
>>> answer = dview.apply_async(double_add,5,7)
>>> answer.ready()
True
>>> answer.get()
[24, 24, 24, 24]
```

Note that the engines can access their local variables in any of the execution methods.

Problem 3. Using one of the `apply` methods, write a function `apply_dist(n=1000000)` that prints the mean, max, and min of n draws on each engine from the standard normal distribution. For example if you have four engines running, your output should look like:

```
means = [0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
maxs = [4.0388107, 4.3664958, 4.2060184, 4.3391623]
mins = [-4.1508589, -4.3848019, -4.1313324, -4.2826519]
```

Problem 4. Using the function you wrote in the previous problem, compare the time it takes to run the function with parallel computing to the time it takes to run the function serially. That is, time how long it takes to run the function on all of your machine's engines simultaneously and how long it takes to run the function in a `for` loop n times, where n is the number of engines on your machine. Print the results for 1,000,000; 5,000,000; 10,000,000; and 15,000,000 samples. You should notice an increase in efficiency as the problem size increases.

Map

The `iPyParallel` module also has a generalized version of the Python function `map` that 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. The `map` function does have a blocking parameter, but, like the `apply` function, it also has children, `map_sync` and `map_async`. A couple of examples follow.

```
# Single input function
>>> num_list = [1,2,3,4,5,6,7,8]
>>> def triple(x):
...     return 3*x
>>> answer = dview.map(triple, num_list, block=True)
[3, 6, 9, 12, 15, 18, 21, 24]

# Multiple input function
>>> def add_three(x,y,z):
...     return x+y+z
>>> x_list = [1,2,3,4]
>>> y_list = [2,3,4,5]
>>> z_list = [3,4,5,6]
>>> dview.map_sync(add_three, x_list, y_list, z_list)
[6, 9, 12, 15]

# Engine variable function
>>> def mult(x):
...     return a*x
>>> answer = dview.map_async(mult, x_list)
>>> answer.get()
[10, 20, 30, 40]
```

The `map` function also represents a key component in the MapReduce algorithm and is very important.

There are additional magic methods supplied by `iPyParallel` that make some of these operations easier. These methods are contained in the Additional Material section. More information on `iPyParallel` architecture, interface, and methods at <https://ipyparallel.readthedocs.io/en/latest/index.html>.

Parallel Computing

To this point, the examples of what parallel computing can do may not seem too interesting since each engine is basically producing the same result. There are, however, circumstances in which the engines return different results. In these situations, parallel computing can drastically speed up processing.

Problem 5. In a previous lab, latitude and longitude points of recycle bins and addresses in New York City were analyzed to find the address furthest from a recycle bin. To do so, a KDTree was implemented with the following:

```
>>> from scipy.spatial import KDTree
# Columns should be latitude then longitude
>>> lat_long_array = np.array([ [1,2], [2,3], [3,4] ])
>>> tree = KDTree(lat_long_array)
>>> sample_point = np.array([2,5])
# Queries can be made with
>>> q = tree.query(sample_point)
>>> q
(1.4142135623730951, 2)
```

Write a function called `bin_parallel` that uses a parallel implementation to find the furthest address from a recycle bin. Return the furthest address, its closest bin, and the runtime of your function as a tuple.

The necessary data points have been given as `recycle_bins.npy` and `ny_addr.npy` with latitude and longitude as rows and records as columns.

In theory, using parallel computing for these simple problems should be approximately N times faster, where N is the number of engines you are using. In practice, however, the scaling is not quite linear. This is due in part to the controller running on one of the engines, the computer's standard processes still running, and the overhead of controller and engine communication.

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 possible to solve through parallel computing because solving them serially would take too long. In these types of 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 6. Consider the problem of numerical integration using the trapezoidal rule, depicted in Figure ???. Recall the following formula for estimating an integral using the trapezoidal rule,

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{k=1}^N (f(x_{k+1}) + f(x_k)),$$

where x_k is the k th point, and h is the distance between any two points (note they are evenly spaced).

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered to be embarrassingly parallel.

Write a function called `parallel_trapezoidal_rule()` that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Utilize what you have learned about parallel computing to parallelize the trapezoidal rule in order to estimate the integral of f . That is, evenly divide the points among all available processors and run the trapezoidal rule on each portion simultaneously. 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.

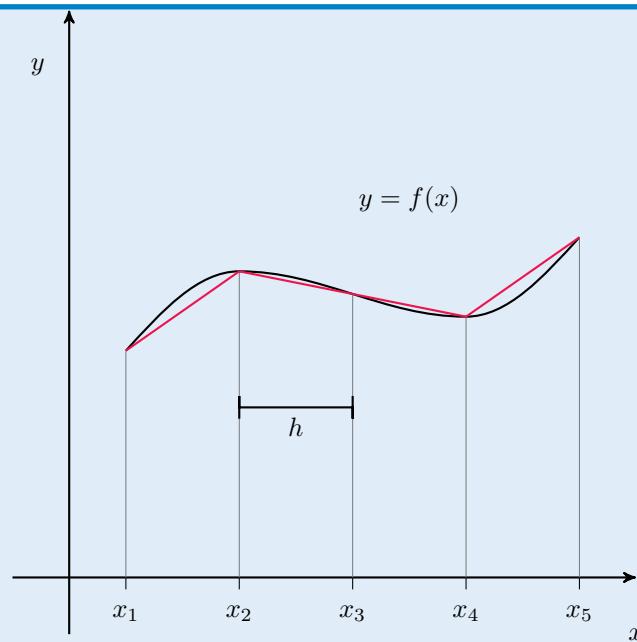


Figure 14.3: A depiction of the trapezoidal rule with uniform partitioning.

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 a later lab.

Additional Material

Installation and Initialization of ipyparallel

If you have not already installed `ipyparallel`, you may do so using the conda package manager.

```
$ conda update conda  
$ conda update anaconda  
$ conda install ipyparallel
```

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 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]]
```

Connecting iPyParallel with MPI

The iPyParallel cluster can be imbued with the ability to interpret MPI commands. More information on making this connection can be found at <https://ipyparallel.readthedocs.io/en/latest/mpi.html>.

15

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. Rather, MPI 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.¹ 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 with a broader architecture in mind.

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.

¹Parallel Programming with MPI, by Peter S. Pacheco, p. 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. If you do not have the MPI library and `mpi4py` installed on your machine, please refer to the Additional Material at the end of this lab. 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.

Using MPI

We will start with a Hello World program.

```

1 #hello.py
2 from mpi4py import MPI
4
5 COMM = MPI.COMM_WORLD
6 RANK = COMM.Get_rank()
7
8 print "Hello world! I'm process number {}".format(RANK)

```

hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpirun -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, though it can oftentimes be a useful tool for debugging.

How does this program work? First, the `mpirun` program is launched. This is the program which starts MPI, a wrapper around whatever program you want to 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 wonderfully 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 `mpirun` 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 "The number of processes is {}".format(SIZE)

```

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

```

```
1 RANK = MPI.COMM_WORLD.Get_rank()
4 print "My rank is {}".format(RANK)
```

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
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)
```

separateCode.py

Problem 1. Write a program in which the processes with an even rank print "Hello" and process with an odd rank print "Goodbye." Print the process number along with the "Hello" or "Goodbye" (for example, "Goodbye from process 3").

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 "Process 1: Sending: {}".format(num_buffer)
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 "Process 0: Waiting for the message... current num_buffer={}".format(
16        num_buffer)
17    COMM.Recv(num_buffer, source=1)
18    print "Process 0: Message received! num_buffer={}".format(num_buffer)

```

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. It cannot, for example, simply pass a string. The string would have to be packaged inside an array first.

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]
```

Send_example.py

Problem 2. Write a script `passVector.py` that runs on two processes and passes an n by 1 vector of random values from one process to the other. Write it so that the user passes the value of n in as a command-line argument. Hint: This code will be useful in remembering how to pass command-line arguments.

```

from sys import argv

# Pass in the first command line argument as n
n = int(argv[1])
```

```
mpirun -n 2 python passVector.py 3
```

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.

Problem 3. Write a script `passCircular.py` 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 program should work for any number of processes. Hint: Remember that `Send` and `Recv` are blocking functions but that `Send`. Does the order in which `Send` and `Recv` are called matter?

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`).

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 π by plotting random points inside a square. The probability of a given point landing inside the inscribed circle should be $\pi/4$.

```

1 import random
2
3 n = 100000
4 s = 0
5
6 for i in range(n):
7     x = random.uniform(-1.0,1.0)
8     y = random.uniform(-1.0,1.0)
```

```
10     if x**2 + y**2 <= 1:  
11         s += 1  
12  
print 4.0*s/n
```

pi.py

Problem 4. Write a script using n processes to find the volume of a unit m -sphere testing p random points in the unit m dimensional box. All of these variables should be passed from the command line (the n processes are passed in the mpirun command as in previous problems). The n processes should pass their individual results up to the root process, which then calculates an overall average.

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.

Additional Material

Installing mpi4py

1. For All Systems: The easiest installation is using `conda install mpi4py`. You may also run `pip install mpi4py`

Part II

Machine Learning Algorithms

16 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

$$\mathbf{x}_{k+1} = F_k \mathbf{x}_k + B_k \mathbf{u}_k + \boldsymbol{\varepsilon}_k \quad (16.1)$$

where F_k is a state-transition model, B_k is a control-input model, \mathbf{u}_k is a control vector, and $\boldsymbol{\varepsilon}_k$ 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 . 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

$$\mathbf{z}_k = H_k \mathbf{x}_k + \boldsymbol{\delta}_k \quad (16.2)$$

where H_k is the observation model mapping the state space to the observation space, and δ_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 , B_k , and H_k are all matrices, and \mathbf{x}_k , \mathbf{u}_k , \mathbf{z}_k , and δ_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 , H_k , \mathbf{u}_k , Q_k , and R_k with F , H , \mathbf{u} , Q , and R . We will also assume that $B = I$ 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 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

$$\mathbf{x} = \begin{pmatrix} s_x \\ s_y \\ V_x \\ V_y \end{pmatrix},$$

where s_x and s_y give the x and y coordinates of the position (in meters), and V_x and V_y give the horizontal and vertical components of the velocity (in meters per second), 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:

$$\begin{aligned}s'_x &= s_x + 0.1V_x \\ s'_y &= s_y + 0.1V_y.\end{aligned}$$

Further, since the only force acting on the projectile is gravity (we are ignoring things like wind resistance), the horizontal velocity remains constant:

$$V'_x = V_x.$$

The vertical velocity, however, does change due to the effects of gravity. From basic Newtonian mechanics, we have

$$V'_y = V_y - 0.1 \cdot 9.8.$$

In summary, over one time step, the state evolves from \mathbf{x} to \mathbf{x}' , where

$$\mathbf{x}' = \begin{pmatrix} s_x + 0.1V_x \\ s_y + 0.1V_y \\ V_x \\ V_y - 0.98 \end{pmatrix}.$$

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

$$\begin{aligned}z_x &= s_x + \delta_x \\ z_y &= s_y + \delta_y,\end{aligned}$$

where (δ_x, δ_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.

We now wish to simulate a sequence of states and observations from the dynamical system. 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 16.1 and 16.2.

Problem 3. Add a method to your `KalmanFilter` class to generate a state and observation sequence by evolving the system from a given initial state (the function `numpy.random.multivariate_normal` will be useful). To do this, implement the following:

```
def evolve(self, x0, N):
    """
```

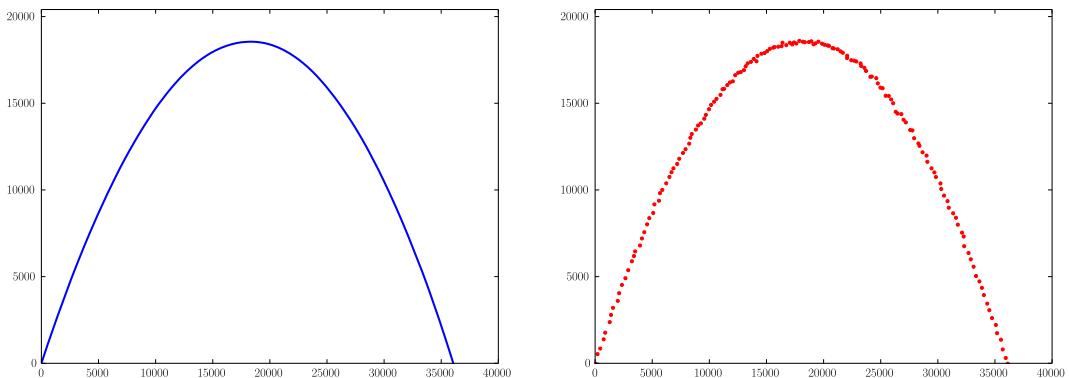


Figure 16.1: State sequence (left) and sampling of observation sequence (right).

Compute the first N states and observations generated by the Kalman system.

Parameters

x0 : ndarray of shape (n,)
 The initial state.
N : integer
 The number of time steps to evolve.

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

Simulate the true and observed trajectory 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). Your results should qualitatively match those given in Figure 16.1.

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.

We evolve the filter recursively, as follows:

Predict	$\hat{\mathbf{x}}_{k k-1} = F\hat{\mathbf{x}}_{k-1 k-1} + \mathbf{u}$
	$P_{k k-1} = FP_{k-1 k-1}F^T + Q$
Update	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - H\hat{\mathbf{x}}_{k k-1}$
	$S_k = HP_{k k-1}H^T + R$
	$K_k = P_{k k-1}H^T S_k^{-1}$
	$\hat{\mathbf{x}}_{k k} = \hat{\mathbf{x}}_{k k-1} + K_k\tilde{\mathbf{y}}_k$
	$P_{k k} = (I - K_kH)P_{k k-1}$

The more observations we have, the greater the accuracy of these estimates becomes (i.e the norm of the accuracy matrix converges to 0).

Problem 4. Add code to your `KalmanFilter` class to estimate 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).
    
```

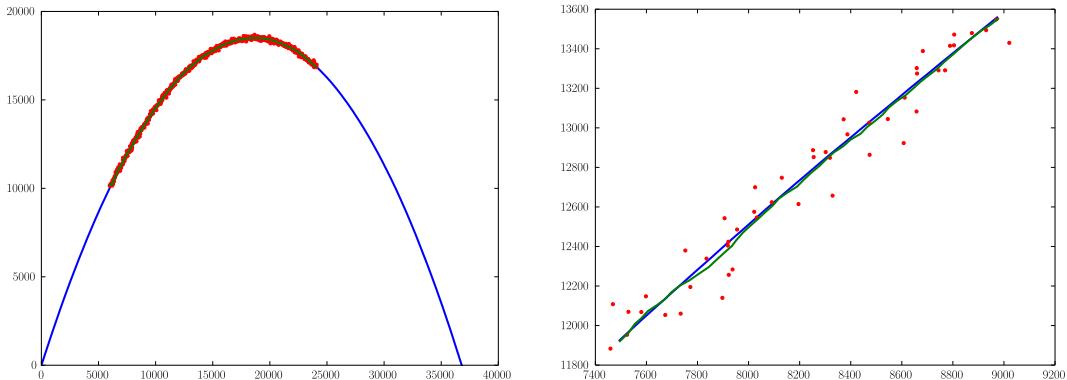


Figure 16.2: State estimates together with observations and true state sequence (detailed view on the right).

```

>Returns
-----
out : ndarray of shape (n,N)
      Sequence of state estimates (each column is an estimate).
"""
pass

```

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: the NumPy function `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 16.2.

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

$$\hat{\mathbf{x}}_{n+1|m} = F\hat{\mathbf{x}}_{n|m} + \mathbf{u}$$

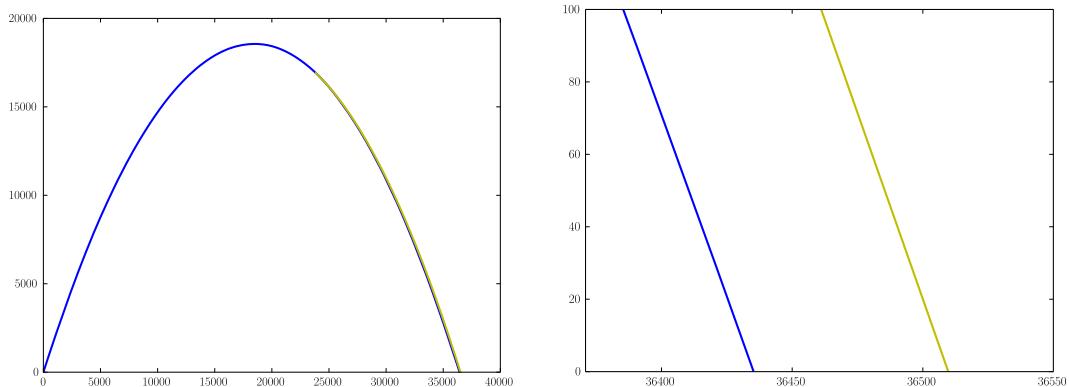


Figure 16.3: Predicted vs. actual point of impact (detailed view on right).

Problem 6. Add a function 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 function:

```
def predict(self,x,k):
    """
    Predict the next k states in the absence of observations.

    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 16.3.

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.

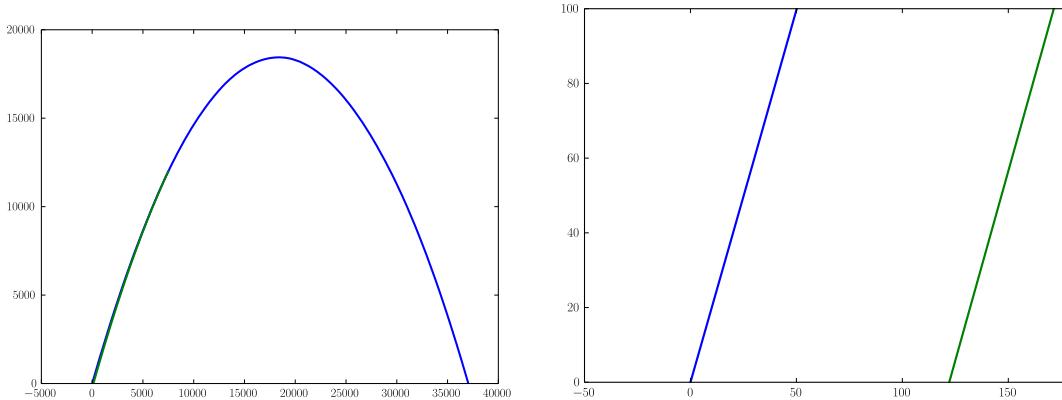


Figure 16.4: Predicted vs. actual point of origin (detailed view on right).

Problem 8. Add a function to your class that rewinds the system from a given state estimate, returning predictions for the previous states. Do so by implementing the following function:

```
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. (The point of origin is the first point along the trajectory where the y coordinate is 0.) Plot these predicted states (in cyan) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 16.4.

Repeat the prediction starting with $\hat{\mathbf{x}}_{600}$. Compare to the previous results. Which is better? Why?

17 ARMA Models

Lab Objective: *Fit and forecast ARMA models.*

An ARMA(p, q) model is a covariance-stationary discrete stochastic process $\{z_t\}$ that satisfies

$$z_t - \mu = \left(\sum_{i=1}^p \phi_i (z_{t-i} - \mu) \right) + a_t + \left(\sum_{j=1}^q \theta_j a_{t-j} \right) \quad (17.1)$$

where $\mu = E[z_t]$ and a_t are identically-distributed Gaussian variables with variance σ_a^2 . We note that the assumption that $\{z_t\}$ is covariance-stationary is equivalent to the condition that the roots of the polynomial in B

$$\phi(B) = 1 - \sum_{i=1}^p \phi_i B^i \quad (17.2)$$

lie outside of the unit circle.

The first sum on the right hand side of 17.1 is interpreted as an “autoregression” since it is a linear combination of previously observed values of z_t . The second sum is interpreted as a “moving average” of the current and previous error terms; though formally similar to an average, note that the θ_j need not be positive nor sum to one. We say that an ARMA(p, q) model is an “autoregressive moving-average model of order p, q ”.

Likelihood via Kalman Filter

In a general ARMA(p, q) model, the likelihood is a function of the unobserved error terms a_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 a state-space representation of an ARMA(p, q) model. If $r = \max(p, q + 1)$, we write

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \quad (17.3)$$

$$H = [1 \ \theta_1 \ \theta_2 \ \cdots \ \theta_{r-1}] \quad (17.4)$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad (17.5)$$

$$w_t \sim \text{MVN}(0, Q), \quad (17.6)$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Then the linear stochastic dynamical system

$$x_{t+1} = Fx_t + w_t \quad (17.7)$$

$$z_t = Hx_t + \mu \quad (17.8)$$

describes the same process as the original ARMA model. Note that the equation for z_t involves a deterministic component, namely μ . The Kalman filter theory developed in the previous lab, however, assumed no deterministic component for the observations z_t , so you should subtract off the mean μ from the time series observations z_t when using them in the predict and update steps.

Let $\Theta = \{\phi_i, \theta_j, \mu, \sigma_a^2\}$ be the set of parameters for an ARMA(p, q) model. Suppose we have a set of observations z_1, z_2, \dots, z_n , denoted collectively by $\{z_t\}$. Using the chain rule, we can factorize the likelihood of the model under these data as

$$p(\{z_t\}|\Theta) = \prod_{t=1}^n p(z_t|z_{t-1}, \dots, z_1, \Theta) \quad (17.9)$$

Since we have assumed that the error terms are Gaussian, each conditional distribution in 17.9 is also Gaussian, and is completely characterized by its mean and variance. But these two quantities are easily found via the Kalman filter, namely

$$\text{mean} \quad H\hat{x}_{t|t-1} + \mu \quad (17.10)$$

$$\text{variance} \quad HP_{t|t-1}H^T \quad (17.11)$$

where $\hat{x}_{t|t-1}$ and $P_{t|t-1}$ are found during the Predict step. The likelihood becomes

$$p(\{z_t\}|\Theta) = \prod_{t=1}^n N(z_t; H\hat{x}_{t|t-1} + \mu, HP_{t|t-1}H^T) \quad (17.12)$$

We begin the recursion by letting

$$\hat{x}_{1|0} = \mathbb{E}(x_1) = 0 \quad (17.13)$$

$$\text{vec}(P_{1|0}) = \mathbb{E}[(x_1 - \mathbb{E}x_1)(x_1 - \mathbb{E}x_1)^T] = [I_{r^2} - (F \otimes F)]^{-1} \cdot \text{vec}(Q) \quad (17.14)$$

where `vec` flattens a matrix and \otimes is the Kronecker product (`numpy.kron`).

Problem 1. Write a function that computes the log-likelihood of an ARMA(p, q) model, given a time series z_t .

```
def arma_likelihood(time_series, phis=array([]), thetas=array([]), mu=0.,
                    sigma=1.):
    """
    Return the log-likelihood of the ARMA model parameters, given the time
    series.

    Parameters
    -----
    time_series : ndarray of shape (n,1)
        The time series in question, z_t
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables

    Returns
    -----
    log_likelihood : float
        The log-likelihood of the model
    """
    pass
```

When done correctly, your function should match the following output:

```
>>> arma_likelihood(time_series_a, phis=array([0.9]), mu=17., sigma=0.4)
-77.6035
```

Identification and Fitting

When modeling a data set with an ARMA(p, q) model, the order of the model must be determined, as well as the other parameters. The process of choosing p and q is called *model identification*. Different methods have been used; for example, Box and Jenkins propose a methodology that involves examining the estimated autocorrelation and partial-autocorrelation functions of the data. We will choose p and q that minimize the Akaike information criterion with a correction (AICc), given by

$$2k \left(1 + \frac{k+1}{n-k} \right) - 2\ell(\Theta) \quad (17.15)$$

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 17.12 over the space of parameters Θ . We can do so by using the function from Problem 1 along with some optimization routine, such as `scipy.optimize.fmin`.

Problem 2. Write a function that accepts a time series $\{z_t\}$ and returns the parameters of the model that minimize the AICc, given the constraint that $p \leq 3$, $q \leq 3$.

```
def arma_fit(time_series):
    """
    Return the ARMA model that minimizes AICc for the given time series,
    subject to p,q <= 3.

    Parameters
    -----
    time_series : ndarray of shape (n,1)
        The time series in question, z_t

    Returns
    -----
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables
    """
    pass
```

Here's a hint for performing the optimization at each step, using `scipy.optimize.fmin`.

```
>>> # assume p, q, and time_series are defined
>>> def f(x): # x contains the phis, thetas, mu, and sigma
>>>     return -1*arma_likelihood(time_series, phis=x[:p], thetas=x[p:p+q-1],
>>>     mu=x[-2], sigma=x[-1])
>>> # create initial point
>>> x0 = np.zeros(p+q+2)
>>> x0[-2] = time_series.mean()
>>> x0[-1] = time_series.std()
>>> sol = op.fmin(f,x0,maxiter=10000, maxfun=10000)
```

The variable `sol` is a flat array of length $p + q + 2$, whose first p entries give the optimal values for the ϕ polynomial, the next q entries give the optimal values for the θ polynomial, and the last two entries give the optimal values for μ and σ_a , respectively. Notice that we defined a wrapper function `f` to feed into the `scipy.optimize.fmin` routine. This wrapper function returns the *negative* of the log likelihood, since the optimization routine we are calling finds the minimum of a function, and we are interested in the *maximum* of the log likelihood.

Your code should produce the following output, where the input data is found in `time_series_a.txt` (it may take a minute or so to run):

```
>>> arma_fit(time_series_a)
(array([ 0.9087]), array([-0.5759]), 17.0652..., 0.3125...)
```

Problem 3. Use your solution from Problem 2 to fit models to the data found in `time_series_a.txt`, `time_series_b.txt`, `time_series_c.txt`. Report the fitted parameters p, q, Θ .

Forecasting

The Kalman filter provides a straightforward way to predict future states, by giving the mean and variance of the conditional distribution of future observations.

$$z_{t+k}|z_1, \dots, z_t \sim N(z_{t+k}; H\hat{x}_{t+k|t} + \mu, HP_{t+k|t}H^T) \quad (17.16)$$

Recall the relations

$$\hat{x}_{t+k|t} = F\hat{x}_{t+k-1|t} \quad (17.17)$$

$$P_{t+k|t} = FP_{t+k-1|t}F^T + Q \quad (17.18)$$

Problem 4. Forecast each data set ahead 20 intervals using the parameters discovered from Problem 3, and plot their expected values along with the original data set. Also plot the expected values plus and minus σ_{t+k} , and plus and minus $2\sigma_{t+k}$ to demonstrate credible intervals.

Note that we need the values of $\hat{x}_{n|n}$ and $P_{n|n}$ to get started. As usual, these estimates can be found using the Predict and Update recursions. Initialize $\hat{x}_{1|0}$ and $P_{1|0}$ as before, run the recursions until you obtain $\hat{x}_{n|n}$ and $P_{n|n}$, and then calculate the future estimates $\hat{x}_{t+k|t}$ and $P_{t+k|t}$. Use these to calculate the expected value and standard deviation for forecasted values (given by $H\hat{x}_{t+k|t} + \mu$ and $\sqrt{HP_{t+k|t}H^T}$, respectively).

```
def arma_forecast(time_series, phis=array([]), thetas=array([]), mu=0.,
                  sigma=1., future_periods=20):
    """
    Return forecasts for a time series modeled with the given ARMA model.
    """


```

Parameters

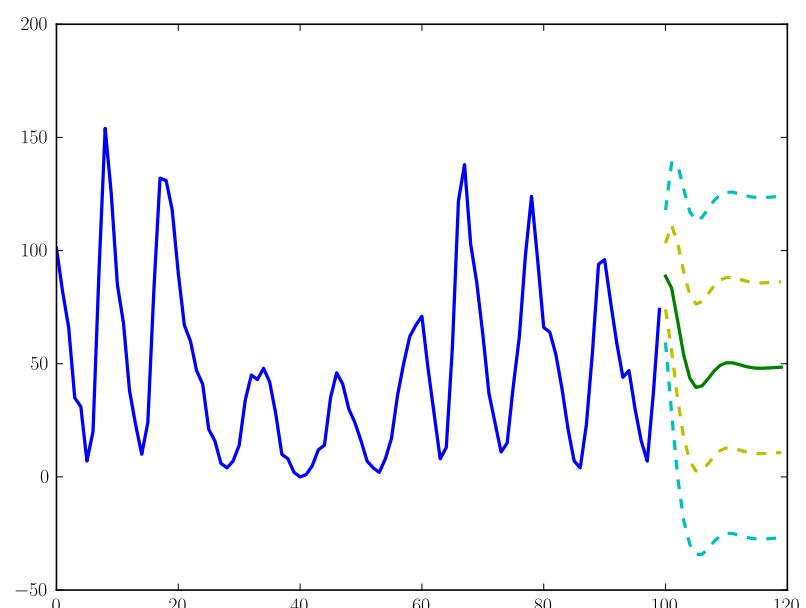
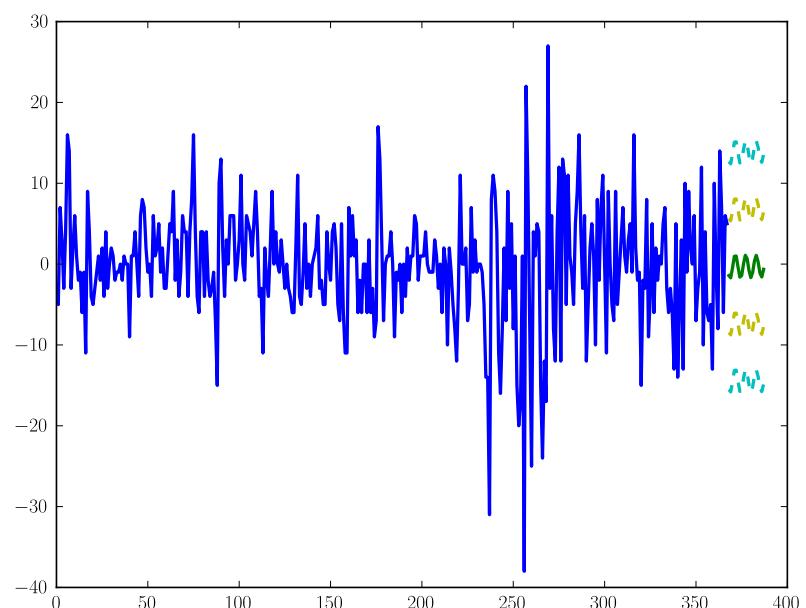
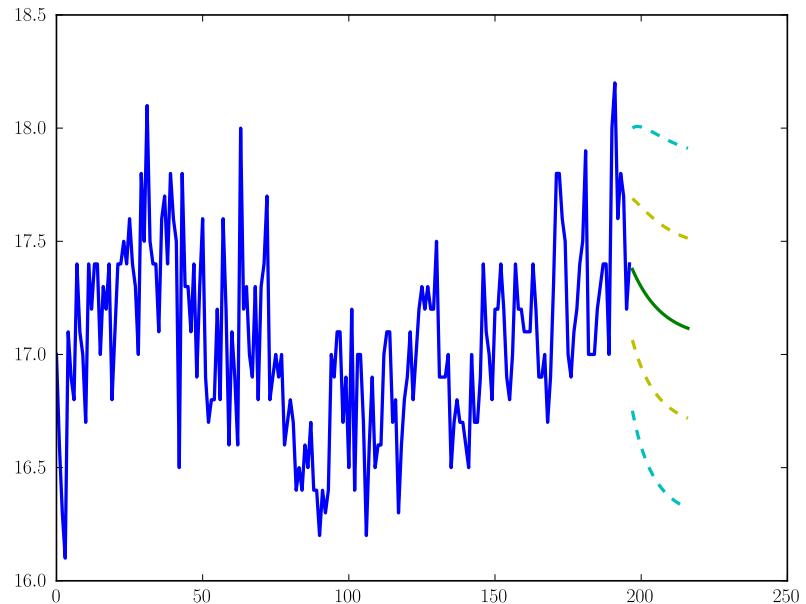
```
time_series : ndarray of shape (n,1)
    The time series in question, z_t
phis : ndarray of shape (p,)
    The phi parameters
thetas : ndarray of shape (q,)
    The theta parameters
mu : float
    The parameter mu
sigma : float
    The standard deviation of the a_t random variables
future_periods : int
    The number of future periods to return

Returns
-----
e_vals : ndarray of shape (future_periods,)
    The expected values of z for times n+1, ..., n+future_periods
sigs : ndarray of shape (future_periods,)
    The standard deviations of z for times n+1, ..., n+future_periods
"""
pass
```

You should get the following result:

```
>>> arma_forecast(time_series_a, phis, thetas, mu, sigma, 4)
(array([ 17.3762,  17.3478,  17.322 ,  17.2986]),
 array([ 0.3125,  0.3294,  0.3427,  0.3533]))
```

Your results (when using twenty future periods) should match those in Figure 17.1.



18

Discrete Hidden Markov Models

Lab Objective: *Understand how to use discrete Hidden Markov Models.*

Given a discrete state-space Hidden Markov Model (HMM) with parameters λ and an observation sequence O , we would like to answer three questions:

1. What is $\mathbb{P}(O|\lambda)$? In other words, what is the likelihood that our model generated the observation sequence?
2. What is the most likely state sequence to have generated O , given λ ?
3. How can we choose the parameters λ that maximize $\mathbb{P}(O|\lambda)$?

The answers to these questions are centered around the *forward-backward* algorithm for HMMs. For the second question, the approach taken in this lab will be to find the state sequence maximizing the expected number of correct states. The third question is an example of *unsupervised learning*, since we are attempting to learn (or fit) model parameters using data (the observation sequence O) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

We assume throughout this lab that the HMM has a discrete state space of cardinality N and a discrete observation space of cardinality M . In this context $\lambda = (A, B, \pi)$, where A is a $N \times N$ column-stochastic matrix (the state transition model), B is a $M \times N$ column-stochastic matrix (the state observation model), and π is a stochastic vector of length N (the initial state distribution). Further, O is a vector of length T with values in the set $\{1, 2, \dots, M\}$.

ACHTUNG!

The mathematical exposition in the lab assumes the standard 1-based indexing of vectors and matrices. Be sure to carefully translate the various formulae into 0-based indexing when implementing these methods for Python coding. This means that, in Python, your array containing the observation sequence O will actually have values in the set $\{0, 1, \dots, M - 1\}$ so that they may be used to index the matrix B correctly.

Throughout this lab, we will be using the following toy HMM to verify your code.

```
>>> # toy HMM example to be used to check answers
>>> A = np.array([[.7, .4], [.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> pi = np.array([.6, .4])
>>> obs = np.array([0, 1, 0, 2])
```

Problem 1. To start off your implementation of the HMM, define a class object which you should call "hmm". Then add the initialization method, in which you should set the *self* aspects A, B, and pi to be None objects. You will be adding methods throughout the remainder of the lab.

The Forward Pass

Our first task is to efficiently compute $\log \mathbb{P}(O|\lambda)$. We can do this using the *forward pass* of the forward-backward algorithm. We must take care to compute all values in a numerically stable way; we do this by properly scaling values as necessary.

We compute a scaled forward probability matrix $\hat{\alpha}$ of dimension $T \times N$ as follows: Let $\hat{\alpha}_{i,:}, B_{i,:}$ denote the i -th rows of $\hat{\alpha}$ and B , respectively, let \odot denote the Hadamard (or entry-wise) product of arrays, and let $\langle \cdot, \cdot \rangle$ denote the standard dot product. (Note that here, using 0-based indexing and the toy HMM example, $B_{O_3,:}$ would refer to $[.5, .1]$.) Then

- $c_1 = \langle \pi, B_{O_1,:} \rangle^{-1}$
- $\hat{\alpha}_{1,:} = c_1(\pi \odot B_{O_1,:})$
- For $t = 2, \dots, T$:

$$c_t = \langle A\hat{\alpha}_{t-1,:}, B_{O_t,:} \rangle^{-1}$$

$$\hat{\alpha}_{t,:} = c_t((A\hat{\alpha}_{t-1,:}) \odot B_{O_t,:})$$

The matrix $\hat{\alpha}$ will be of use when fitting parameters, but we can compute the desired log probability using the scaling factors c_t as follows:

$$\log \mathbb{P}(O|\lambda) = - \sum_{t=1}^T \log c_t.$$

Problem 2. Implement the forward pass by adding the following method to your class:

```
def _forward(self, obs):
    """
    Compute the scaled forward probability matrix and scaling factors.

    Parameters
    -----
    obs : ndarray of shape (T,)
        The observation sequence
```

```

Returns
-----
alpha : ndarray of shape (T,N)
    The scaled forward probability matrix
c : ndarray of shape (T,)
    The scaling factors c = [c_1,c_2,...,c_T]
"""
pass

```

To verify that your code works, you should get the following output using the toy HMM:

```

>>> h = hmm()
>>> h.A = A
>>> h.B = B
>>> h.pi = pi
>>> alpha, c = h._forward(obs)
>>> print -(np.log(c)).sum() # the log prob of observation
-4.6429135909

```

The Backward Pass

The backward pass of the forward-backward algorithm produces values that can be used to calculate the most likely state sequence corresponding to an observation sequence.

We compute a scaled backward probability matrix $\hat{\beta}$ of dimension $T \times N$ as follows:

- $\hat{\beta}_{T,i} = c_T$ for $i = 1, \dots, N$
- $\hat{\beta}_{t,:} = c_t A^T (B_{O_{t+1},:} \odot \hat{\beta}_{t+1,:})$ for $t = T - 1, \dots, 1$

(Above, A^T is the *transpose* of A , not the T -th power of A .)

It turns out that

$$\mathbb{P}(\mathbf{x}_t = i | O, \lambda) = \frac{\hat{\alpha}_{t,i} \hat{\beta}_{t,i}}{\sum_{j=1}^N \hat{\alpha}_{t,j} \hat{\beta}_{t,j}}$$

and so we can easily compute the most likely state at time t by

$$\mathbf{x}_t^* = \operatorname{argmax}_i \hat{\alpha}_{t,i} \hat{\beta}_{t,i}.$$

This is the solution to the second question posed at the beginning of the lab.

Problem 3. Implement the backward pass by adding the following method to your class:

```

def _backward(self, obs, c):
    """
    Compute the scaled backward probability matrix.

    Parameters

```

```
-----
obs : ndarray of shape (T,)
    The observation sequence
c : ndarray of shape (T,)
    The scaling factors from the forward pass

Returns
-----
beta : ndarray of shape (T,N)
    The scaled backward probability matrix
"""
pass
```

Using the same toy example as before, your code should produce the following output:

```
>>> beta = h._backward(obs, c)
>>> print beta
[[ 3.1361635  2.89939354]
 [ 2.86699344  4.39229044]
 [ 3.898812   2.66760821]
 [ 3.56816483  3.56816483]]
```

Computing the δ and γ Probabilities

Having implemented both parts of the forward-backward algorithm, we are closing in on the solution to question three, namely that of fitting parameters λ that maximize $\mathbb{P}(O|\lambda)$. At this stage, we combine the information accumulated in the forward-backward algorithm to produce a three-dimensional array $\hat{\delta}$ of shape $(T-1) \times N \times N$ whose entries are related to $\mathbb{P}(\mathbf{x}_t = i, \mathbf{x}_{t+1} = j | O, \lambda)$, as well as a $T \times N$ matrix $\hat{\gamma}$ whose entries are related to $\mathbb{P}(\mathbf{x}_t = i | O, \lambda)$. The relevant formulae are

$$\hat{\delta}_{t,i,j} = \frac{\hat{\alpha}_{t,i} A_{j,i} B_{O_{t+1},j} \hat{\beta}_{t+1,j}}{\sum_{k,l} \hat{\alpha}_{t,k} A_{l,k} B_{O_{t+1},l} \hat{\beta}_{t+1,l}}$$

for $t = 1, \dots, T-1$ and $i, j = 1, \dots, N$,

$$\hat{\gamma}_{t,i} = \sum_{j=1}^N \hat{\delta}_{t,i,j}$$

for $t = 1, \dots, T-1$ and $i = 1, \dots, N$, and finally

$$\hat{\gamma}_{T,:} = \frac{\hat{\alpha}_{T,:} \odot \hat{\beta}_{T,:}}{\langle \hat{\alpha}_{T,:}, \hat{\beta}_{T,:} \rangle}.$$

Problem 4. Add the following method to your class to compute the δ and γ probabilities.

```
def _delta(self, obs, alpha, beta):
```

```

"""
Compute the delta probabilities.

Parameters
-----
obs : ndarray of shape (T,)
    The observation sequence
alpha : ndarray of shape (T,N)
    The scaled forward probability matrix from the forward pass
beta : ndarray of shape (T,N)
    The scaled backward probability matrix from the backward pass

Returns
-----
delta : ndarray of shape (T-1,N,N)
    The delta probability array
gamma : ndarray of shape (T,N)
    The gamma probability array
"""

pass

```

While writing a triply-nested loop may be the simplest way to convert the formula into code, it is possible to use array broadcasting to eliminate two of the loops, which will speed up your code.

Check your code by making sure it produces the following output, using the same toy example as before.

```

>>> delta, gamma = h._delta(obs, alpha, beta)
>>> print delta
[[[ 0.14166321  0.0465066 ]
 [ 0.37776855  0.43406164]]

 [[ 0.17015868  0.34927307]
 [ 0.05871895  0.4218493 ]]

 [[ 0.21080834  0.01806929]
 [ 0.59317106  0.17795132]]]

>>> print gamma
[[ 0.18816981  0.81183019]
 [ 0.51943175  0.48056825]
 [ 0.22887763  0.77112237]
 [ 0.8039794   0.1960206 ]]

```

Choosing Better Parameters

After running the forward-backward algorithm and computing the δ probabilities, we are now in a position to choose new parameters $\lambda' = (A', B', \pi')$ that increase the probability of observing our data, i.e.

$$\mathbb{P}(O | \lambda') \geq \mathbb{P}(O | \lambda).$$

The update formulas are given by

$$A'_{i,j} = \frac{\sum_{t=1}^{T-1} \hat{\delta}_{t,j,i}}{\sum_{t=1}^{T-1} \hat{\gamma}_{t,j}}$$

$$B'_{i,j} = \frac{\sum_{t=1}^T \hat{\gamma}_{t,j} \mathbf{1}_{\{O_t=i\}}}{\sum_{t=1}^T \hat{\gamma}_{t,j}}$$

$$\pi' = \hat{\gamma}_{1,:}$$

where $\mathbf{1}_{\{O_t=i\}}$ is one if $O_t = i$ and zero otherwise.

Problem 5. Implement the parameter update step by adding the following method to your class:

```
def _estimate(self, obs, delta, gamma):
    """
    Estimate better parameter values.

    Parameters
    -----
    obs : ndarray of shape (T,)
        The observation sequence
    delta : ndarray of shape (T-1,N,N)
        The delta probability array
    gamma : ndarray of shape (T,N)
        The gamma probability array
    """
    # update self.A, self.B, self.pi in place
    pass
```

Verify that your code produces the following output on the toy HMM from before:

```
h._estimate(obs, delta)
>>> print h.A
[[ 0.55807991  0.49898142]
 [ 0.44192009  0.50101858]]
>>> print h.B
[[ 0.23961928  0.70056364]
 [ 0.29844534  0.21268397]
 [ 0.46193538  0.08675238]]
>>> print h.pi
[ 0.18816981  0.81183019]
```

Fitting the Model

We are now ready to put everything together into a learning algorithm. Given a sequence of observations, a maximum number of iterations K , and a convergence tolerance threshold ϵ , we fit a HMM model using the following procedure:

- Randomly initialize parameters $\lambda = (A, B, \pi)$
- Compute $\log \mathbb{P}(O | \lambda)$
- For $i = 1, 2, \dots, K$:
 - Run forward pass
 - Run backward pass
 - Compute δ probabilities
 - Update model parameters
 - Compute $\log \mathbb{P}(O | \lambda)$ according to new parameters
 - If change in log probabilities is less than ϵ , break
 - Else, continue

The most convenient way to randomly initialize stochastic matrices is to draw from the Dirichlet distribution, which produces vectors with nonnegative entries that sum to 1. The following Python code initializes M , A , B , and π using this technique:

```
>>> # assume N is defined
>>> # define M to be the number of distinct observed states
>>> M = len(set(obs))
>>> A = np.random.dirichlet(np.ones(N), size=N).T
>>> B = np.random.dirichlet(np.ones(M), size=N).T
>>> pi = np.random.dirichlet(np.ones(N))
```

The learning algorithm is essentially an optimization over the parameter space (i.e. the space of tuples of stochastic arrays having the proper dimensions) with respect to the objective function $\mathbb{P}(O | \lambda)$. The algorithm is guaranteed to increase the objective function at each iteration, so it is sure to converge. However, the objective function is riddled with local maxima, and so the outcome depends heavily on the randomly selected starting values for A , B , and π . Figure 22.2 illustrates the issues involved. The log probability stays approximately constant for the first 100 iterations. This indicates that the algorithm is not exploring the parameter space enough, and the parameters found at the 100-th iteration are virtually the same as those found at the first or second iteration. After the first 100 iterations, however, the algorithm is finally able to explore more of the parameter space and hence make better progress toward increasing the objective function. The moral of the story is that you may need to train the HMM a few times, using different starting values, and then keep the model that has the highest log likelihood.

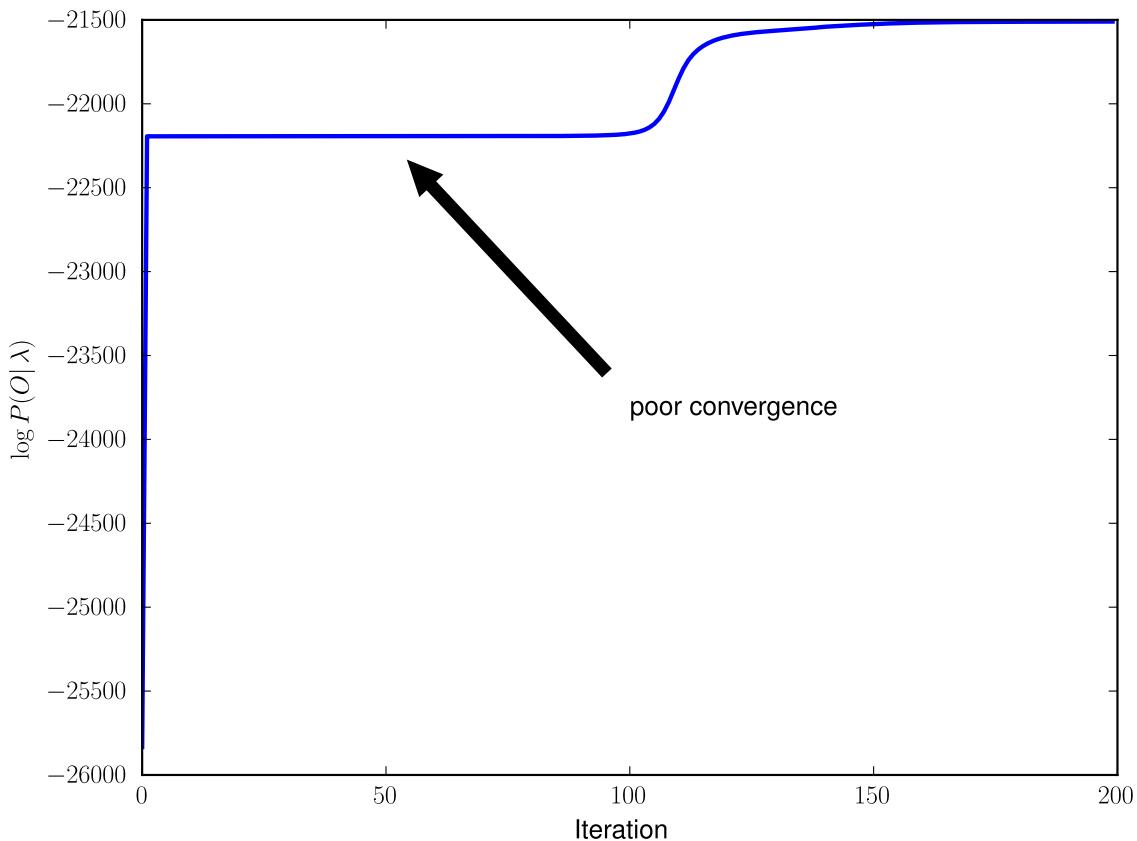


Figure 18.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

Problem 6. Implement the learning algorithm by adding the following method to your class:

```
def fit(self, obs, A, B, pi, max_iter=100, tol=1e-3):
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    -----
    obs : ndarray of shape (T,)
        Observation sequence on which to train the model.
    A : stochastic ndarray of shape (N,N)
        Initialization of state transition matrix
    B : stochastic ndarray of shape (M,N)
        Initialization of state observation matrix
    pi : stochastic ndarray of shape (N,)
        Initialization of initial state distribution
```

```

max_iter : integer
    The maximum number of iterations to take
tol : float
    The convergence threshold for change in log-probability
"""
# initialize self.A, self.B, self.pi
# run the iteration
pass

```

We now turn to the data found in the file `declaration.txt`. This file contains the text of the Declaration of Independence. We will 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 a useable data structure, we need to read in the file, process the string as necessary, and then map the characters to integer values. We provide helper code below to accomplish this task for various files in various languages:

```

>>> 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):
    # 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\f"}
    >>> 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
```

Now apply this helper code to `declaration.txt`.

```
>>> symbols, obs = prep_data('declaration.txt')
```

Problem 7. You are now ready to train a HMM using the Declaration of Independence data. Use $N = 2$ states and $M = \text{len}(\text{set}(obs)) = 27$ observation values (26 lower case characters and 1 whitespace character), and run for 200 iterations with the default value for `tol`. Generally speaking, if you converge to a log probability greater than -21550 , then you have reached an acceptable set of parameters for this dataset.

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 required for languages other than English.

```
>>> for i in xrange(len(h.B)):
>>>     print u"%0}, {1:0.4f}, {2:0.4f}"%format(symbols[i], h.B[i,0], h.B[←
    [i,1])
, 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
h, 0.0806, 0.0070
k, 0.0031, 0.0005
j, 0.0040, 0.0000
m, 0.0360, 0.0000
l, 0.0569, 0.0001
o, 0.0009, 0.1331
n, 0.1207, 0.0000
q, 0.0015, 0.0000
p, 0.0345, 0.0000
s, 0.1195, 0.0000
r, 0.1062, 0.0000
u, 0.0000, 0.0546
t, 0.1600, 0.0000
w, 0.0242, 0.0000
v, 0.0185, 0.0000
```

```
y, 0.0147, 0.0058
x, 0.0022, 0.0000
z, 0.0010, 0.0000
```

What do you notice about the second column of B ? It seems that the HMM has detected a vowel state and a consonant state, without any prior input from an English speaker. Interestingly, the whitespace character is grouped together with the vowels. A HMM can also detect the vowel/consonant distinction in other languages.

Problem 8. Repeat the previous calculation with 3 hidden states and again with 4 hidden states. Interpret/explain your results.

Now we turn to the Russian file `WarAndPeace.txt`, which is a small subset of the book *War and Peace* by Tolstoy.

```
>>> symbols, obs = prep_data('WarAndPeace.txt')
```

Problem 9. Repeat the calculations for 2, and 3 hidden states for `WarAndPeace.txt`. Interpret/explain your results. Which Cyrillic characters appear to be vowels?

19

Gaussian Mixture Models

Lab Objective: *Understand the formulation of Gaussian Mixture Models (GMMs) and how to estimate GMM parameters.*

You've already seen GMMs as the observation distribution in certain continuous density HMMs. Here, we will discuss them further and learn how to estimate their parameters, given data.

The main idea behind a mixture model is contained in the name, i.e. it is a *mixture* of different models. What do we mean by a mixture? A mixture model is composed of K *components*, each component being responsible for a portion of the data. The responsibilities of these components are represented by mixture *weights* w_i , for $i = 1, \dots, k$. As you may have guessed, these weights are nonnegative and sum to 1. Thus component j is responsible for $100 \cdot w_j$ percent of the data generated by the model.

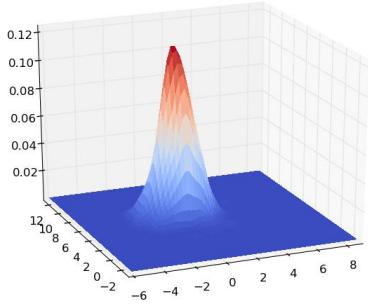
Each component is itself a probability distribution. In a GMM, each component is specifically a Gaussian (multivariate normal) distribution. Thus we additionally have parameters μ_i, Σ_i for $i = 1, \dots, K$, i.e. a mean and covariance for each component in the GMM. It is important here to keep in mind 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. In the first case, we would simply have a different multivariate normal distribution, whereas in the second case we have a mixture. Refer to Figure ?? for a visualization of this.

Thus, a fully defined GMM has parameters $\lambda = (w, \mu, \Sigma)$. The density of a GMM is given by $\mathbb{P}(x|\lambda) = \sum_{i=1}^K w_i \mathcal{N}(x; \mu_i, \Sigma_i)$ where

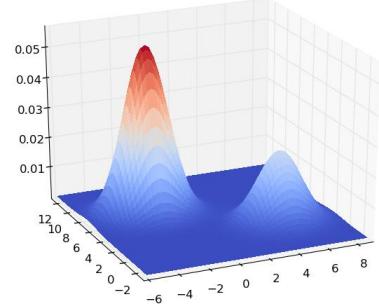
$$\mathcal{N}(x; \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{\frac{K}{2}} |\Sigma_i|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1} (x-\mu_i)}$$

Problem 1. Write a function to evaluate the density of a normal distribution at a point x , given parameters μ and Σ . Include the option to return the log of this probability, but be sure to do it intelligently! Also write a function that computes the density of a GMM at a point x , given the parameters λ , along with the log option.

Throughout this lab, we will build a GMM class with various methods. We will outline this now.



(a) Sum of weighted multivariate normal random variables.



(b) Weighted mixture of multivariate normal random variables.

Problem 2. Write the skeleton of a GMM class. In the `__init__` method, it should accept the non-null parameter `n_components`, as well as parameters for the weights, means, and covariance matrices which define the GMM. Include a function to generate data from a fully defined GMM (you may use your code from the CDHMM lab for this), as well as the density function you recently defined.

The main focus of this lab will be to estimate the parameters of a GMM, given observed multivariate data $Y = y_1, y_2, \dots, y_T$. This can be done via Gibbs sampling, as well as with EM (Expectation Maximization). We choose the latter approach for this lab. To do this, we must compute the probability of an observation being from each component of a GMM with parameters $\lambda^{(n)} = (w_i^{(n)}, \mu_i^{(n)}, \Sigma_i^{(n)})$. This is simply

$$\mathbb{P}(x_t = i | y_t, \lambda) \propto w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$$

Just as with HMMs, we refer to these probabilities as $\gamma_t(i)$, and this is the *E*-step in the algorithm. This might seem straightforward, except this direct computation will likely lead to numerical issues. Instead, we work in the log space, which means we have to be a bit more careful.

It is feasible (and occurs quite often) that each term $w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$ is 0, because of underflow in the computation of the multivariate normal density. Letting $l_i^{(n)} = \ln w_i^{(n)} + \ln \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$, we can compute these probabilities more carefully, as follows:

$$\begin{aligned} \mathbb{P}(x_t = i | y_t, \lambda) &= \frac{e^{l_i}}{\sum_{j=1}^K e^{l_j}} \\ &= \frac{e^{l_i} e^{-\max_k l_k}}{\sum_{j=1}^K e^{l_j} e^{-\max_k l_k}} \\ &= \frac{e^{l_i - \max_k l_k}}{\sum_{j=1}^K e^{l_j - \max_k l_k}} \end{aligned}$$

which will effectively avoid underflow problems.

Problem 3. Add a method to your class to compute $\gamma_t(i)$ for $t = 1, \dots, T$ and $i = 1, \dots, K$. Don't forget to do this intelligently to avoid underflow!

Given our matrix γ , we can reestimate our weights, means, and covariance matrices as follows:

$$\begin{aligned} w_i^{(n+1)} &= \sum_{t=1}^T \gamma_t(i) \\ \mu_i^{(n+1)} &= \frac{\sum_{t=1}^T \gamma_t(i) y_t}{\sum_{t=1}^T \gamma_t(i)} \\ \Sigma_i^{(n+1)} &= \frac{\sum_{t=1}^T \gamma_t(i) (y_t - \mu_i^{(n+1)}) (y_t - \mu_i^{(n+1)})^T}{\sum_{t=1}^T \gamma_t(i)} \end{aligned}$$

for $i = 1, \dots, K$. These updates are the M -step in the algorithm.

Problem 4. Add methods to your class to update w, μ and Σ as described above.

With the above work, we are almost ready to complete our class. To train, we will randomly initialize our parameters λ , and then iteratively update them as above.

Problem 5. Add a method to initialize λ . Do this intelligently, i.e. your means should not be far from your actual data used for training, and your covariances should neither be too big nor too small. Your weights should roughly be equal, and still sum to 1. Also add a method to train your model, as described previously, iterating until convergence within some tolerance.

We will use our work to train the “Mickey Mouse” GMM, which has parameters

$$\begin{aligned} w &= [0.7 \quad 0.15 \quad 0.15] \\ \mu_1 &= [0.0 \quad 0.0] \\ \mu_2 &= [-1.5 \quad 2.0] \\ \mu_3 &= [1.5 \quad 2.0] \\ \Sigma_1 &= I_3 \\ \Sigma_2 &= 0.25 \cdot I_3 \\ \Sigma_3 &= 0.25 \cdot I_3 \end{aligned}$$

To look at this GMM, we will evaluate the density at each point on a grid, as follows:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(-3, 3, 0.1)
>>> y = np.arange(-2, 3, 0.1)
>>> X, Y = np.meshgrid(x, y)
>>> N, M = X.shape
>>> immat = np.array([[model.dgmm(np.array([X[i,j],Y[i,j])) for j in xrange(M)] for i in xrange(N)])
```

```
>>> plt.imshow(immat, origin='lower')
>>> plt.show()
```

See Figure 19.2 for this plot.

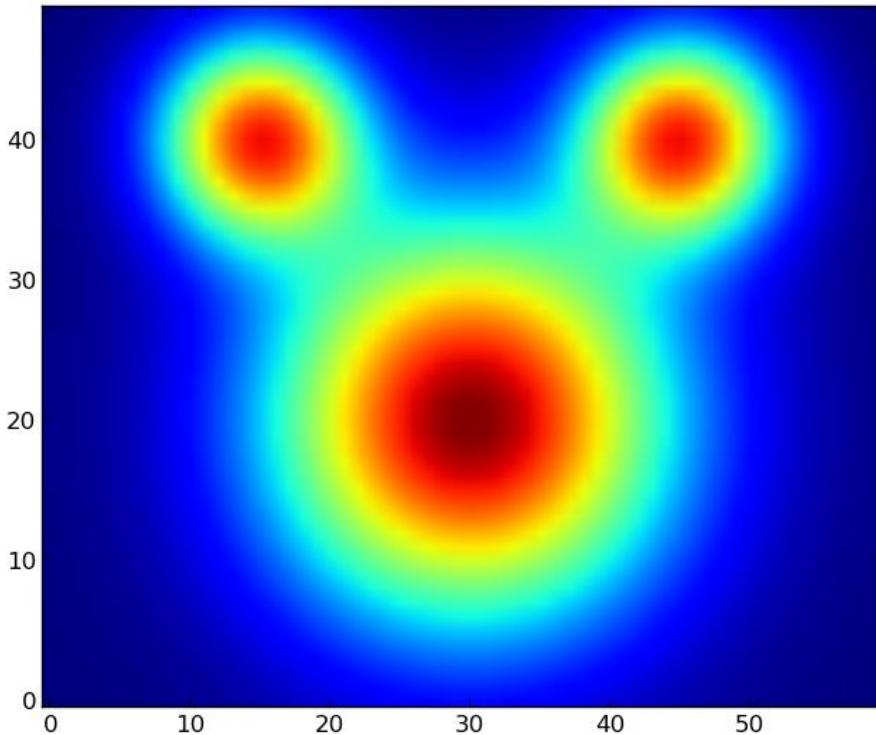


Figure 19.2: Density of true “Mickey Mouse” GMM.

Problem 6. Generate 750 samples from the above mixture model. Using just the drawn samples, retrain your model. Evaluate and plot your density on the grid used above. How similar is your density to the original?

How close is our trained model to the original one? We can use the symmetric Kullback-Liebler divergence to measure the distance between two probability distributions with densities $p(x)$ and $p'(x)$:

$$SKL(p, p') = \left| \frac{1}{2} \int p(x) \ln \frac{p(x)}{p'(x)} dx + \frac{1}{2} \int p'(x) \ln \frac{p'(x)}{p(x)} dx \right|$$

We cannot analytically compute this, so we use a Monte Carlo approximation, which uses the fact that

$$\frac{1}{N} \sum_{i=1}^N f(x_i) \rightarrow \int f(x)p(x)dx$$

as $N \rightarrow \infty$, assuming that each $x_i \sim p$. Then we have the following approximation of the symmetric KL divergence:

$$SKL(p, p') \approx \frac{1}{2N} \left| \sum_{i=1}^N \ln \frac{p(x_i)}{p'(x_i)} + \sum_{i=1}^N \ln \frac{p'(x'_i)}{p(x'_i)} \right|$$

where $x_i \sim p$ and $x'_i \sim p'$, for large N .

Problem 7. Write a function to compute the approximate the SKL of two GMMs. Compute the SKL between a randomly initialized GMM and the known GMM. Compute the SKL between the trained GMM and the known GMM. Is our trained model a good fit?

20

Speech Recognition using CDHMMs

Lab Objective: *Understand how speech recognition via CDHMMs works, and implement a simplified speech recognition system.*

20.0.1 Continuous Density Hidden Markov Models

Some of the most powerful applications of HMMs (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), and they have two standard formulations: Gaussian HMMs and Gaussian Mixture Model HMMs (GMMHMMs). In fact, the former is a special case of the latter, so we will just discuss GMMHMMs in this lab.

In order to understand GMMHMMs, we need to be familiar with a particular continuous, multivariate distribution called a *mixture of Gaussians*. A mixture of Gaussians is a distribution composed of several Gaussian (or Normal) distributions with corresponding weights. Such a distribution is parameterized by the number of mixture components M , the dimension N of the normal distributions involved, a collection of component weights $\{c_1, \dots, c_M\}$ that are nonnegative and sum to 1, and a collection of mean and covariance parameters $\{(\mu_1, \Sigma_1), \dots, (\mu_M, \Sigma_M)\}$ for each Gaussian component. To sample from a mixture of Gaussians, one first chooses the mixture component i according to the probability weights $\{c_1, \dots, c_M\}$, and then one samples from the normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$. The probability density function for a mixture of Gaussians is given by

$$f(x) = \sum_{i=1}^M c_i N(x; \mu_i, \Sigma_i),$$

where $N(\cdot; \mu_i, \Sigma_i)$ denotes the probability density function for the normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$. See Figure 20.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

In a GMMHMM, we seek to model a hidden state sequence $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ and a corresponding observation sequence $\{O_1, \dots, O_T\}$, just as with discrete HMMs. The major difference, of course, is that each observation O_t is a real-valued vector of length K distributed according to a mixture of Gaussians with M components. The parameters for such a model include the initial state distribution π and the state transition matrix A (just as with discrete HMMs). Additionally, for each state $i = 1, \dots, N$, we have component weights $\{c_{i,1}, \dots, c_{i,M}\}$, component means $\{\mu_{i,1}, \dots, \mu_{i,M}\}$, and component covariance matrices $\{\Sigma_{i,1}, \dots, \Sigma_{i,M}\}$.

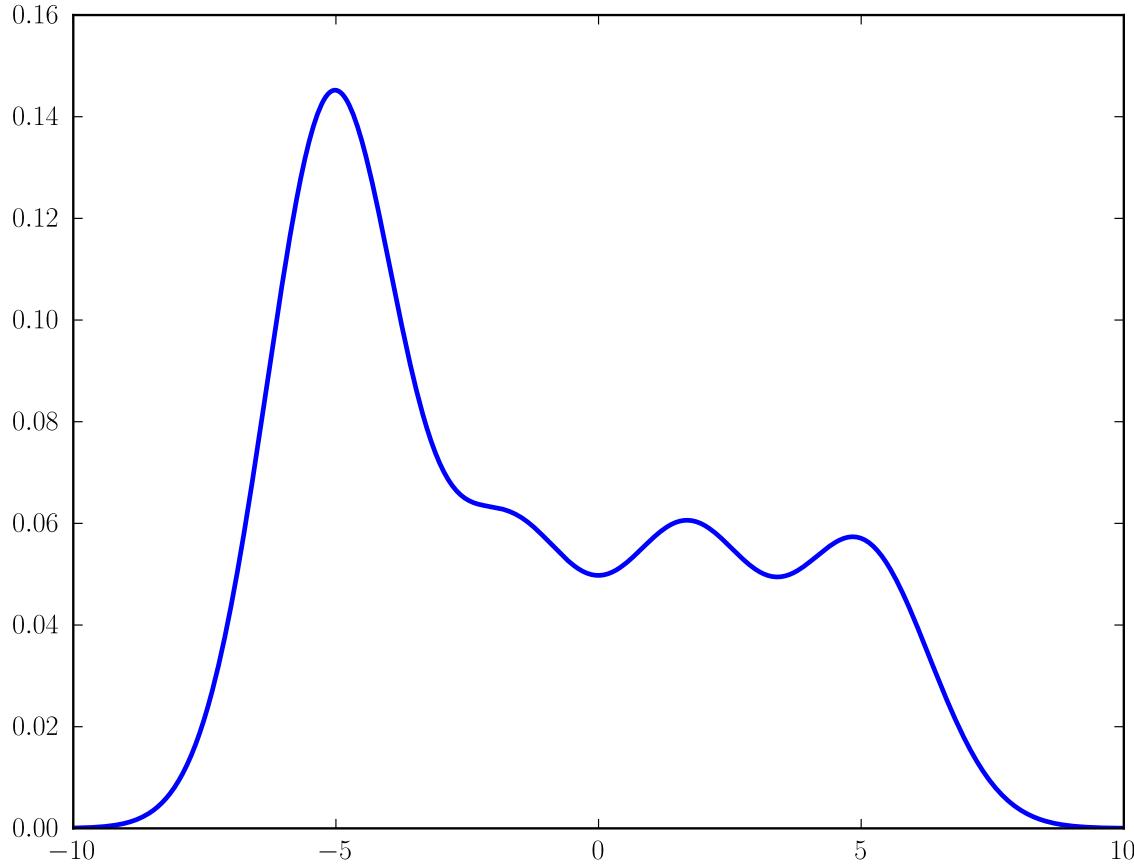


Figure 20.1: The probability density function of a mixture of Gaussians with four components.

Let's define a full GMMHMM with $N = 2$ states, $K = 3$, and $M = 3$ components.

```
>>> import numpy as np
>>> A = np.array([[.65, .35], [.15, .85]])
>>> pi = np.array([.8, .2])
>>> weights = np.array([[.7, .2, .1], [.1, .5, .4]])
>>> means1 = np.array([[0., 17., -4.], [5., -12., -8.], [-16., 22., 2.]])
>>> means2 = np.array([[-5., 3., 23.], [-12., -2., 14.], [15., -32., 0.]])
>>> means = np.array([means1, means2])
>>> covars1 = np.array([5*np.eye(3), 7*np.eye(3), np.eye(3)])
>>> covars2 = np.array([10*np.eye(3), 3*np.eye(3), 4*np.eye(3)])
>>> covars = np.array([covars1, covars2])
>>> gmmhmm = [A, weights, means, covars, pi]
```

We can draw a random sample from the GMMHMM corresponding to the second state as follows:

```
>>> sample_component = np.argmax(np.random.multinomial(1, weights[1,:]))
```

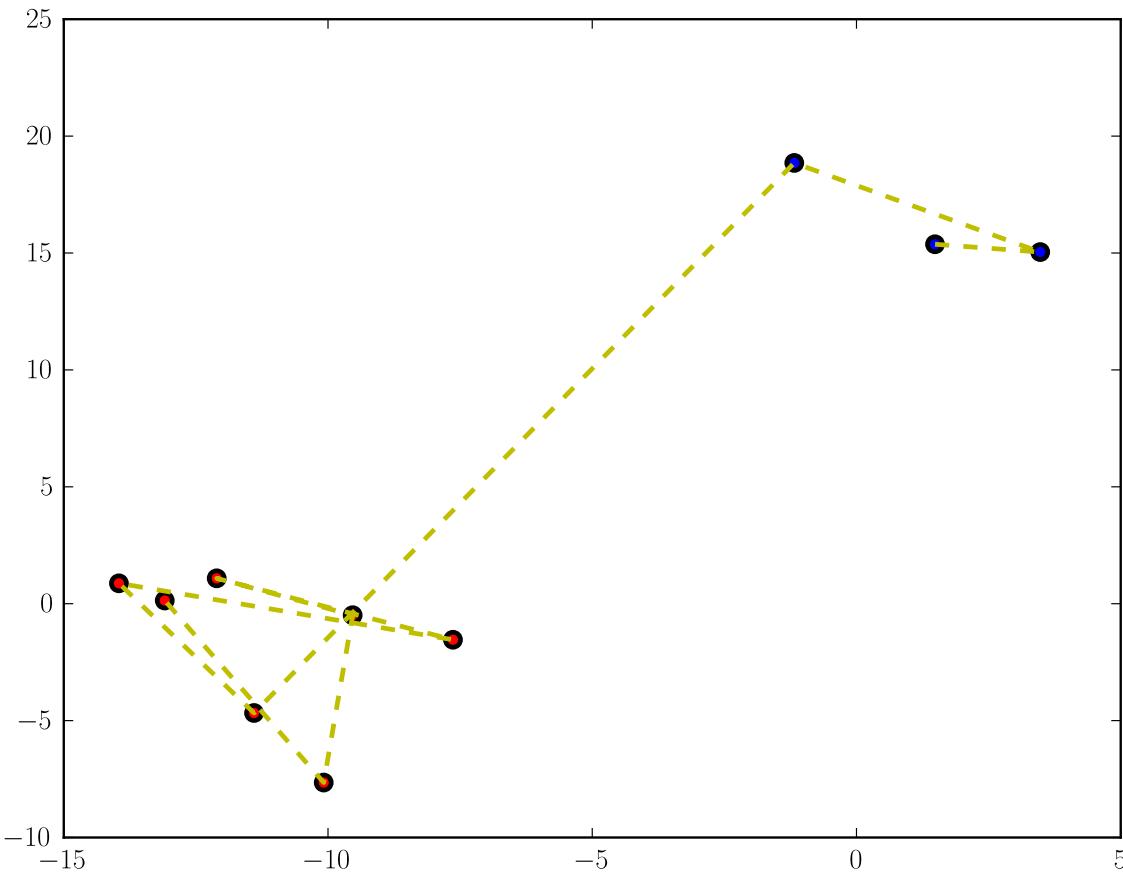


Figure 20.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

```
>>> sample = np.random.multivariate_normal(means[1, sample_component, :], ←
    covars[1, sample_component, :, :])
```

Figure 20.2 shows an observation sequence generated from a GMMHMM with one mixture component and two states.

Problem 1. Write a function which accepts a GMMHMM in the format above as well as an integer n_sim , and which simulates the GMMHMM process, generating n_sim different observations. Do so by implementing the following function declaration.

```
def sample_gmmhmm(gmmhmm, n_sim):
    """
    Simulate sampling from a GMMHMM.
```

```

Returns
-----
states : ndarray of shape (n_sim,)
    The sequence of states
obs : ndarray of shape (n_sim, K)
    The generated observations (column vectors of length K)
"""
pass

```

The classic problems for which we normally use discrete observation HMMs can also be solved by using CDHMMs, though with continuous observations it is much more difficult to keep things numerically stable. We will not have you implement any of the three problems for CDHMMs yourself; instead, you will use a stable module we will provide for you. Note, however, that the techniques for solving these problems are still based on the forward-backward algorithm; the implementation may be trickier, but the mathematical ideas are virtually the same as those for discrete HMMs.

Speech Recognition and Hidden Markov Models

Hidden Markov Models are the basis of modern speech recognition systems. However, a fair amount of signal processing must precede the HMM stage, and there are other components of speech recognition, such as language models, that we will not address in this lab.

The basic signal processing and HMM stages of the speech recognition system that we develop in this lab can be summarized as follows: The audio to be processed is divided into small frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these intervals. We can then take this framed signal and, through a series of transformations, represent it by mel-frequency cepstral coefficients (MFCCs), keeping only the first K (say $K = 10$). Viewing these MFCCs as continuous observations in \mathbb{R}^K , we can train a GMMHMM on sequences of MFCCs for a given word, spoken multiple times. Doing this for several words, we have a collection of GMMHMMs, one for each word. Given a new speech signal, after framing and decomposing it into its MFCC array, we can score the signal against each GMMHMM, returning the word whose GMMHMM scored the highest.

Industrial-grade speech recognition systems do not train a GMMHMM for each word in a vocabulary (that would be ludicrous for a large vocabulary), but rather on *phonemes*, or distinct sounds. The English language has 44 phonemes, yielding 44 different GMMHMMs. As you could imagine, this greatly facilitates the problem of speech recognition. Each and every word can be represented by some combination of these 44 distinct sounds. By correctly classifying a signal by its phonemes, we can determine what word was spoken. Doing so is beyond the scope of this lab, so we will simply train GMMHMMs on five words/phrases: biology, mathematics, political science, psychology, and statistics.

Problem 2. Obtain 30 (or more) recordings for each of the words/phrases *mathematics*, *biology*, *political science*, *psychology*, and *statistics*. These audio samples should be 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. Load the recordings into Python using `scipy.io.wavfile.read`.

If the audio files have two channels, average these channels to obtain an array of length 88200 for each sample. Extract the MFCCs from each sample using code from the file `MFCC.py`:

```
>>> import MFCC
>>> # assume sample is an array of length 88200
>>> mfccs = MFCC.extract(sample)
```

Store the MFCCs for each word in a separate list. You should have five lists, each containing 50 MFCC arrays, corresponding to each of the five words under consideration.

For a specific word, given enough distinct samples of that word (decomposed into MFCCs), we can train a GMMHMM. Recall, however, that the training procedure does not always produce a very effective model, as it 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, we will use the file we have provided called `gmmhmm.py`, as this is a stable implementation of GMMHMM algorithms. To facilitate random restarts, we need a function to provide initializations for the initial state distribution and the transition matrix.

Let `samples` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Using a function `initialize()` that returns a random initial state distribution and (row-stochastic) transition matrix, we can train a GMMHMM with 5 states and 3 mixture components and view its log-likelihood as follows:

```
>>> import gmmhmm
>>> startprob, transmat = initialize(5)
>>> model = gmmhmm.GMMHMM(n_components=5, n_mix=3, transmat=transmat, startprob=startprob, cvtype='diag')
>>> # these values for covars_prior and var should work well for this problem
>>> model.covars_prior = 0.01
>>> model.fit(samples, init_params='mc', var=0.1)
>>> print model.logprob
```

Problem 3. Partition each list of MFCCs into a training set of 20 samples, and a test set of the remaining 10 samples.

Using the training sets, train a GMMHMM on each of the words from the previous problem with at least 10 random restarts, keeping the best model for each word (the one with the highest log-likelihood). This process may take several minutes. Since you will not want to run this more than once, you will want to save the best model for each word to disk using the `pickle` module so that you can use it later.

Given a trained model, we would like to compute the log-likelihood of a new sample. Letting `obs` be an array of MFCCs for a speech sample we do this as follows:

```
>>> score = model.score(obs)
```

We classify a new speech sample by scoring it against each of the 5 trained GMMHMMs, and returning the word corresponding to the GMMHMM with the highest score.

Problem 4. Classify the 10 test samples for each word. How does your system perform? Which words are the hardest to correctly classify? Make a dictionary containing the accuracy of the classification of your five testing sets. Specifically, the words/phrases will be the keys, and the values will be the percent accuracy.

21

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

$$\mathbb{P}(x_1, \dots, x_n | \mathbf{y}).$$

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

$$\mathbb{P}(x_i | \mathbf{x}_{-i}, \mathbf{y})$$

where $\mathbf{x}_{-i} = x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$.

Algorithm 21.1 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  $x \sim \mathbb{P}(x_i | \mathbf{x}_{-i}, \mathbf{y})$ 
6:       Fix  $x_i = x$ 
7:      $\mathbf{x}^{(k)} = (x_1, x_2, \dots, x_n)$ 

```

A Gibbs sampler proceeds according to Algorithm 21.1. 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

$$\mathbb{P}(x_1, \dots, x_n | \mathbf{y}).$$

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.csv`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean μ and variance σ^2 . Because we are unsure of the true value of μ and σ^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 μ and σ^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

$$\begin{aligned}\mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) &= N(\mu^*, (\sigma^*)^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) &= IG(\alpha^*, \beta^*),\end{aligned}$$

where

$$\begin{aligned}(\sigma^*)^2 &= \left(\frac{1}{\tau^2} + \frac{N}{\sigma^2} \right)^{-1} \\ \mu^* &= (\sigma^*)^2 \left(\frac{\nu}{\tau^2} + \frac{1}{\sigma^2} \sum_{i=1}^N y_i \right) \\ \alpha^* &= \alpha + \frac{N}{2} \\ \beta^* &= \beta + \frac{1}{2} \sum_{i=1}^N (y_i - \mu)^2\end{aligned}$$

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling μ and sampling σ^2 . We can sample from a normal distribution and an inverse gamma distribution as follows:

```
>>> from math import sqrt
>>> from scipy.stats import norm
>>> from scipy.stats import invgamma
>>> mu = 0. # the mean
>>> sigma2 = 9. # the variance
>>> normal_sample = norm.rvs(mu, scale=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. Implement a Gibbs sampler for the exam scores problem using the following function declaration.

```
def gibbs(y, nu, tau2, alpha, beta, n_samples):
    """
    Assuming a likelihood and priors
    y_i ~ N(mu, sigma2),
    mu ~ N(nu, tau2),
    sigma2 ~ IG(alpha, beta),
    sample from the posterior distribution
    P(mu, sigma2 | y, nu, tau2, alpha, beta)
    using a gibbs sampler.

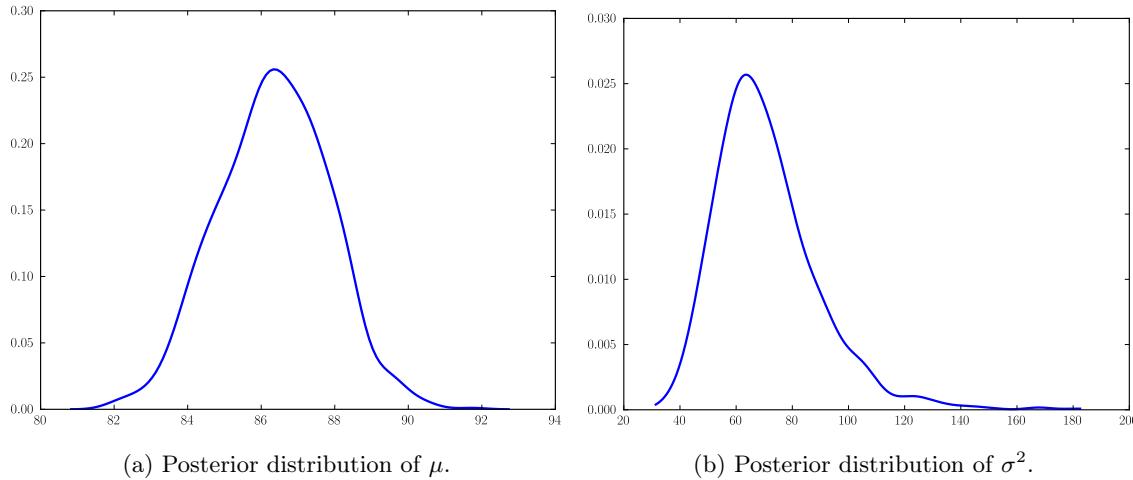
    Parameters
    -----
    y : ndarray of shape (N,)
        The data
    nu : float
        The prior mean parameter for mu
    tau2 : float > 0
        The prior variance parameter for mu
    alpha : float > 0
        The prior alpha parameter for sigma2
    beta : float > 0
        The prior beta parameter for sigma2
    n_samples : int
        The number of samples to draw

    Returns
    -----
    samples : ndarray of shape (n_samples,2)
        1st col = mu samples, 2nd col = sigma2 samples
    """
    pass
```

Test it with priors $\nu = 80, \tau^2 = 16, \alpha = 3, \beta = 50$, collecting 1000 samples. Plot your samples of μ and your samples of σ^2 . How long did it take for each to converge? It should have been very quick.

We'd like to look at the posterior marginal distributions for μ and σ^2 . To plot these from the samples, we will use a kernel density estimator. If our samples of μ are called `mu_samples`, then we can do this as follows:

```
>>> import numpy as np
>>> from scipy.stats import gaussian_kde
>>> import matplotlib.pyplot as plt
>>> mu_kernel = gaussian_kde(mu_samples)
>>> x_min = min(mu_samples) - 1
```

Figure 21.1: Posterior marginal probability densities for μ and σ^2 .

```
>>> x_max = max(mu_samples) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
```

Problem 2. Plot the kernel density estimators for the posterior distributions of μ and σ^2 . You should get plots similar to those in Figure 21.1.

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 Θ denotes our parameters (in our case μ and σ^2) and λ denotes our prior parameters (in our case ν, τ^2, α , and β).

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.

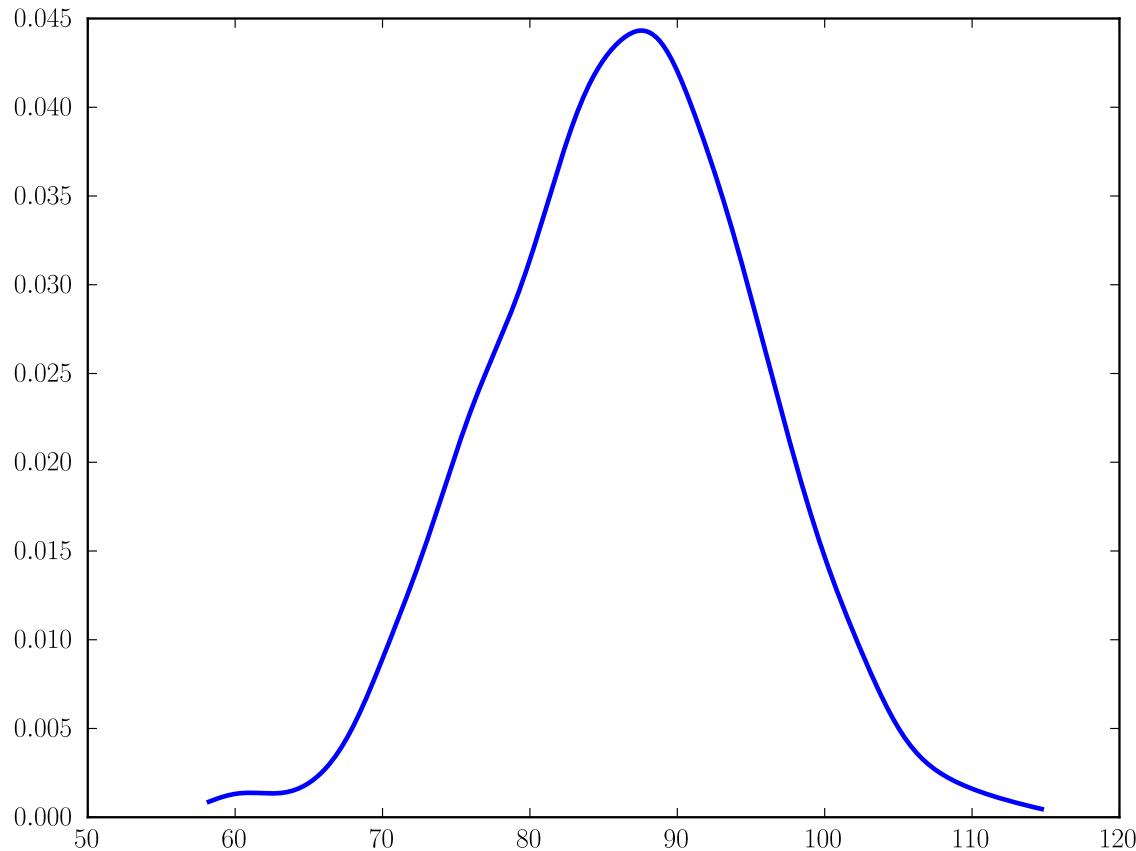


Figure 21.2: Predictive posterior distribution of exam scores.

Problem 3. Use your samples of μ and σ^2 to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. It should resemble the plot in Figure 21.2.

Latent Dirichlet Allocation

Gibbs sampling can be applied to an interesting problem in language processing: 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 ϕ_k over the vocabulary, each with a Dirichlet prior β . What this means is that $\phi_{k,v}$ 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 θ_m over the topics is drawn from a Dirichlet distribution with parameter α . Thus $\theta_{m,k}$ is the probability that document m is assigned the 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 θ_m , and then we draw a word $w_{m,n}$ from the categorical distribution $\phi_{z_{m,n}}$. Throughout this implementation, we assume α and β are scalars. 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 $w_{m,n} \sim \text{Cat}(\phi_{z_{m,n}})$ for $1 \leq n \leq N_m$.

What we end up with here for document m is n words which represent the document. Note that these words are *not* 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 21.3.

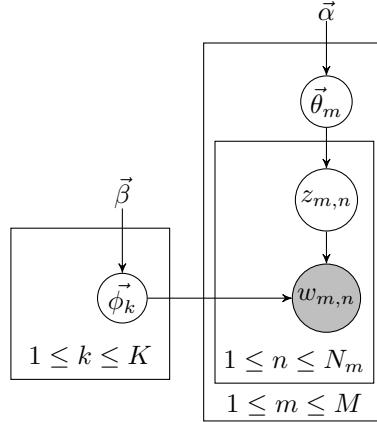


Figure 21.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables $w_{m,n}$ are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each ϕ_k and each θ_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 $\mathbb{P}(\mathbf{z}|\mathbf{w}, \alpha, \beta)$, where \mathbf{w} is the collection words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$, the conditional posterior distributions

$$\mathbb{P}(z_{m,n} = k | \mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta)$$

have nice, closed form solutions, making them easy to sample from.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k | \mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta) \propto \frac{(n_{(k,m,\cdot)}^{-} + \alpha)(n_{(k,\cdot,w_{m,n})}^{-} + \beta)}{n_{(k,\cdot,\cdot)}^{-} + V\beta}$$

where

$$\begin{aligned}
 n_{(k,m,\cdot)} &= \text{the number of words in document } m \text{ assigned to topic } k \\
 n_{(k,\cdot,v)} &= \text{the number of times term } v = w_{m,n} \text{ is assigned to topic } k \\
 n_{(k,\cdot,\cdot)} &= \text{the number of times topic } k \text{ is assigned in the corpus} \\
 n_{(k,m,\cdot)}^{-(m,n)} &= n_{(k,m,\cdot)} - \mathbf{1}_{z_{m,n}=k} \\
 n_{(k,\cdot,v)}^{-(m,n)} &= n_{(k,\cdot,v)} - \mathbf{1}_{z_{m,n}=k} \\
 n_{(k,\cdot,\cdot)}^{-(m,n)} &= n_{(k,\cdot,\cdot)} - \mathbf{1}_{z_{m,n}=k}
 \end{aligned}$$

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 θ and ϕ .

We have provided for you the structure of a Python object LDACGS with several methods. The object is already defined to have attributes `n_topics`, `documents`, `vocab`, `alpha`, and `beta`, where `vocab` is a list of strings (terms), and `documents` is a list of dictionaries (a dictionary for each document). Each entry in dictionary m is of the form $n : w$, where w is the index in `vocab` of the n^{th} word in document m .

Throughout this lab we will guide you through writing several more methods in order to implement the Gibbs sampler. The first step is to initialize our assignments, and create the count matrices $n_{(k,m,\cdot)}$, $n_{(k,\cdot,v)}$ and vector $n_{(k,\cdot,\cdot)}$.

Problem 4. Complete the method `initialize`. By randomly assigning initial topics, fill in the count matrices and topic assignment dictionary. In this method, you will initialize the count matrices (among other things). Note that the notation provided in the code is slightly different than that used above. Be sure to understand how the formulae above connect with the code.

To be explicit, you will need to initialize nmz , nzw , and nz to be zero arrays of the correct size. Then, in the second for loop, you will assign z to be a random integer in the correct range of topics. In the increment step, you need to figure out the correct indices to increment by one for each of the three arrays. Finally, assign `topics` as given.

The next method we need to write fully outlines a sweep of the Gibbs sampler.

Problem 5. Complete the method `_sweep`, which needs to iterate through each word of each document. It should call on the method `_conditional` to get the conditional distribution at each iteration.

Note that the first part of this method will undo what the `initialize` method did. Then we will use the conditional distribution (instead of the uniform distribution we used previously) to pick a more accurate topic assignment. Finally, the latter part repeats what we did in `initialize`, but does so using this more accurate topic assignment.

We are now prepared to write the full Gibbs sampler.

Problem 6. Complete the method `sample`. The argument `filename` is the name and location of a .txt file, where each line is considered a document. The corpus is built by method `buildCorpus`, and stopwords are removed (if argument `stopwords` is provided). Burn in the Gibbs sampler, computing and saving the log-likelihood with the method `_loglikelihood`. After the burn in, iterate further, accumulating your count matrices, by adding `nzw` and `nmz` to `total_nzw` and `total_nmz` respectively, where you only add every $sample_rate^{th}$ iteration. Also save each log-likelihood.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on Ronald Reagan's State of the Union addresses.

Problem 7. Create an `LDACGS` object with 20 topics, letting `alpha` and `beta` be the default values. Load in the stop word list provided. Run the Gibbs sampler, with a burn in of 100 iterations, accumulating 10 samples, only keeping the results of every 10th sweep. Plot the log-likelihoods. How long did it take to truly burn in?

We can estimate the values of each ϕ_k and each θ_m as follows:

$$\hat{\theta}_{m,k} = \frac{n_{(k,m,\cdot)} + \alpha}{K \cdot \alpha + \sum_{k=1}^K n_{(k,m,\cdot)}}$$

$$\hat{\phi}_{k,v} = \frac{n_{(k,\cdot,v)} + \beta}{V \cdot \beta + \sum_{v=1}^V n_{(k,\cdot,v)}}$$

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions ϕ_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 8. Using the methods described above, examine the topics for Reagan's addresses. As best as you can, come up with labels for each topic. Note that if $ntopics = 20$ and $n = 10$, we will get the top 10 words that represent each of the 20 topics. What you will want to do for each topic is decide what these ten words jointly represent represent. Save your topic labels in a list or an array.

We can use $\hat{\theta}$ to find the 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 paragraphs are most centered on the Cold War.

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.

22

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 task in many different applications found throughout the sciences. When these distributions are complicated, as is often the case when modeling real-world problems, direct sampling methods can become difficult, as they might involve computing high-dimensional integrals. The Metropolis algorithm is an effective method to sample from many distributions, requiring only that we be able to evaluate the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

The Metropolis algorithm is an MCMC sampling method which generates a sequence of random variables, similar to Gibbs sampling. These random variables form a Markov Chain whose invariant distribution is equal to the distribution from which we wish to sample. Suppose that $h : \mathbb{R}^n \rightarrow \mathbb{R}$ is the probability density function of distribution, and suppose that $f(\theta) = c \cdot h(\theta)$ for some nonzero constant c (in practice, we assume that f is an easy function to evaluate, while h is difficult). Let $Q : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ be a symmetric *proposal function* (so that $Q(\cdot, y)$ is a probability density function for all $y \in \mathbb{R}^n$, and $Q(x, y) = Q(y, x)$ for all $x, y \in \mathbb{R}^n$) and let $A : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ be an *acceptance function* defined by

$$A(x, y) = \min\left(1, \frac{f(y)}{f(x)}\right).$$

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 22.1. The Metropolis algorithm can be interpreted as follows: given our current state y , we propose a new state according to the distribution $Q(\cdot, y)$. We then accept or reject it according to A . We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h . Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

$$\log A(x, y) = \min(0, \log f(x) - \log f(y)).$$

Algorithm 22.1 Metropolis Algorithm

```

1: procedure METROPOLIS ALGORITHM
2:   Choose initial point  $x_0$ .
3:   for  $t = 1, 2, \dots$  do
4:     Draw  $x' \sim Q(\cdot, x_{t-1})$ 
5:     Draw  $a \sim \text{unif}(0, 1)$ 
6:     if  $a \leq A(x', x_{t-1})$  then
7:        $x_t = x'$ 
8:     else
9:        $x_t = x_{t-1}$ 
10:    Return  $x_1, x_2, x_3, \dots$ 
```

Let's apply the Metropolis algorithm to a simple example of Bayesian analysis. Consider the problem of computing the posterior distribution over the mean μ and variance σ^2 of a normal distribution for which we have N data points y_1, \dots, y_N . For concreteness, we use the data in `examscores.csv` and we assume the prior distributions

$$\begin{aligned}\mu &\sim N(\mu_0 = 80, \sigma_0^2 = 16) \\ \sigma^2 &\sim IG(\alpha = 3, \beta = 50).\end{aligned}$$

In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 | y_1, \dots, y_N) = \frac{p(\mu)p(\sigma^2) \prod_{i=1}^N N(y_i | \mu, \sigma^2)}{\int_{-\infty}^{\infty} \int_0^{\infty} p(\mu)p(\sigma^2) \prod_{i=1}^N N(y_i | \mu, \sigma^2) d\sigma^2 d\mu}$$

. However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to μ and σ^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:

$$Q(x, y) = N(x | y, sI),$$

where I is the 2×2 identity matrix and s is some positive scalar. Let's create these functions in Python:

```

import numpy as np
from math import sqrt, exp, log
import scipy.stats as st
from matplotlib import pyplot as plt
from scipy.stats import gaussian_kde

# load in the data
scores = np.loadtxt('examscores')

# initialize the hyperparameters
alpha = 3
beta = 50
mu0 = 80
sig20 = 16
```

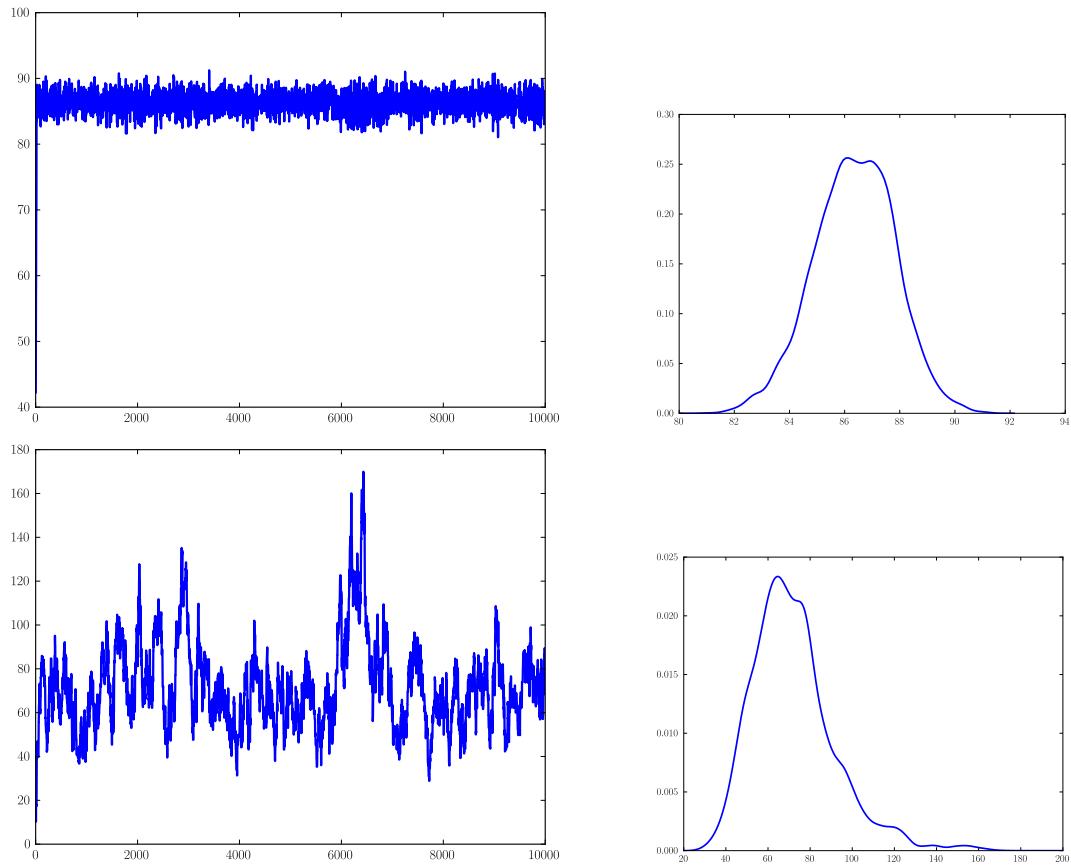


Figure 22.1: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

```
# initialize the prior distributions
muprior = st.norm(loc=mu0, scale=sqrt(sig20))
sig2prior = st.invgamma(alpha,scale=beta)

# define the proposal function
def proposal(y, s):
    return st.multivariate_normal.rvs(mean=y, cov=s*np.eye(len(y)))

# define the log of the proportional density
def propLogDensity(x):
    return muprior.logpdf(x[0])+sig2prior.logpdf(x[1])+st.norm.logpdf(scores,←
        loc=x[0],scale=sqrt(x[1])).sum()
```

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 densities of the samples and the proportion of proposed samples that were accepted. Study the implementation below to make sure you understand the process:

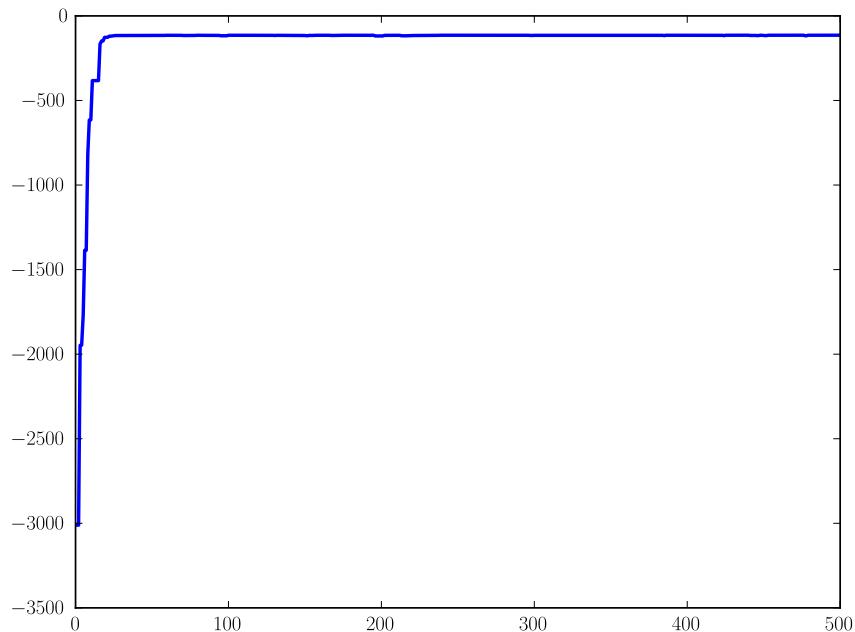


Figure 22.2: Log densities of the first 500 Metropolis samples.

```
def metropolis(x0, s, n_samples):
    """
    Use the Metropolis algorithm to sample from posterior.

    Parameters
    -----
    x0 : ndarray of shape (2,)
        The first entry is mu, the second entry is sigma2
    s : float > 0
        The standard deviation parameter for the proposal function
    n_samples : int
        The number of samples to generate

    Returns
    -----
    draws : ndarray of shape (n_samples, 2)
        The MCMC samples
    logprobs : ndarray of shape (n_samples)
        The log density of the samples
    accept_rate : float
        The proportion of proposed samples that were accepted
    """
    accept_counter = 0
    draws = np.empty((n_samples,2))
    logprob = np.empty(n_samples)
```

```

x = x0.copy()
for i in xrange(n_samples):
    xprime = proposal(x,s)
    u = np.random.rand(1)[0]
    if log(u) <= propLogDensity(xprime) - propLogDensity(x):
        accept_counter += 1
        x = xprime
    draws[i] = x
    logprob[i] = propLogDensity(x)
return draws, logprob, accept_counter/float(n_samples)

```

Now let's sample from the posterior. We will choose an initial guess of $\mu = 40$ and $\sigma^2 = 10$, and we will set $s = 20$. We draw 10000 samples as follows:

```

>>> draws, lprobs, rate = metropolis(np.array([40, 10], dtype=float), 20., ←
    10000)
>>> print "Acceptance Rate:", r
Acceptance Rate: 0.3531

```

We can evaluate the quality of our results by plotting the log probabilities, the μ samples, the σ^2 samples, and kernel density estimators for the marginal posterior distributions of μ and σ^2 . The code below will accomplish this task:

```

>>> # plot the first 500 log probs
>>> plt.plot(lprobs[:500])
>>> plt.show()
>>> # plot the mu samples
>>> plt.plot(draws[:,0])
>>> plt.show()
>>> # plot the sigma2 samples
>>> plt.plot(draws[:,1])
>>> plt.show()
>>> # build and plot KDE for posterior mu
>>> mu_kernel = gaussian_kde(draws[50:,0])
>>> x_min = min(draws[50:,0]) - 1
>>> x_max = max(draws[50:,0]) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
>>> # build and plot KDE for posterior sigma2
>>> sig_kernel = gaussian_kde(draws[50:,1])
>>> x_min = 20
>>> x_max = 200
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,sig_kernel(x))
>>> plt.show()

```

Your results should be close to those given in Figures 22.1 and 22.2.



Figure 22.3: Spin configuration from random initialization.

The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice Λ 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 σ , 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 100×100 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 100×100 array, with entries of ± 1 .

Problem 1. Write a function that initializes a spin configuration for an $n \times n$ lattice. It should return an $n \times n$ array, each entry of which is either 1 or -1 , chosen randomly. Test this for the grid described above, and plot the spin configuration using `matplotlib.pyplot.imshow`. It should look fairly random, as in Figure 22.3.

Problem 2. Write a function that computes the energy of a wrap-around $n \times n$ lattice with a given spin configuration, as described above. Make sure that you do not double count site pair interactions!

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 β , 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:

$$\begin{aligned} \frac{\mathbb{P}_\beta(\sigma^*)}{\mathbb{P}_\beta(\sigma)} &= \frac{e^{-\beta H(\sigma^*)}}{e^{-\beta H(\sigma)}} \\ &= e^{\beta(H(\sigma) - H(\sigma^*))} \end{aligned}$$

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 our acceptance probability would be

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases}$$

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 σ^* given σ , 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 $H(\sigma^*) = H(\sigma) + 2 \sum_{j:j \sim i} \sigma_i \sigma_j$.

Problem 3. Write a function that proposes a new spin configuration given the current spin configuration on an $n \times n$ lattice, as described above. This function simply needs to return a pair of indices (i, j) , chosen with probability $\frac{1}{n^2}$.

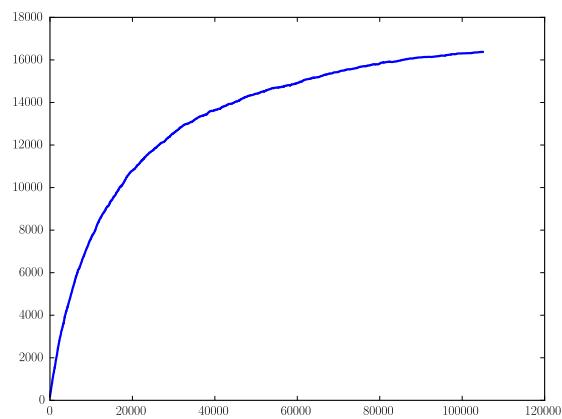
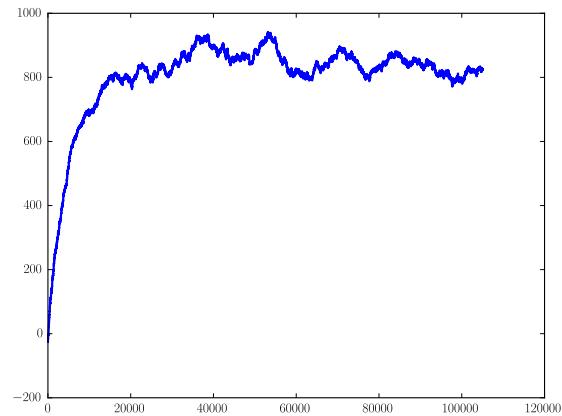
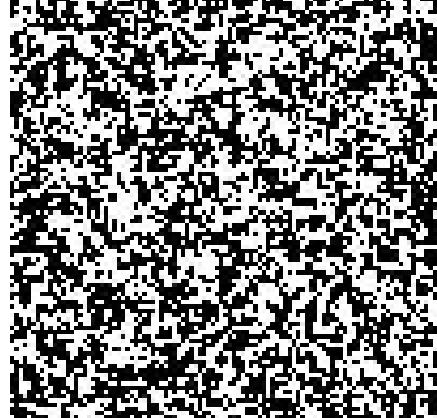
Problem 4. Write a function that computes the energy of a proposed spin configuration, given the current spin configuration, its energy, and the proposed spin flip site indices.

Problem 5. Write a function that accepts or rejects a proposed spin configuration, given the current configuration. It should accept the current energy, the proposed energy, and β , and should return a boolean.

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_β , which—as we explained previously—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 initializes a spin configuration for an $n \times n$ lattice as done previously, and then performs the Metropolis algorithm, choosing new spin configurations and accepting or rejecting them. It should burn in first, and then iterate $n_samples$ times, keeping every 100th sample (this is to prevent memory failure) and all of the above values for $-\beta H(\sigma)$ (keep the values even for the burn-in period). It should also accept β as an argument, allowing us to effectively adjust the temperature for the model.

Problem 7. Test your Metropolis sampler on a 100×100 grid, with 200000 iterations, with $n_samples$ large enough so that you will keep 50 samples, testing with $\beta = 1$ and then with $\beta = 0.2$. Plot the proportional log probabilities, and also plot a late sample from each test using `matplotlib.pyplot.imshow`. How does the ferromagnetic material behave differently with differing temperatures? Recall that β is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figures 22.4b and 22.4d.

(a) Proportional log probs when $\beta = 1$.(c) Proportional log probs when $\beta = 0.2$.(b) Spin configuration sample when $\beta = 1$.(d) Spin configuration sample when $\beta = 0.2$.

23

Principal Component Analysis and Latent Semantic Indexing

Lab Objective: *Understand the basics of principal component analysis and latent semantic indexing.*

Principal Component Analysis

Understanding the variance in complex data is one of the first tasks encountered in exploratory data analysis. For an example, consider the scatter plot displaying the sepal and petal lengths of 100 different irises shown in Figure 23.1. There are three distinct types of iris flowers present: *setosa*, *versicolor*, and *virginica*. Considering this data, we might ask how to best distinguish the different types of irises based on their given sepal and petal lengths. We can answer this question by finding the characteristic that causes the greatest variance in the data. (Greater variance implies a greater ability to distinguish between data points. If the variance is very small, the data are clustered tightly together, and it is difficult to distinguish well.)

Upon examination, we see that the petal length ranges between 3 and 7 cm, while the sepal length only ranges between 5 and 8 cm. We might be tempted to say that the most distinguishing aspect of irises is their petal length, but this is only considering the features of the data individually, and not collectively. The two features of the data are clearly correlated, and a more careful consideration would lead us to conclude that the most distinguishing aspect of irises is their overall size. Some irises are much larger than others, while the sepal and petal lengths stay roughly in proportion.

Principal Component Analysis (PCA) is a multivariate statistical tool used to orthogonally change the basis of a set of observations from the basis of original features (which may be correlated) into a basis of uncorrelated (in fact, orthonormal) variables called the *principal components*. It is a direct application of the singular value decomposition (SVD) from linear algebra. More specifically, the first principal component will account for the greatest variance in the set of observations, the second principal component will be orthogonal to the first, accounting for the second greatest variance in the set of observations, etc. The first several principal components capture most of the variance in the observation set, and hence provide a great deal of information about the data. By projecting the observations onto the space spanned by the principal components, we can reduce the dimensionality of the data in a manner that preserves most of the variance.

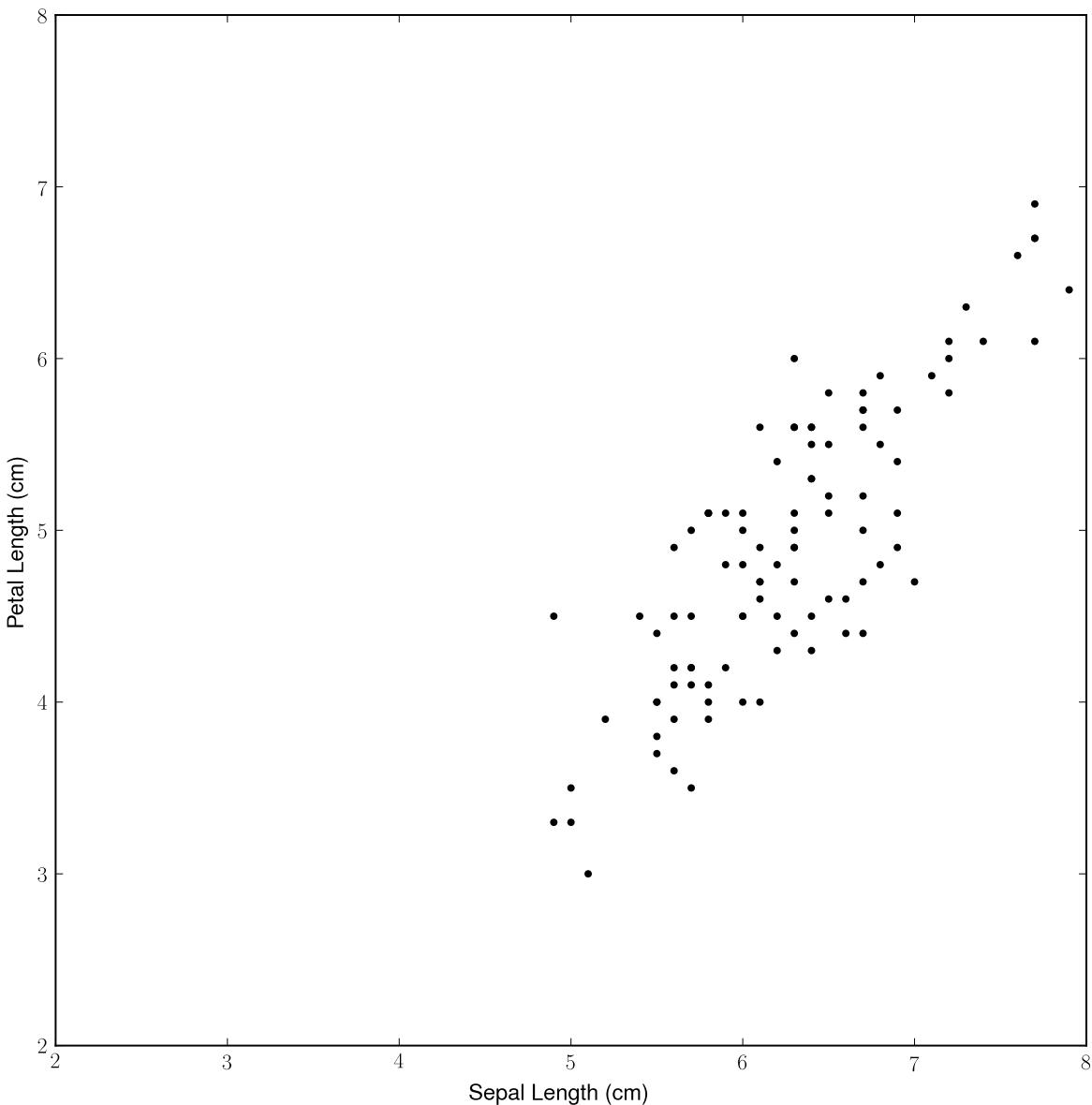


Figure 23.1: Sepal Length vs. Petal Length for 100 iris flowers. Note the strong correlation of these variables.

In our iris example, the two principal components are shown in Figure 23.2. The first principal component, corresponding intuitively to iris size, accounts for 96% of the variance in the data. The second, which accounts for only 4% of the variance, corresponds to the relative sepal and petal length of irises of the same size.

Computing the Principal Components

We now explore how to use the SVD to compute the principal components of a dataset. Throughout this lab we will use the `sklearn` iris data set, which can be obtained as follows:

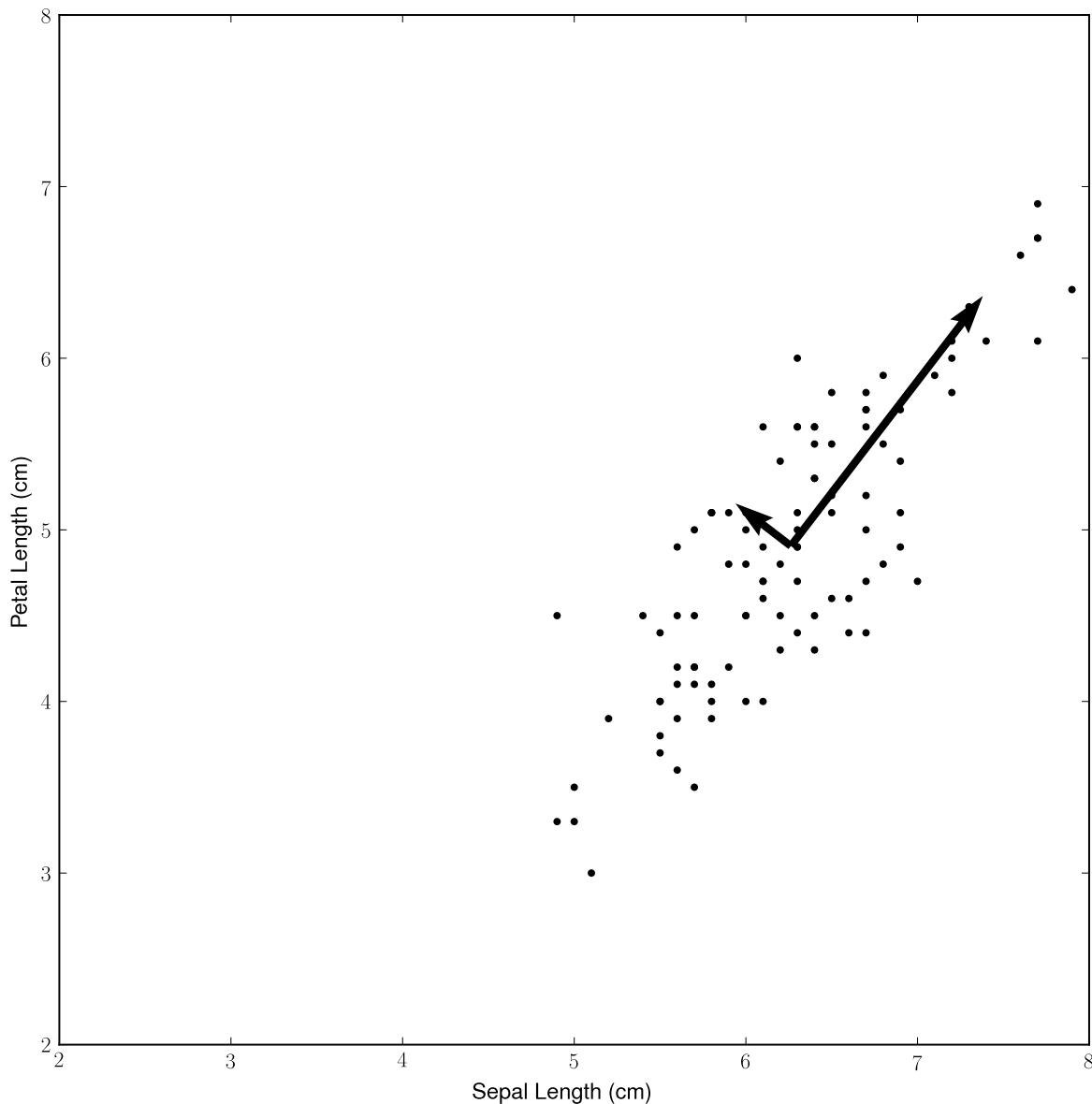


Figure 23.2: The vectors indicate the two principal components, which are weighted by their contribution to the variance.

```
>>> import numpy as np
>>> from scipy import linalg as la
>>> import sklearn.datasets as datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
```

We represent the collection of observations as an $n \times m$ matrix X , where each row of X is an observation, and each column is a specific feature. Let $k = \min(m, n)$. We will use this later. In the iris example, X contains 150 observations, each consisting of 4 features (so $k = 4$), as shown below:

```
>>> X.shape
(150L, 4L)
>>> iris.feature_names
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The first step in PCA is to pre-process the data. In particular, we first translate the columns of X to have mean 0. The data may then be optionally scaled to remove discrepancies arising from different units of measure (i.e. centimeters vs meters), and we call the new matrix containing the centered and scaled data Y . In this lab, we will not have any scaling issues, so we won't address this issue any further. Thus we can pre-process our iris data simply as follows:

```
>>> Y = X - X.mean(axis=0)
```

We next compute the truncated SVD of our centered and scaled data,

$$Y = U\Sigma V^T$$

where U is $n \times k$, Σ is a $k \times k$ diagonal matrix containing the singular values of Y in decreasing order along the diagonal, and V is $m \times k$. The columns of V are the principal components (which form an orthonormal basis for the space spanned by the observations), and the corresponding singular values provide us information about how much variance is captured in each principal component. More specifically, let σ_i be the i -th non-zero singular value. Then the value

$$\frac{\sigma_i^2}{\sum_{j=1}^k \sigma_j^2}$$

is the percentage of the variance captured by the i -th principal component. We compute the truncated SVD of the iris data and show the variance percentages for each component below:

```
>>> U,S,VT = la.svd(Y, full_matrices=False)
>>> S**2/(S**2).sum() # variance percentages
array([ 0.92461621,  0.05301557,  0.01718514,  0.00518309])
```

In general, we are only interested with the first several principal components. But just how many principal components should we keep? There are a number of ways to decide this. One is to only keep the first two principal components, as these enable us to project the data into 2-dimensional space, which is easy to visualize. Another way is to only keep the set of principal components accounting for a certain percentage (say 80%) of the variance. A third method is to examine the *scree plot* of the variance percentages for each principal component, as in Figure 23.3. Upon examination of the iris scree plot, we see that there is a distinct change after the first principal component. This method is referred to as finding the "elbow" of the scree plot, and we keep all the principal components on the left of the elbow. In the case of the iris data, that is simply the first principal component, which accounts for 92% of the variance.

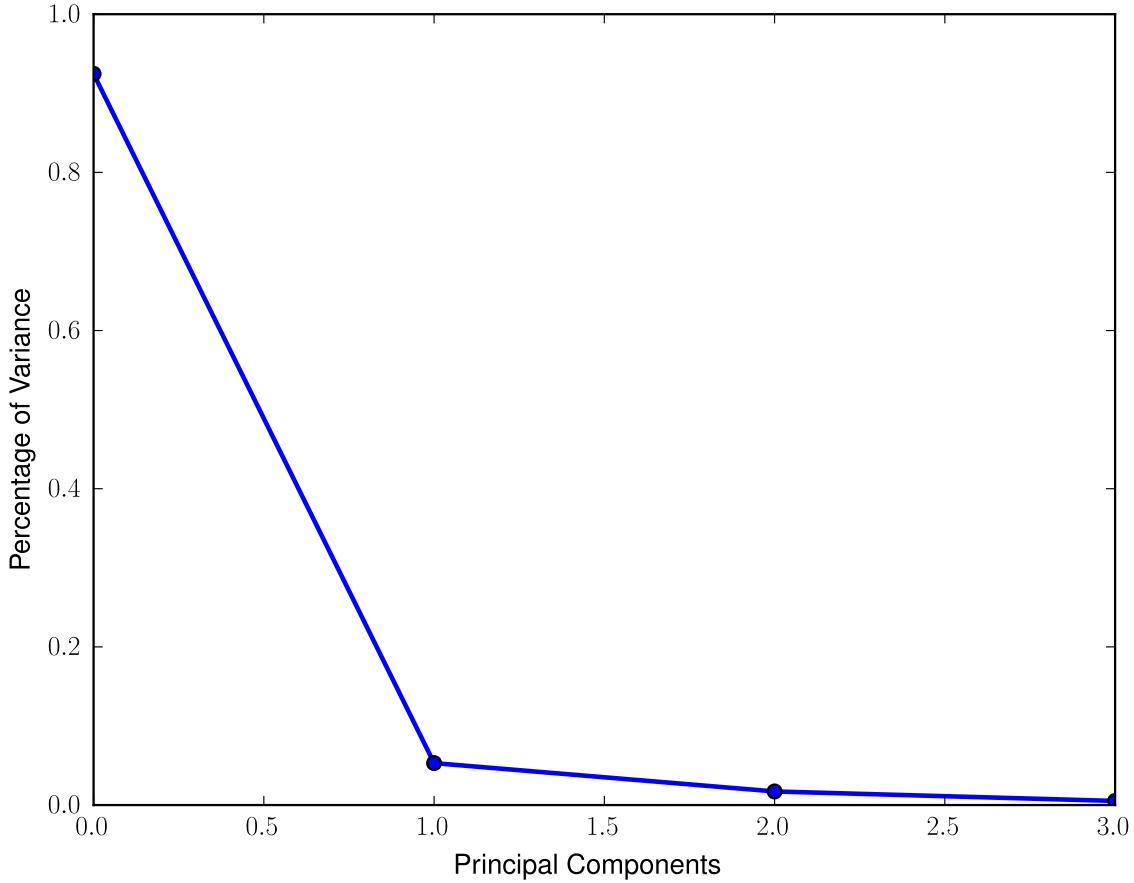


Figure 23.3: Scree plot of the percentage of variance for PCA on the iris dataset.

Once we have decided how many principal components to keep (say the first l), we can project the observations from the original feature space onto the principal component space by computing

$$\hat{Y} = U_{:,l} \Sigma_{:l,:}$$

where $\Sigma_{:l,:}$ is the first l rows and columns of Σ and $U_{:,l}$ is the first l columns of U . Using the SVD formula, note that

$$\hat{Y} = Y V_{:,l},$$

where $V_{:,l}$ is the first l columns of V . In this way, we see that the i -th row of \hat{Y} is simply the projection of the i -th observation onto the orthonormal set of the first l principal components. Under this projection, the data is represented in fewer dimensions, and in such a way that accentuates the variance (which can help with finding patterns within the data).

In Figure 23.4 we display the transformed iris data set, plotting the first principal component against the second. This reduction helps us to see the distinctions between the three different species, using only two dimensions instead of the full four dimensions of the feature space.

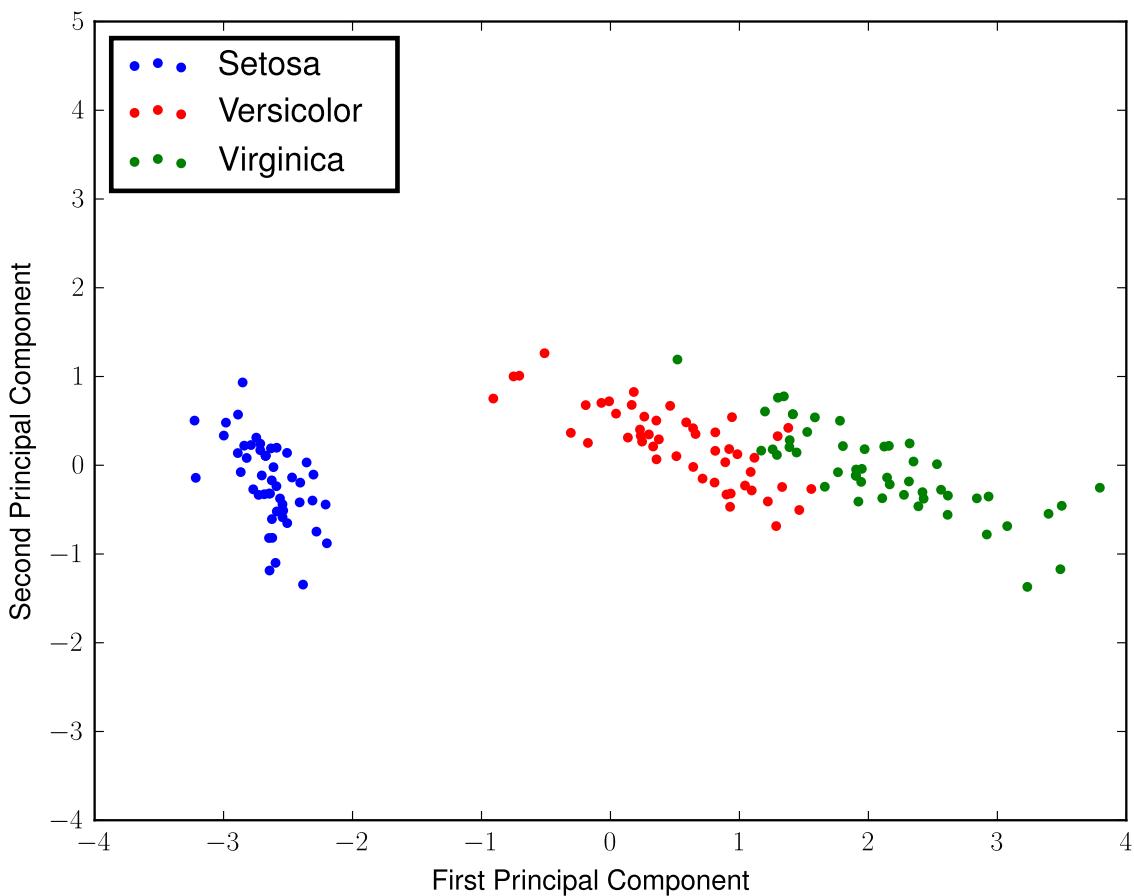


Figure 23.4: Plot of the transformed iris data, keeping only the first two principal components.

Problem 1. Recreate the plot shown in Figure 23.4 by performing PCA on the iris dataset, keeping the first two principal components.

Note: If `Yhat` is your 150×2 array of transformed observations, you can access the rows corresponding to the setosa flowers as follows:

```
>>> Yhat[iris.target==0]
```

To get the rows corresponding to versicolor and virginica specimens, simply replace the 0 with 1 and 2, respectively.

Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an application of PCA which applies the ideas we have discussed 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 dealing with various statistical and mathematical topics. How can we find an article about PCA? We might consider simply choosing the article which contains the acronym *PCA* 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, each row being 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 one document likely doesn't 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. We now have $X = U\Sigma V^T$. Once we have selected the number of principal components to keep, say l , we can represent the corpus of documents by the matrix

$$\widehat{X} = U_{:,l}\Sigma_{:,l}V_{:,l} = X V_{:,l}.$$

Note that \widehat{X} will no longer be a sparse matrix, but it has dimensions $n \times l$, which is much smaller than $n \times m$ when $l \ll m$.

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 just the cosine of the angle between the vectors; a small angle (and hence large cosine) indicates greater similarity, while a large angle (hence 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 \widehat{X} , notated \widehat{X}_i and \widehat{X}_j , respectively) is just

$$\frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

To find the document most similar to document i , we simply compute

$$\operatorname{argmax}_{j \neq i} \frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

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 will accomplish this task:

```
>>> # get list of filepaths to each text file in the folder
>>> import string
>>> from os import.listdir
>>> path_to_addresses = "./Addresses/"
>>> paths = [path_to_addresses + p for p in os.listdir(path_to_addresses) if p[-4:]==" .txt"]

>>> # helper function to get list of words in a string
>>> def extractWords(text):
>>>     trans = string.maketrans("", "")
>>>     return text.strip().translate(trans, string.punctuation+string.digits).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') as f:
>>>         for line in f:
>>>             vocab.update(extractWords(line))
```

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 whose key-value pairs are of the form (word, index):

```
>>> # load stopwords
>>> with open("stopwords.txt", 'r') as f:
>>>     stopwords = 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(stopwords))}
```

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 scipy import sparse
>>> 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') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             ctr.update(extractWords(line))
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_index.append(vocab[word])
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>             except KeyError: # if word isn't in vocab, skip it
>>>                 pass

>>> # create sparse matrix holding these word counts
>>> X = sparse.csr_matrix((counts, [doc_index,word_index]), shape=(len(paths),←
    len(vocab)), dtype=np.float)

```

Problem 2. Using the techniques of LSI discussed above—applied to the word count matrix X , and keeping the first 7 principal components—find the most similar and least similar speeches to both Bill Clinton’s 1993 speech and to Richard Nixon’s 1974 speech. Are the results plausible?

Hint: 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 make sure to read the documentation.

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 quite a few addresses, whereas *Afghanistan* will not. Thus two speeches sharing the word *war* ought to be considered more 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 choose to employ the following approach:

Let t_j be the total number of times term j appears in the entire collection of documents. Define

$$p_{i,j} = \frac{X_{i,j}}{t_j}.$$

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.

To calculate the matrix A in a streaming manner, we must alter our code above somewhat:

```
>>> from itertools import izip
>>> from math import log
>>> t = np.zeros(len(vocab))
>>> counts = []
>>> doc_index = []
>>> word_index = []

>>> # get doc-term counts and global term counts
>>> for doc, path in enumerate(paths):
>>>     with open(path, 'r') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             words = extractWords(line)
>>>             ctr.update(words)
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_ind = vocab[word]
>>>                 word_index.append(word_ind)
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>                 t[word_ind] += count
>>>             except KeyError:
>>>                 pass

>>> # get global weights
>>> g = np.ones(len(vocab))
>>> logM = log(len(paths))
>>> for count, word in izip(counts, word_index):
>>>     p = count/float(t[word])
>>>     g[word] += p*log(p+1)/logM

>>> # get globally weighted counts
>>> gwcounts = []
>>> for count, word in izip(counts, word_index):
>>>     gwcounts.append(g[word]*log(count+1))

>>> # create sparse matrix holding these globally weighted word counts
>>> A = sparse.csr_matrix((gwcounts, [doc_index, word_index]), shape=(len(paths) <-
, len(vocab)), dtype=np.float)
```

Problem 3. Repeat Problem 2 using the matrix A . Do your answers seem more reasonable than before?

24

Naive Bayes

Lab Objective: *Implement Naive Bayes Classification Models.*

Introduction

Naive Bayes classification methods are a good introduction to machine learning techniques. They are relatively straightforward to understand and to implement, while they are also very effective for certain applications. However, they are somewhat limited by their strong dependence on assumptions of independence.

Recall that “the classification problem” tries to assign the correct label to a given set of features (called a *feature vector*). For example, suppose we wish to give the correct labels (names) to two different pieces of fruit. The given features of the first fruit are that it is red and round, and the features of the second are that the fruit is long and yellow. If we assign to these fruits the names “apple” and “banana” respectively, then these are our labels.

It is common in classification problems to start with a set of correctly-labeled feature vectors called a *training set*. We use the training set to train our algorithm to make predictions. It is also common to have another smaller set of correctly-labeled feature vectors called a *test set*. To verify the effectiveness of our algorithm, we predict the labels of the test set, and then compare the predicted labels to the true labels.

Recall that Bayes rule for random variables gives that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{\int P(X|Y)P(Y)dy}$$

where $P(Y)$ is our prior distribution and $P(X|Y)$ is our likelihood function.

Suppose that we have a set of features that we wish to label. Let $x = (x_1, \dots, x_n)$ be this feature vector and let $C = \{c_1, \dots, c_k\}$ be our set of possible labels for x . We may apply Bayes rule to this problem as follows:

$$P(c_i|x) = P(c_i|x_1, \dots, x_n) = \frac{P(x_1, \dots, x_n|c_i)P(c_i)}{P(x_1, \dots, x_n)}$$

If we make no further assumptions, this problem is intractable. To effectively estimate even the simplest case where each feature is a boolean value would require us to estimate around $k2^n$ parameters. The problem gets exponentially worse if we were to consider non-boolean features.

However, if we can make the assumption that the features are conditionally-independent of one another, the problem can be simplified dramatically. If we make this assumption and apply Bayes rule, we have that

$$P(c_i|x_1, \dots, x_n) = \frac{P(x_1|c_i)P(x_2|c_i)\dots P(x_n|c_i)P(c_i)}{P(x_1, \dots, x_n)}.$$

In this case, we only need to estimate kn parameters. The Naive Bayes classification algorithm chooses the label with the highest probability. Since this is independent of the denominator in Bayes rule, we can simplify the problem further. Given an unlabeled feature vector $x = (x_1, \dots, x_n)$, we assign the label

$$c = \operatorname{argmax}_{i \in \{1, \dots, k\}} P(c_i) \prod_{j=1}^n P(x_j|c_i).$$

To assign a label to a set of features, we calculate each $P(x_j|c_i)$ and choose a prior $P(c_i)$. We then choose the argmax as above. The calculation of conditional probabilities and our choice of a prior will depend on the type of problem that we are solving.

Gaussian Classifiers

Gaussian classifiers are commonly used when dealing with continuous data. We assume that each feature is normally distributed and conditionally-independent of all other features. We may then calculate $\mu_{j,i}$ and $\sigma_{j,i}^2$ (the mean and variance) corresponding to each feature of each class. Then $P(x_j|c_i)$ can be calculated using the gaussian pdf

$$P(x_j|c_i) = \frac{1}{\sqrt{2\pi\sigma_{j,i}^2}} \exp -\frac{(x_j - \mu_{j,i})^2}{2\sigma_{j,i}^2}.$$

For example, suppose we have a training set with labels indicating the sex of a person (1 for female, 2 for male), and features consisting of hair length and height (features 1 and 2, respectively). Further suppose that the mean hair length of the women in the training set is 15 centimeters with standard deviation 1.5 cm, and the mean height is 1.25 meters with standard deviation 6 cm. Similarly, suppose that the mean hair length of the men in the training set is 5 centimeters with standard deviation 2.5 cm, and the mean height is 1.75 meters with standard deviation 7 cm. Under this setup, we have

$$\begin{array}{ll} \mu_{1,1} = 15, & \sigma_{1,1} = 1.5, \\ \mu_{2,1} = 1.25, & \sigma_{2,1} = .06, \\ \mu_{1,2} = 5, & \sigma_{1,2} = 2.5, \\ \mu_{2,2} = 1.75, & \sigma_{2,2} = .07. \end{array}$$

If we wish to classify a person with hair length 17 centimeters who is 1.4 meters tall, we calculate the probability of each label using the parameters given above and a uniform prior ($P(F) = P(M) = \frac{1}{2}$) as follows:

$$\begin{aligned} P(F | 17, 1.4) &= P(F) \left(\frac{1}{\sqrt{2\pi\sigma_{1,1}^2}} \exp -\frac{(17 - \mu_{1,1})^2}{2\sigma_{1,1}^2} \right) \left(\frac{1}{\sqrt{2\pi\sigma_{2,1}^2}} \exp -\frac{(1.4 - \mu_{2,1})^2}{2\sigma_{2,1}^2} \right) \\ &= .016 \\ P(M | 17, 1.4) &= P(M) \left(\frac{1}{\sqrt{2\pi\sigma_{1,2}^2}} \exp -\frac{(17 - \mu_{1,2})^2}{2\sigma_{1,2}^2} \right) \left(\frac{1}{\sqrt{2\pi\sigma_{2,2}^2}} \exp -\frac{(1.4 - \mu_{2,2})^2}{2\sigma_{2,2}^2} \right) \\ &= 1.7 \times 10^{-11} \end{aligned}$$

The Female label has a greater probability given the feature vector, and so we classify the person as Female.

A nice way to visualize how a classifier works is to plot the decision boundaries for two-dimensional subspaces of the feature vector space. For example, the decision boundaries for a Gaussian Naive Bayes classifier trained on a dataset consisting of the sepal widths and sepal lengths of three different types of flowers are shown in Figure 24.1.

Working in Log Space

In the example presented above, notice that the value of $P(M | 17, 1.4)$ is very small. This is often the case in classification problems; certain classes may be very unlikely, and so calculating these probabilities may lead to numerical underflow. This is especially pronounced in the Naive Bayes model, which involves taking the product of several numbers between 0 and 1. A useful technique to avoid underflow is to perform all of the computations in logarithmic space, where the products all become sums. When we do so, the Naive Bayes label assignment is

$$c = \underset{i \in \{1, \dots, k\}}{\operatorname{argmax}} \log P(c_i) + \sum_{j=1}^n \log P(x_j | c_i).$$

Since the logarithm is a monotone increasing function, the argmax is the same whether in log space or in the original formulation.

Problem 1. Download the `seeds_dataset.txt` file. This file contains 7 features describing 3 species of wheat.

1. Area
2. Perimeter
3. Compactness
4. Length
5. Width
6. Asymmetry Coefficient

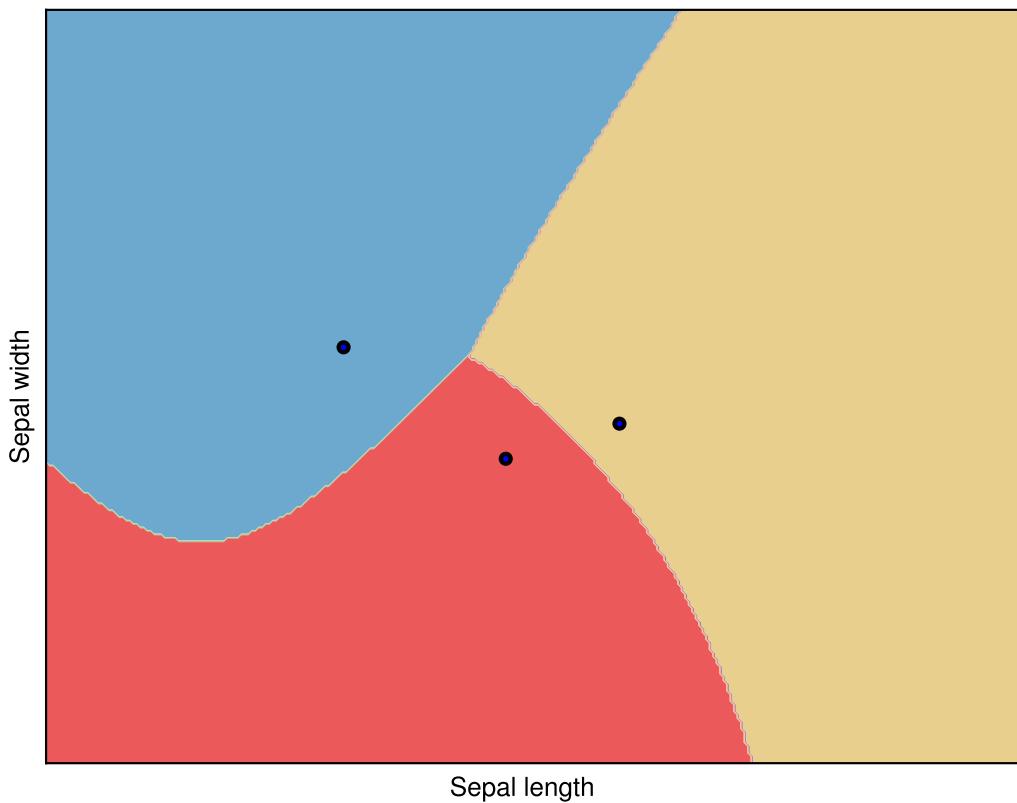


Figure 24.1: Decision boundaries for a Gaussian Naive Bayes classifier on the iris flower dataset, together with the means for each flower. Each point in the plane represents a 2-dimensional feature vector, and the color associated with each point indicates which class label was assigned to that feature vector.

7. Groove length

The species of wheat are

1. Kama
2. Rosa
3. Canadian

The measurements of the kernels are real valued, making this a good example on which to try our Gaussian classifier. Make a function that classifies a subset of the data in this file and returns the accuracy of your calculations by proceeding in the following manner:

1. Randomly select a test set consisting of 40 vectors. Make the remaining vectors into your training set.
2. Calculate the mean and variance for each feature of each label using the training set.

3. Using a uniform prior, predict the labels of your test set.
4. Compare your predictions to the correct labels of the test set. In particular, report the accuracy of the prediction, which is the number of correctly predicted test samples divided by the total number of test samples.

We may also use SciPy's `sklearn` library to implement a Gaussian classifier. After importing the library, we can create a new classifier and train it with just a few lines of code. To create a Gaussian classifier, use the following code.

```
from sklearn.naive_bayes import GaussianNB
nb_classifier = GaussianNB()
```

Given a training set, we can also quickly train the classifier to a certain problem. This requires the training set and labels as two arguments.

```
nb_classifier.fit(training_set, labels)
```

Once the classifier has been trained, we can predict the labels for a test set.

```
pred_labels = nb_classifier.predict(test_set)
```

The `predict` method returns an array of labels for the test set.

Problem 2. Repeat the previous problem using `sklearn`'s Naive Bayes classifier. Check that your implementation from the previous problem predicts the same labels as the `sklearn` implementation does.

Document Classification and Spam Filters

Naive Bayes classifiers are often used in document classification, a major example being spam detection. When it comes to document classification, a common choice for the feature vector is simply a count for the number of times each word in the specified vocabulary occurs in the document. For example, suppose we are trying to classify a document, and the vocabulary of relevant words is the ordered set

(bank, tree, wealth, money, river, water).

Suppose that the document is the sentence

"The woman deposited her money in the bank, and then made her way down to the bank of the river, contemplating her wealth."

Then the feature vector for this document is

(2, 0, 1, 1, 1, 0).

Notice in particular that the i -th entry of the feature vector indicates the number of occurrences of the i -th vocabulary word in the document. Such a feature vector is often called a *word-count vector*. Notice that the count vector ignores words in the document that are not part of the vocabulary, and it also disregards the order of the words. This simple representation of text documents is known as the *bag-of-words model* or the *vector space model*. The hypothesis that drives spam filters is that spam messages will use words with different frequencies. For example, a spam message will often have a sales pitch, so the word “buy” and “cheap” will appear often. On the other hand, legitimate messages will probably use different language. Thus, it is reasonable to use word-count vectors as our feature vectors when attempting to distinguish between spam and legitimate email.

We now introduce formalisms to derive the Naive Bayes model for document classification. Let $V = (v_1, v_2, \dots, v_n)$ be an ordered list of words, called the vocabulary, and let $x = (x_1, x_2, \dots, x_n)$ be a word-count vector. Let $\{c_1, c_2, \dots, c_k\}$ be the set of classification labels. In the case of continuous data and Gaussian Classifiers, each class label was associated with corresponding mean and variance parameters, and these determined the likelihood of the feature vector given the class label. In the case of document classification, each class label c_i has a corresponding probability vector $(p_{i,1}, p_{i,2}, \dots, p_{i,n})$ whose entries are nonnegative and sum to 1. This probability vector defines a categorical probability distribution over the vocabulary V , where $p_{i,j}$ represents the probability of seeing word v_j given class label c_i . With this notation in place, our Naive Bayes model takes the form

$$c = \operatorname{argmax}_{i \in \{1, \dots, k\}} P(c_i) \prod_{j=1}^n p_{i,j}^{x_j}.$$

Given a training set of labeled documents, we can calculate the prior probabilities $P(c_i)$ and word probabilities $p_{i,j}$ as follows. Each prior probability $P(c_i)$ is simply the proportion of training documents that have the label c_i . Next, let $\text{count}(c_i, v_j)$ denote the number of occurrences of word v_j among all training documents that have label c_i . Then we have

$$p_{i,j} = \frac{\text{count}(c_i, v_j) + 1}{\sum_{j=1}^n (\text{count}(c_i, v_j) + 1)}.$$

(Note that adding 1 to the number of occurrences of each word is known as *add-one smoothing*, and is a common technique to prevent over-fitting.)

Once we have calculated these parameters (the “fitting” stage), we are ready to classify new documents (the “prediction” stage) using the argmax equation given above. Remember to perform calculations in log space to prevent numerical underflow.

Problem 3. Implement a Naive Bayes model for document classification. We provide an interface below.

```
class naiveBayes(object):
    """
    This class performs Naive Bayes classification for word-count document←
    features.
    """
    def __init__(self):
        """
        Initialize a Naive Bayes classifier.
        """
        pass
```

```

def fit(self,X,Y):
    """
    Fit the parameters according to the labeled training data (X,Y).

    Parameters
    -----
    X : ndarray of shape (n_samples, n_features)
        Each row is the word-count vector for one of the documents
    Y : ndarray of shape (n_samples,)
        Gives the class label for each instance of training data. ←
        Assume class labels are in {0,1,...,k-1} where k is the ←
        number of classes.
    """
    # get prior class probabilities P(c_i)
    # (you may wish to store these as a length k vector as a class ←
    # attribute)

    # get (smoothed) word-class probabilities
    # (you may wish to store these in a (k, n_features) matrix as a ←
    # class attribute)

    pass

def predict(self, X):
    """
    Predict the class labels of a set of test data.

    Parameters
    -----
    X : ndarray of shape (n_samples, n_features)
        The test data

    Returns
    -----
    Y : ndarray of shape (n_samples,)
        Gives the classification of each row in X
    """
    pass

```

Problem 4. In this problem, you will train a Naive Bayes classifier using a corpus of emails extracted from the Enron dataset.

Load in the data from `SpamFeatures.txt`. This is a text file containing a whitespace-delimited numerical array with several thousand columns and several thousand rows, each row representing an email as a count vector. Also load in the data from `SpamLabels.txt`, which is a text file containing a 1 (for legitimate email) or 0 (for spam email) on each line, in correspondence with the rows of the count vector array. Using your document classification implementation, do the following:

1. Randomly create a test set from the data (500 documents), leaving the remaining documents as the training set.
2. Create a Naive Bayes classifier and fit it using the training set.
3. Predict the labels of the test set and compare them to the true labels (by reporting the classification accuracy).

Next, perform the same task using `sklearn`'s implementation and the same training and testing sets:

```
>>> # assume train_vectors, train_labels, and test_vectors are defined
>>> from sklearn.naive_bayes import MultinomialNB
>>> mnb = MultinomialNB()
>>> mnb.fit(train_vectors, train_labels)
>>> predicted = mnb.predict(test_vectors)
```

Again report the accuracy of the predicted labels. The result should be on par with those produced by your own implementation.

25 Logistic Regression

Lab Objective: *Understand the basics of Logistic Regression, and apply to the Titanic problem.*

Binary Logistic Regression

A *Logistic Regression Model* is a probability model that can be used to predict outcomes for a set of data. Usually "Logistic Regression" refers to what is more appropriately called *binary logistic regression*. This is a model which can assign data points to one of two sets, and is used in many different fields. One common medical example is predicting whether or not a patient has a particular disease. Based upon several factors, which may be both continuous (age, height, weight) and categorical (gender, race), we can quantify the probability of infection. This probability is computed by way of the *logistic function*, which, given the contributing factors, will return a probability value between 0 and 1. This probability will then be used to assign a label to our input data ('infected' or 'not infected', for example) by using some cut-off value, and will depend on the need for accuracy in the specific application. Success corresponds to a label of 1, and failure to a label of 0.

The logistic function takes in as input any real number and returns a value between 0 and 1, and is defined explicitly as

$$\phi(t) = \frac{1}{1 + e^{-s}}, \quad (25.1)$$

where s is some combination of the input variables x_1, \dots, x_n . The graph of this function can be seen in Figure ???. $\phi(x)$ can then be interpreted as the probability of success. In some cases it is not possible to find a closed-form expression for the correct combination of the input variables. However, in many cases we can achieve reasonable accuracy by using a linear combination of x_i, \dots, x_n , i.e.

$$s = c_0 + c_1 x_1 + \dots + c_n x_n.$$

This can be written more compactly as

$$s = \mathbf{c}^T \mathbf{x},$$

where $\mathbf{x} = (1, x_1, \dots, x_n)^T$ and $\mathbf{c} \in \mathbb{R}^{n+1}$.

Given a training set of labeled data points, a Logistic Regression Model will find the optimal \mathbf{c} for the data, and can then be used to predict labels for further data points.

The Titanic Problem

The Titanic dataset is especially useful for Logistic Regression. This dataset is composed of actual data obtained concerning the passengers on the ill-fated Titanic voyage, given in the bulleted list below. Using logistic regression, we can predict whether or not a passenger survived based on this data. We do so by training a model on a portion of the dataset, the training set, and predicting labels for the remaining data, the test set.

Before beginning our classification, however, we will first need to process our data. The Titanic dataset contains much more information than is currently relevant for our purposes. You can obtain this data in Excel Spreadsheet form at [Insert link here]. We recommend using pandas to read in and process the data. The columns are as follows:

- **pclass**: An integer in {1, 2, 3} which describes the class the passenger was in.
- **survived**: The dependent variable. 1 indicates survival, and 0 death.
- **name**: A string containing the passenger's name.
- **sex**: A string, either 'male' or 'female'.
- **age**: Either an integer or a float.
- **sibsp**: An integer giving the number of siblings and/or spouse who embarked with the passenger.
- **parch**: An integer giving the number of parents and/or children who embarked with the passenger.
- **ticket**: A string containing the transaction code for the ticket(s) purchased.
- **fare**: A float giving the cost of the ticket purchased.
- **cabin**: A string giving the assigned sleeping cabin for the passenger (note that the majority of this column is blank).
- **embarked**: A string in {S, C, Q} corresponding to the location of the passenger's embarkment, Southampton (UK), Cherbourg (France), or Queenstown (Ireland), respectively.
- **boat**: An int or string for those who survived giving which life boat they rode in.
- **body**: An int giving the number of body for those who died who were found and identified.
- **home.dest**: A string giving the home of or location to which the passenger was headed.

Problem 1. Create a function called `initialize` which will process the Titanic data set into useable format by doing the following:

1. Choose the coulmns that you believe will be relevant in predicting the survival of the passengers, and drop the other columns. You may not use `boat` or `body`, as these are dependent on whether or not the passenger survived. Be sure to include `survived`, which will be separated later as the independent variable, as well as `sex` and `pclass`.
2. Since `sex` is really a binary variable, make it one explicitly by changing "female" and "male" to be binary values.

3. Drop the rows that contain missing values. Make sure you have a significant number of rows left. If you have too few, you may need to choose fewer columns to keep before deleting the incomplete rows.
4. Because the `pclass` column is an integer in $\{1, 2, 3\}$, it will be treated as a ranked variable instead of simply a categorical variable. It may be useful to rank this variable, or it may mess up our classification. Include a keyword argument `pclass_change` with default `True`. If it is set to `True`, eliminate this ranking by dividing `pclass` into two binary columns. Make one column a boolean for being 1st class and the other a boolean for being 2nd class. (This means that a value of 1 would correspond to [1, 0], 2 to [0, 1], and 3 to [0, 0].)
5. Split the remaining rows into a training and a test set using a 60/40 split. Be sure and pick random rows for each group and not rows in any particular order.

Have your function return the training set and the test set, in that order.

Model Evaluation

Now that we have our training set, we can train a model, which can be used to obtain the probability of success for each data point. The label chosen depends heavily on the probability cut-off value mentioned previously, which we represent as τ . In the simplest manner, we can simply pick a value of $\tau = 0.5$, which will then assign the label with the highest likelihood. However, it is often beneficial to choose a different value of τ . In regards to the probability of infection for serious diseases, it might be best to give a patient medicine if they have even a 10 percent chance of infection. So how can we find the “best” cut-off value?

In order to determine this, we need to discuss how to measure the accuracy of the labels predicted. Say that we have picked a cut-off value τ , and have assigned labels to the test set. Using the predicted labels and the actual labels, we can obtain four important values: the number of *true positives*, *false positives*, *true negatives*, and *false negatives*, which are abbreviated TP, FP, TN, and FN, respectively. These values are integers which together sum to the number of labels predicted. You can see the definition of these in Table ??? (until the figure is up, true positives are the points with predicted label 1 and true label 1, false positives have predicted label 1 true label 0, true negatives predicted label 0 true label 0, and false negatives true label 1 predicted label 0). We can now use these to report our accuracy in various metrics:

- *Prediction accuracy* is defined as $\frac{TP+TN}{TP+FN+FP+TN}$, and is the percentage of correctly predicted cases.
- *Sensitivity*, also known as the *true positive rate* or *TPR*, is given by the fraction of correctly predicted cases where the actual outcome is 1, $\frac{TP}{TP+FN}$.
- *Specificity*, the *true negative rate*, or *TNR* is the proportion of correctly predicted cases where the true outcome is 0, $\frac{TN}{FP+TN}$.
- *False Positive Rate*, or *FPR*, is the proportion of incorrectly predicted cases where the true outcome is 0, and is given by $\frac{FP}{TP+TN}$.

All of these depend strongly on the value chosen for τ .

A *roc curve* is useful in measuring the accuracy of a model. To make a roc curve, we pick many values for τ , and obtain the False Positive Rate and the True Positive Rate for each. Then we plot the data points (FPR_τ, TPR_τ) for each value of τ chosen and connect them into a curve. A completely random label assignment would result in a nearly-linear roc curve, while a more accurate assignment would result in a more steeply-rising curve (see Figure ???). A good choice for τ is the one that intersects the family of lines $y = x + b$ at only one point, which intuitively is the point closest to the vertex $(0, 1)$. Mathematically, this is given by

$$\arg \max_{\tau} (TPR_\tau - FPR_\tau). \quad (25.2)$$

Problem 2. Use the function declaration below to find the best value for τ . You should use evenly spaced values from 0 to 1, exclusive.

```
def best_tau(predicted_labels, true_labels, n_tau=100, plot=True):
    """
    Parameters
    -----
    predicted_labels : ndarray of shape (n,)
        The predicted labels for the data
    true_labels : ndarray of shape (n,)
        The actual labels for the data
    n_tau : int
        The number of values to try for tau
    plot : boolean
        Whether or not to plot the roc curve

    Returns
    -----
    best_tau : float
        The optimal value for tau for the data.
    """
    pass
```

Now that we have a good value for τ , we can quantify the accuracy of a model. We will do so for a few different types of models for the Titanic data you initialized previously. For the first two, we will use the logistic classifier found in `sklearn.linear_model.LogisticRegression`. The first model we use will be our “Unchanged Logistic Classifier”, which will use our Titanic data with `pclass` unchanged. The second model is the “Changed Logistic Classifier”, which use the data with `pclass` changed.

When using this package to create a classifier, we need to input a keyword argument `C`. This value represents the inverse of the regularization strength. Different values will yield different results. A better value for `C` will yield a more steeply-rising roc curve. The model can find the coefficients (the vector `c`), along with the probabilities of failure and success for each label. With these in hand, we can use the function `sklearn.metrics.roc_curve` to obtain the False Positive Rates, the True Positive Rates, and the optimal value for τ . You will need to pass in the test data, the probability of success, and the keyword argument `pos_label = 1`. The accuracy of the model with input value `C` can then be obtained using `sklearn.metrics.auc`, which will give the area under the curve. The larger the area, the more steeply-rising curve we have, and the better the model. Note that we create a single roc curve using one value for `C` and multiple values for τ .

Problem 3. Use the following function declaration to return the auc score for the two Logistic Regression models described.

```
def auc_scores(unchanged_logreg, changed_logreg):
    """
    Parameters
    -----
    unchanged_logreg : float in (0,1)
        The value to use for C in the unchanged model
    changed_logreg : float in (0,1)
        The value to use for C in the changed model

    Returns
    -----
    unchanged_auc : float
        The auc for the unchanged model
    changed_auc : float
        The auc for the changed model
    """
    pass
```

We can test a Naive Bayes model against our Logistic Regression model for both the unchanged and changed models to see the comparative accuracy. Use the model `MultinomialNB`, found in `sklearn.naive_bayes`. You can use it in the same manner as `LogisticRegression`, except instead of passing in the keyword argument `C`, you will pass in a keyword argument `alpha` corresponding to a smoothing parameter.

Problem 4. Add input variables `unchanged_bayes` and `changed_bayes` to your function from the previous problem to obtain the auc for each of these models. Your function should return all four areas, unchanged logistic regression, changed logistic regression, unchanged Bayes, and changed Bayes, in that order.

Different values for `C` and `alpha` will yield different results. We seek to find those that will maximize the area under the curve. One way to do so is to pick a number of evenly-spaced points between 0 and 1, exclusive, and try each one in turn, keeping the value that yields the greatest accuracy.

Problem 5. Use the function declaration below to find the optimal values for `C` and `alpha` as described.

```
def find_best_parameters(choices):
    """
    Parameters
    -----
    choices : int
        The number of values to try for C and alpha

    Returns
    -----
    best : list of length 4
        The best values for C for the unchanged and changed logistic
        regression models, and the best values for alpha for the
        unchanged and changed Naive Bayes models, respectively.
    """
    pass
```

Now that we have found the optimal inputs for these functions, we can test them against one another.

Problem 6. Create a function called `results` which will graph of the roc curves for each of the methods, and will print out the names of the models with their corresponding areas, in numerically descending order.

26

Classification Trees

Lab Objective: *Understand how to build a classification tree and use it to predict survival of Titanic passengers.*

Classification trees are a class of decision trees, and are used in a wide variety of settings where labeled training data is available, and where the desired outcome is a model which is able to accurately assign labels to unlabeled data. We assume that each sample d has P attributes, which can be real-valued or categorical, and that each sample belongs to some class k , where there are K classes. The tree is composed of many *nodes*, which represent a decision point (i.e. a question is asked about the sample which has a boolean response). 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, i.e. it is the end of the line and there is not a question asked. Each leaf has a classification assigned to it, and an unlabeled sample is labeled with that classification upon arrival at the leaf node.

How do we train a classification tree? We start with a labeled data set D and choose the best attribute p and value x by which to *split* the data. We have now partitioned D into two sets, which we may then split as well. We continue in this manner until some stopping criterion is met (often a maximum depth of the tree). To formalize this, we need several definitions.

Definition 26.1. Let D be a data set with K different classes. Let N_k be the number of samples labeled class k for each $1 \leq k \leq K$, and let $f_k = \frac{N_k}{N}$ where N is the total number of samples in D . We define the Gini impurity to be

$$G(D) = 1 - \sum_{k=1}^K f_k^2.$$

Problem 1. Write a function that accepts a list of class assignments and a list of all the K possible classes, and computes the Gini impurity.

Definition 26.2. We define the split $s_D(p, x)$ of the data set D on attribute p using value x , to be a partition D_1, D_2 such that

1. $d_p \leq x$ for all $d \in D_1$ and $d_p > x$ for all $d \in D_2$, where d_p is the value of attribute p in d , assuming real values; or
2. $d_p = x$ for all $d \in D_1$ and $d_p \neq x$ for all $d \in D_2$, where d_p is the value of attribute p in d , assuming categorical values.

Problem 2. Write a function that computes the split of a data set for a given variable p and given value x . It should return the partitioned data set, as well as the partitioned class labels.

Definition 26.3. Let $s_D(p, x) = D_1, D_2$ be a split. We define the information gain of this split to be

$$I(s_D(p, x)) = G(D) - \sum_{i=1}^2 \frac{|D_i|}{|D|} \cdot G(D_i)$$

Problem 3. Write a function that computes the information gain for the split of a data set for a given variable p , value x .

We define the optimal split of a data set to be

$$s_D^* = s_D(p^*, x^*),$$

where

$$p^*, x^* = \operatorname{argmax}_{p,x} I(s_D(p, x)).$$

From this partition, we create two child nodes, assigning the left child node data set D_1 , and right child node data set D_2 .

Problem 4. Write a function that computes the optimal split of a data set. You may need to separate this into two tasks: finding the optimal split for each attribute p , and then choosing the optimal split over all the attributes.

Let's put all of this together to create the full classification tree.

Problem 5. Write a class called `Node` that creates and trains a classification tree. It should accept a training data set D , class labels y , current depth (which when initialized should be 1), some maximum depth which is greater than 1, and some tolerance for the Gini impurity (say 0.2). Use recursion to build the tree, i.e. after determining the optimal split, create two new nodes (`leftchild` and `rightchild`), with incremented depth. If the depth equals the maximum depth or the Gini impurity for a node is less than the tolerance, assign the majority label to the node and do not split further.

Problem 6. Write a method for the class `Node` that prints out the tree structure. For each node it should show which attribute p and value x provide the optimal split, and for the leaf nodes, it should show the assigned label. You may use your own creativity for how to display this.

Problem 7. Write a method for the class `Node` that assigns the class label for a new sample. You will probably have to make this method recursive also.

We would like to test our classifier on a real data set. Provided for you is a data set on about 1000 passengers aboard the Titanic. We would like to predict their survival or death depending on several attributes: class (1^{st} , 2^{nd} , or 3^{rd}), gender (male or female), and age.

Problem 8. Using the Titanic data set, train a classification tree with a maximum depth of 10 nodes and Gini impurity tolerance .1, and predict labels for the test set. What is your misclassification rate? Print out the tree structure. Is it what you expected? Was there any optimal split which surprised you?

The free parameters which we can vary are the maximum depth and the Gini impurity tolerance. Higher values for the maximum depth creates more refined, specific trees, as does a smaller Gini impurity tolerance. In this case, we are making the classifier more *complex*. As such, it will perform better on the data on which it is trained (it has learned the training data well), but perform worse on new, test data (it is not very generalizable). Keeping the Gini impurity tolerance at 0.1 and increasing the maximum depth yields the following interesting misclassification curves. What is the take away message?

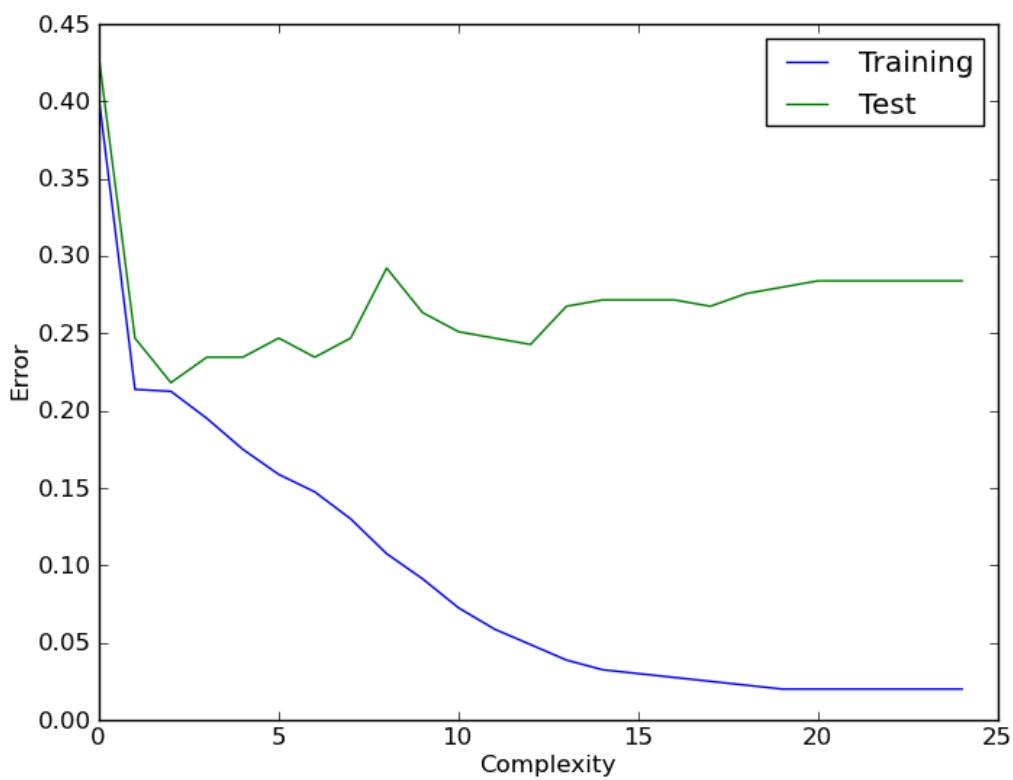


Figure 26.1: Misclassification rate on training data and test data with increasing complexity.

27

Random Forests

Lab Objective: *Understand how to build a random forest and use it to predict survival of Titanic passengers.*

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, random subset of the attributes is available by which to determine the next split. Each trained tree in the forest casts a vote for the labeling of a new sample, and the sample is labeled according to the majority vote of the trees.

Your approach to the classification tree may have been sloppy, depending on how careful you were about odd cases (say trying to split a data set on gender when each sample is male). It doesn't affect us much when all the attributes are available on which to split, unless we grow the tree too deeply. However, with the random forests these odd cases crop up more frequently, as we only have a small subset of attributes to choose from. We need to be more careful then, and keep track of which attributes are still available to split on and only consider these.

Problem 1. Modify your code for classification trees so that we keep track of which attributes are available to split on at each node. We can only split on an attribute if it assumes two or more distinct values present in the data set. For example, in the Titanic data set, once we have split on gender, we can never split on it again in any descendant node, since one child data set will only have males and the other will only have females.

We must next add the randomness to our trees.

Problem 2. Modify your code for classification trees so that each tree is trained *randomly*, i.e. when determining the optimal split, randomly select a small subset of the available variables, and use them to split on. You should be able to specify the size of the subset. If the number of available variables is smaller than the size of the random subset, then terminate the node (make it a leaf node).

We can now train the whole forest.

Problem 3. Make a class *Forest* which trains a collection of random trees. Use the following implementation:

```
class Forest(data, targets, Gini, max_depth, num_trees, num_vars):
    """
    Train a collection of random trees.

    Parameters
    -----
    data : ndarray of shape (n,k)
        Each row is an observation.
    targets : ndarray of shape (K,)
        The possible labels or classes.
    Gini : float
        The Gini impurity tolerance
    max_depth : int
        The maximum depth for the the trees
    num_trees : int
        The number of trees in the forest.
    num_vars : int
        The number of variables randomly selected at each node.
    """

    pass
```

Note that `num_vars` should be small, i.e. $\text{num_vars} \approx \sqrt{P}$ where P is the total number of attributes in the data set. The number of trees in the forest should be somewhat large, say greater than 100.

Problem 4. Write a method that assigns a label to a new sample, by considering the majority vote of the trees.

Let us reexamine the Titanic data set and see if we get any significant improvement.

Problem 5. Train a random forest on the Titanic data set, and for your inputs use `num_vars` = 2 and 100 trees. Let the Gini impurity tolerance be 0.1 and the maximum depth be 10. What is your misclassification rate? Was there any significant improvement?

28

K-Nearest Neighbors and Support Vector Machines

Lab Objective: *Implement the k-Nearest Neighbor (KNN) and binary Support Vector Machine (SVM) classifiers.*

For numerical data, one of the most simple classification methods is the k-nearest neighbor (KNN) classifier, which labels a new sample according to the majority vote of the nearest k training samples. As k is the only parameter for the model, this choice determines the effectiveness of the classifier. Throughout this lab we will explore how different values of k affect the accuracy of our classifier.

Suppose we have numerical data $\mathbf{x}_1, \dots, \mathbf{x}_N$ and associated labels y_1, \dots, y_N , along with a metric d on our feature space. We define the k -neighborhood of a new sample \mathbf{x} to be

$$n(\mathbf{x}, k) = \{\mathbf{x}_i : d(\mathbf{x}_i, \mathbf{x}) < d(\mathbf{x}_j, \mathbf{x}) \text{ for all but fewer than } k \text{ samples } \mathbf{x}_j\}$$

Thus the k -neighborhood of a new sample is the set of samples from our training set which are the k closest samples according to our metric.

Problem 1. Write a function that computes the k -neighborhood of a sample \mathbf{x} given k and a training set. Assume the use of the Euclidean metric.

We define the k -neighborhood votes of a new sample \mathbf{x} to be

$$v(\mathbf{x}, k) = \{y_i : \mathbf{x}_i \in n(\mathbf{x}, k)\}$$

The label assigned to \mathbf{x} according to the standard KNN classifier is the mode of the k -neighborhood votes.

Problem 2. Write a function that labels a new sample \mathbf{x} given k and a training set. Assume the use of the Euclidean metric.

Problem 3. Write a KNN class which accepts initial training data and training labels. It should have a method to classify new samples, given a value of k . Load the iris dataset from `sklearn.datasets`, and by separating the data into training and testing sets, implement your class. Test your classifier on the test data given different values of k . What are the misclassification rates?

Different values of k lead to different results. Essentially, our choice of k determines how far-reaching we would like the influence of a sample to be. Larger k means that a sample is influenced by points farther away from it. Smaller k means that a sample is influenced only by the few points nearest it. In either case, extreme choices of k (too small or too big) often yield poor results.

Another powerful classifier is the support vector machine (SVM). There are two main ideas in this classifier: maximum-margin hyperplanes and kernel functions. The first is simply the thought that the simplest binary classifier is a separating hyperplane, the best being the hyperplane that is “farthest” from the nearest two points of opposing classes, while perfectly partitioning the training data. There are very few interesting classification problems where this is possible in the standard feature space. However, if we can transform the feature space into a higher-dimensional space, then we might be able to find such a hyperplane. Unfortunately, working in this higher-dimensional space can be quite costly, which is where the kernel functions come into play.

The second big idea is that instead of working directly in the higher-dimensional space, we can choose our transformation in such a way that we can use *kernel* functions for any necessary computations whose domain is the product space of the original feature space with itself. There are many, sometimes exotic, kernel functions to choose from, though in practice only a few forms are used. We let $\phi(\mathbf{x})$ be the transformation of \mathbf{x} into the higher-dimensional space, and we let this transformation be determined by some kernel function

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j).$$

We assume now that our labels are simply ± 1 , and we assume $\phi(\mathbf{x}) \in \mathbb{R}^K$. We consider the hyperplane defined by $f(\phi(\mathbf{x})) = \mathbf{w}^T \phi(\mathbf{x}) + b$, where $\mathbf{w} \in \mathbb{R}^K$ and $b \in \mathbb{R}$. We wish to find \mathbf{w} and b such that $f(\phi(\mathbf{x}_i)) > 0$ if $y_i = 1$ and $f(\phi(\mathbf{x}_i)) < 0$ if $y_i = -1$. Thus we need \mathbf{w} and b to satisfy $y_i (\mathbf{w}^T \phi(\mathbf{x}) + b) > 0$ for all $i = 1, \dots, N$. Additionally, we would like the distance between the boundary $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$ and the nearest points to be maximized. We can determine this distance geometrically to be

$$\frac{2}{\|\mathbf{w}\|},$$

and is called the margin. Thus we would like to solve the following optimization problem:

$$\begin{aligned} &\text{minimize } \|\mathbf{w}\| \\ &\text{subject to } y_i (\mathbf{w}^T \phi(\mathbf{x}) + b) > 0 \quad \text{for } i = 1, \dots, N. \end{aligned}$$

Considering the Lagrangian of this optimization problem, we have the dual formulation of this optimization problem as

$$\begin{aligned} &\text{maximize } \sum_{n=1}^N a_n - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N a_i a_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \\ &\text{subject to } a_i \geq 0 \quad i = 1, \dots, N \\ &\qquad \sum_{i=1}^N a_i y_i = 0. \end{aligned}$$

This is simply a quadratic programming problem, where the objective function is $\mathbf{a}^T \mathbf{1} - \frac{1}{2} \mathbf{a}^T Q \mathbf{a}$, where

$$Q_{ij} = y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)^T = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j).$$

Quadratic programming problems have nice solutions, so this will be easy to solve. The classifier function $f(\mathbf{x}) = \sum_{i=1}^N a_i y_i k(\mathbf{x}, \mathbf{x}_i)$ also has a nice closed-form solution.

Given a training set of size `n_samples` and a kernel k , with data X and target Y , we can use `cvxopt` to solve this quadratic programming problem:

```
>>> import cvxopt
>>> import numpy as np
>>> K = np.zeros((n_samples,n_samples))
>>> for i in xrange(n_samples):
>>>     for j in xrange(n_samples):
>>>         K[i,j] = k(X[i,:], X[j,:])
>>> Q = cvxopt.matrix(np.outer(Y, Y) * K)
>>> q = cvxopt.matrix(np.ones(n_samples) * -1)
>>> A = cvxopt.matrix(Y, (1, n_samples))
>>> b = cvxopt.matrix(0.0)
>>> G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
>>> h = cvxopt.matrix(np.zeros(n_samples))
>>> solution = cvxopt.solvers.qp(Q, q, G, h, A, b)
>>> a = np.ravel(solution['x'])
```

From this value a , our kernel k , a training set X , and target Y , we have everything we need to build an SVM classifier. But what should our kernel k be? There are three common kernels used:

$$\text{Polynomial: } k(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + a)^d$$

$$\text{Radial Basis Function: } k(\mathbf{x}, \mathbf{y}) = e^{-\gamma \|\mathbf{x} - \mathbf{y}\|^2}$$

$$\text{Sigmoid: } k(\mathbf{x}, \mathbf{y}) = \tanh(\mathbf{x}^T \mathbf{y} + r)$$

Problem 4. Write an SVM class. Upon initialization, it should accept a training data set and training target set. It should have a method called `setKernel` which accepts one of the three kernel types, and defines a kernel function for the object. It should also have a method to train the classifier, and another method to predict the class of a new sample. It should predict 1 if $f(\mathbf{x}) > 0$ and -1 if $f(\mathbf{x}) < 0$.

A data set on breast cancer has been provided to you. This is from a breast cancer database from the University of Wisconsin Hospital, Madison, from Dr. W. H. Wolberg. The attributes are (in order): clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitoses, all on a scale from 1 to 10. The targets are either 1 or -1, signifying malignant or benign, respectively.

Problem 5. Load the data set. Separate it into a training set and a test set. Train SVMs on the data, trying each kernel with various parameter values. What are your misclassification rates?

29

Crime Mapping

Lab Objective: *Use Gaussian Mixture Models and Kernel Density Estimates to map homicides in Baltimore.*

Suppose we are moving to Baltimore, and we need to find a place to live. We would like to be aware of the crime (homicides in particular) in each neighborhood, as this will certainly influence our decision. We could simply look at a map of all homicides in the city over several years, as shown in Figure 29.1.

This is helpful, but we would like to find the largest geographical area of Baltimore from which to choose a home, with only a small fraction (say 5%) of the homicides. This is equivalent to finding the smallest geographical area of Baltimore containing 95% of all homicides. To do this, we'll need to estimate the probability density of homicides in Baltimore.

We will use two approaches to estimating this pdf. The first is via GMMs.

Problem 1. Unpickle the file `homicides`, which contains the approximate longitude (first column) and latitude (second column) of a homicide in Baltimore. Train a 3-component GMM on this data set.

We can plot this density over the Baltimore region as in the previous lab, using `matplotlib.pyplot.imshow`.

Problem 2. Compute the density at each point on a sufficiently fine grid of the Baltimore region. Save a plot of this pdf.

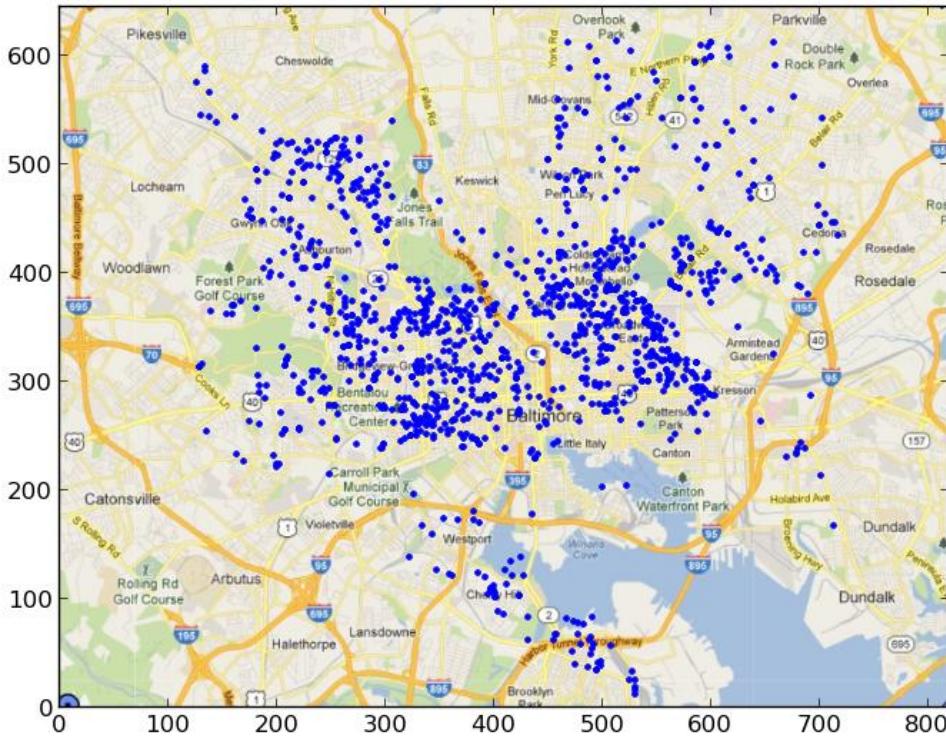


Figure 29.1: Map of homicides in Baltimore over several years.

From this model, we can get a more probabilistic viewpoint of crime in the Baltimore area, as in Figure 29.2.

Alternatively, we can use a Kernel Density Estimate (KDE) of our data. A KDE is a method of smoothing data. We initially examined a scatterplot of the homicide data. Suppose that given N data points, we replace each data point x_i with some symmetric probability distribution K_h . We call this distribution K_h a *kernel*, and it is centered at the origin and has some smoothing parameter h . If we scale each kernel by $\frac{1}{N}$, then their sum will be a probability distribution. Thus the KDE for the data x_1, \dots, x_n is

$$\hat{f}_h(x) = \frac{1}{n} \sum_{i=1}^n K_h(x - x_i) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

This formulation allows us to choose a kernel family K , and then alter it with a smoothing parameter h to get various kernels. Of course, this formulation only makes sense for univariate data (otherwise h couldn't simply be a scalar).

Choosing h is often the most challenging aspect of dealing with KDEs. If h is chosen too small, then the pdf becomes too multimodal (a large mode at each data point). If it is too large, it is too smooth, approaching a uniform distribution. For univariate data with a Gaussian kernel, an optimal choice for the bandwidth is $h \approx 1.06\hat{\sigma}n^{-1/5}$ where here, $\hat{\sigma}$ is the standard deviation of our data.

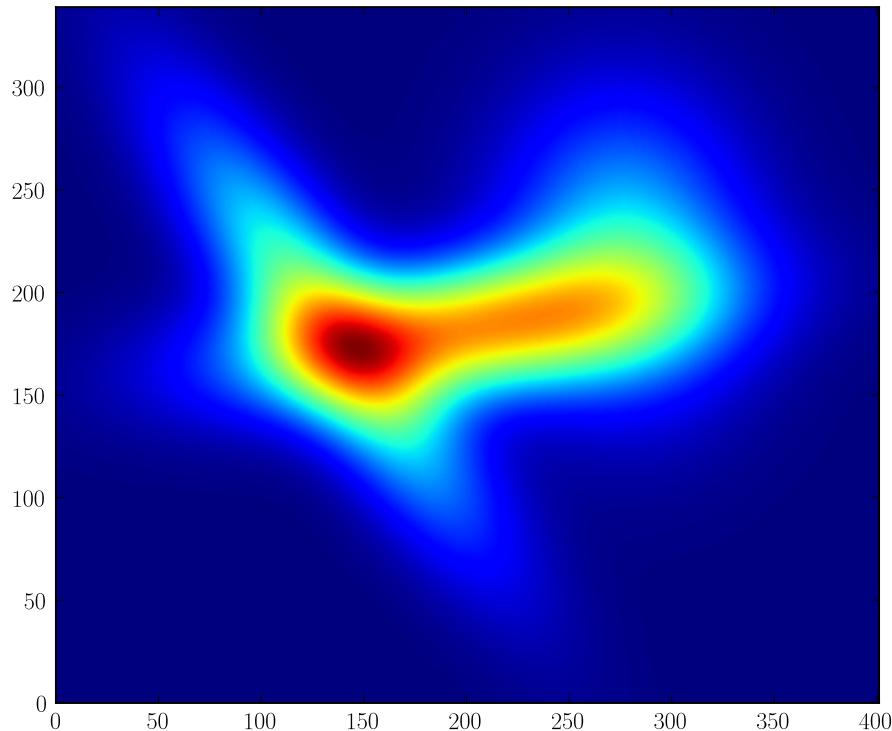


Figure 29.2: GMM density over the Baltimore region.

For multivariate data, the KDE is

$$\hat{f}_{\mathbf{H}}(x) = \frac{1}{n} \sum_{i=1}^n K_{\mathbf{H}}(x - x_i) = \frac{1}{n|\mathbf{H}|^{\frac{1}{2}}} \sum_{i=1}^n K(\mathbf{H}^{-\frac{1}{2}}(x - x_i))$$

where \mathbf{H} is a bandwidth matrix, and K is some multivariate symmetric density centered at the origin. Choosing \mathbf{H} is even more difficult in the multivariate case, but decent results can be obtained by choosing the diagonal matrix \mathbf{H} where

$$H_{ii} = \left(\frac{4}{d+2}\right)^{\frac{2}{d+4}} \cdot n^{\frac{-2}{d+4}} \cdot \hat{\sigma}_i^2$$

where d is the dimension of the data, and $\hat{\sigma}_i$ is the standard deviation of the i^{th} dimension. We will use `scipy`'s implementation of the KDE.

The `gaussian_kde` class is initialized with data where each column is a sample, the number of rows being the dimension of the space, and the number of columns being the number of samples. We can evaluate the density of a new sample using the method `evaluate`.

```
>>> import scipy as sp
>>> from scipy import stats
>>> kernel = stats.gaussian_kde(data)
```

```
>>> print kernel.evaluate(sp.zeros(data.shape[0]))
```

Problem 3. Using `stats.gaussian_kde`, initialize a KDE with the homicide data (make sure it's in the right format!), and compute the KDE of each point on your grid of Baltimore. Plot the density and save your plot.

With this approach, we can get an alternative (and likely better) probabilistic view of crime in Baltimore, as shown in Figure 29.3.

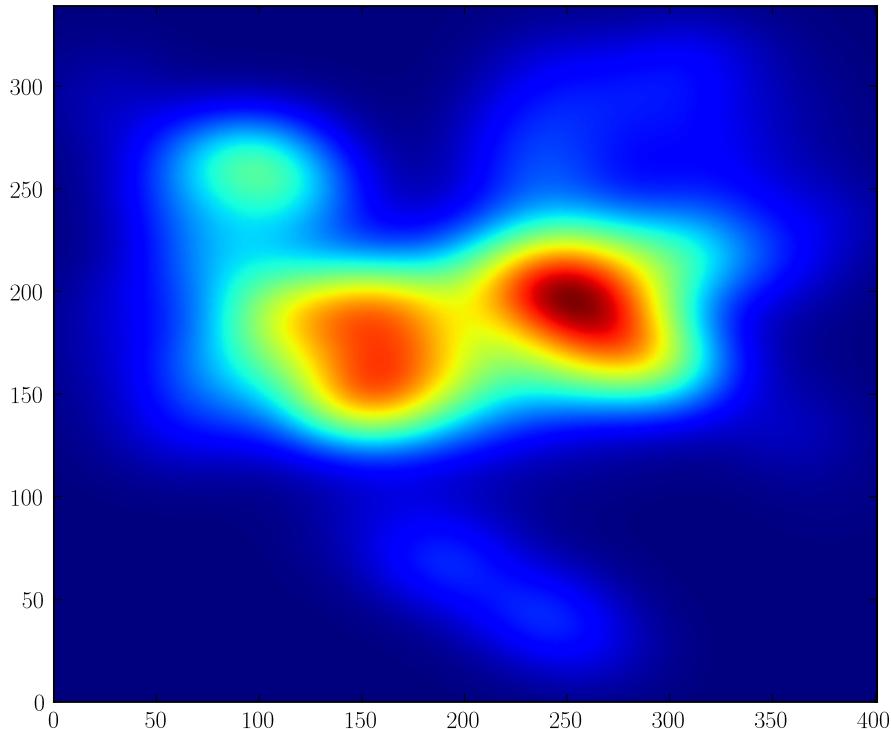


Figure 29.3: KDE over the Baltimore region.

We must still solve the initial problem, i.e. find the smallest geographical area of Baltimore containing 95% of all homicides. We do this by “vertically integrating” our densities. In other words, we choose some threshold t , find all points x on our grid such that $\hat{f}_{\mathbf{H}}(x) \geq t$, and compute the integral of our KDE over this region. If the length of each side for each square of our grid is h , then we can estimate this integral by

$$\sum_{x: \hat{f}_{\mathbf{H}}(x) > t} \hat{f}_{\mathbf{H}}(x) \cdot h^2$$

Ultimately, we are trying to find the region where this integral is approximately 0.95. Since t is the only thing we can vary, this becomes an optimization problem over t , where we are minimizing the objective function

$$g(t) = \left| \sum_{x:\hat{f}_{\mathbf{H}}(x)>t} \hat{f}_{\mathbf{H}}(x) \cdot h^2 - 0.95 \right|$$

Problem 4. Find this 95% region for both the GMM and the KDE and plot it using `imshow`, where each “safe” point on the grid is 0 and each “unsafe” region is 1.

30 Image Recognition Tasks

Lab Objective: *Use the KNN and SVM algorithms to solve two image recognition problems.*

Two important image recognition problems are character recognition and face recognition. The first is generally framed as a post office problem. Every day, millions of pieces of mail are sent through the US Postal Service each day. This requires an automated way of routing much of the mail. The problem is to automatically determine the zip code of the addressee for a piece of mail. There are two parts to this problem: find the zip code on the letter, and then determine what it is. We will only consider the second part in this lab.

Given that we have an image of a single digit, how can we decide what it is without human intervention? This is a classification problem, with the classes being the digits 0 through 9. We will use both the KNN and SVM classifiers to predict each digit.

We will use the `digits` data set from `sklearn.datasets` for our data, using the method `sklearn.datasets.load_digits()`. Each sample is an 8×8 image which has been flattened into a length-64 vector.

Problem 1. Load the digits data and separate it into a training set and a test set.

The module `sklearn.neighbors` has a nice class `KNeighborsClassifier` that implements the KNN classifier.

Problem 2. Find and read some of the documentation for the aforementioned class. Implement a KNN classifier on the training set. What is the misclassification rate on your test set?

While the SVM was originally designed as a binary classifier, it has been extended into a multi-class classifier as well. We won't go into the details here, but one common extension is to train K different SVMs (where K is the number of classes), each being a "one-versus-all" classifier. After some calibration, a new sample is predicted to be the class k where $f_k(\mathbf{x})$ is greatest, f_k being the function defining the hyperplane for class k against all other classes. Again, `sklearn.svm` has a nice class `SVC` that implements this.

The module `sklearn.grid_search` provides a nice way to find the classifier that performs the best on the training set, considering a grid of parameters. We will use this to help us find an optimal SVM for the digits data set, where we use a radial basis function for the kernel.

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5], 'gamma': [0.0001, 0.0005, ←
    0.001, 0.005, 0.01, 0.1],}
>>> clf = GridSearchCV(SVC(kernel='rbf', class_weight='auto'), param_grid)
>>> clf = clf.fit(training_data, training_target)
```

Problem 3. Predict the values for the test set using the SVM we just trained. What is your misclassification rate? How does the SVM's performance compare to the KNN classifier?

We now consider our second image recognition problem: face recognition. This is much harder than simply recognizing a digit on a white envelope, as there is bound to be so much more noise and variation.

We use as our data set the “Labeled Faces in the Wild”, a database of face photographs. In fact, we will only consider a small subset of this database, including only images of Ariel Sharon, Colin Powell, Donald Rumsfeld, George W Bush, Gerhard Schroeder, Hugo Chavez, and Tony Blair. Through `sklearn.datasets` we can download and process this data set rather easily, though it might take some time.

```
>>> from sklearn.datasets import fetch_lfw_people
>>> people = fetch_lfw_people(min_faces=70, resize=0.4)
>>> data = people.data
>>> target = lfw_people.target
```

This data set consists of 1288 images of size 50×37 , each flattened. The targets are digits from 0 to 6, corresponding with the ordered names above. This might seem counterintuitive, considering the main ideas of SVMs, but it is sometimes useful to reduce the dimensionality of our feature space before implementing an SVM, using PCA so we can retain as much information as possible while still reducing the dimensionality.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=150, whiten=True).fit(data)
>>> data_pca = pca.transform(data)
```

Problem 4. Separate the data set into training data and test data. Create a PCA object fit to the training data. With this object, reduce the dimensionality of both the training data and the test data.

Problem 5. Train an SVM on the image set with the same parameter grid search used above. Label the test data. What is your misclassification rate?

Let's also compare and see how well the KNN classifier does on this data set.

Problem 6. Train a KNN classifier on the image data set. What is your misclassification rate on the test data? Does this surprise you?

This allows us to end this course with a very important take-home message: the No Free Lunch Theorem. In essence, this theorem states that there is no single machine learning classifier to rule them all—each has its strengths and weaknesses. More specifically, if there is a ML classifier that outperforms all other classifiers on a data set, then we can find a data set where a different classifier will be superior. This means that we have to be intelligent in how we choose which classifiers to try, because there isn't any “go-to” classifier that will always work.

31

K-Means Clustering

Lab Objective: *Understand the basics of k-means clustering, and apply to the problem of clustering earthquake epicenters.*

Clustering

In Lab 23, we analyzed the iris dataset using PCA; we have reproduced the first two principal components of the iris data in Figure 31.1. Upon inspection, a human can easily see that there are two very distinct groups of irises. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*.

The objective of clustering is to find a partition 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 .

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 μ_i is the mean of the elements in S_i , i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

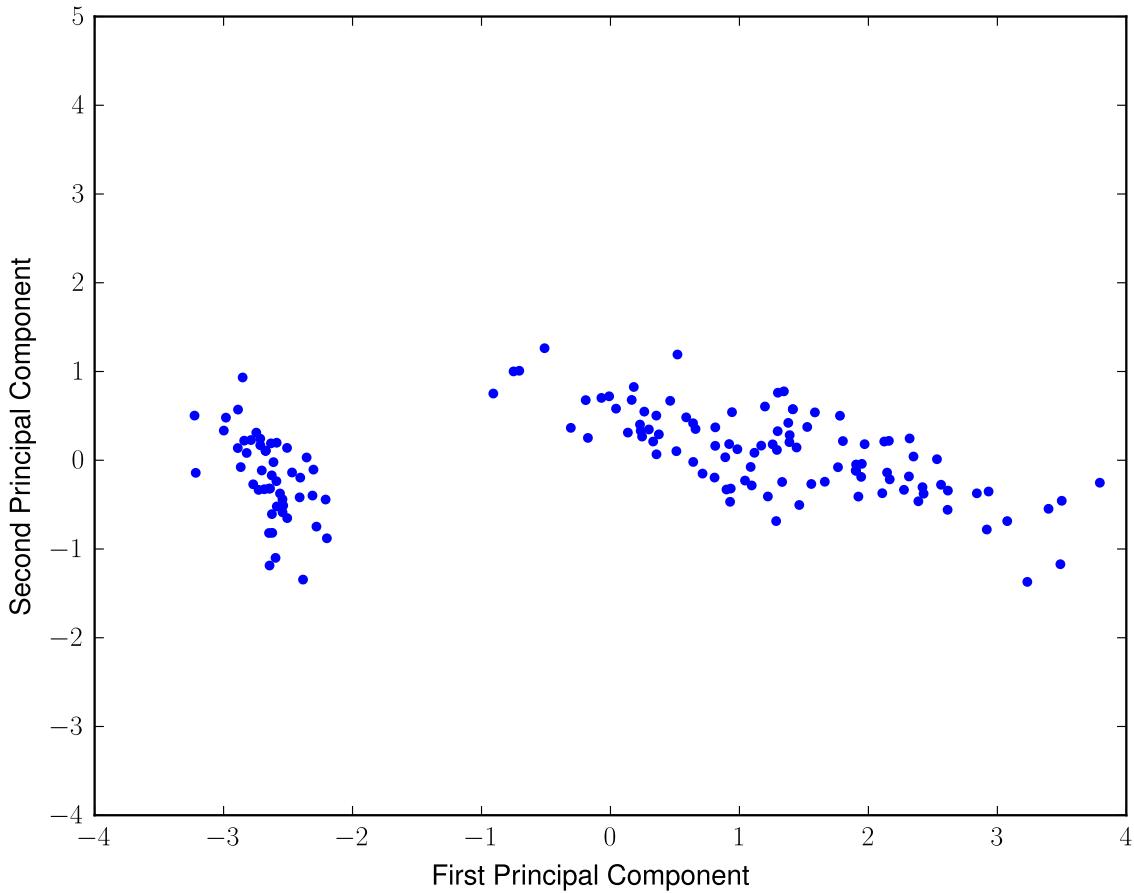


Figure 31.1: The first two principal components of the iris dataset.

The K-Means Method

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.

Examine Figure 31.2, which 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.

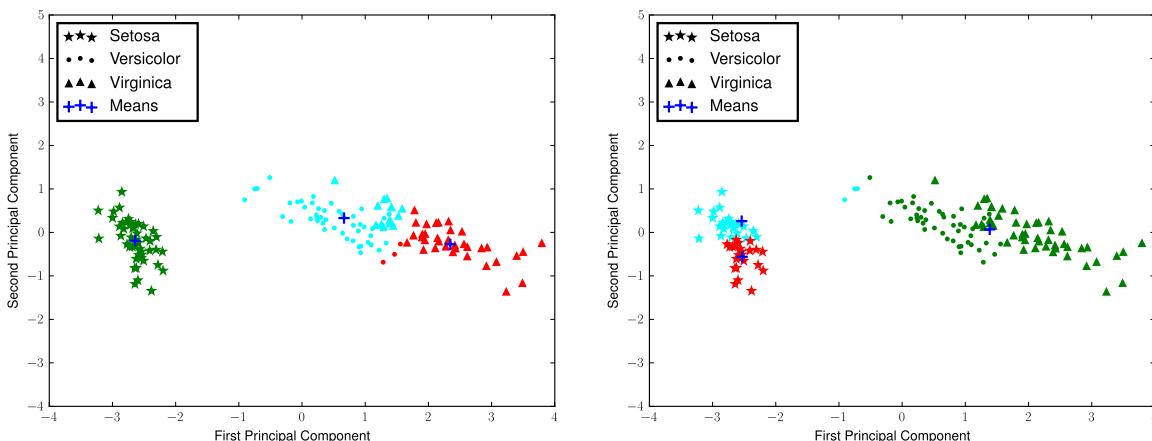


Figure 31.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

Problem 1. Implement the *k-means* algorithm using the following function declaration.

```
def kmeans(data,n_clusters,init='random',max_iter=300):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    data : ndarray of shape (n,k)
        Each row is an observation.
    n_clusters : int
        The number of clusters.
    init : string or ndarray of shape (n_clusters,k)
        If init is the string 'random', then randomly initialize the ←
        cluster means.
        Else, the initial cluster means are given by the rows of init.
    max_iter : int
        The maximum allowable number of iterations.

    Returns
    -----
```

```
-----
means : ndarray of shape (n_cluster,k)
    The final cluster means, given as the rows.
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 relative to
    the rows of means.
measure : float
    The within-cluster sum of squares quality measure.
...
pass
```

Test your function on the first two principal components of the iris dataset. Run it 10 times, using a different random initialization of the means each time. Retain the clustering with the smallest within-cluster sum of squares. Your clustering should be similar to the first clustering in Figure 31.2.

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.

Our data is contained in 6 text files, each with earthquake data throughout the world covering a time period of one month, giving us data from January 2010 through June 2010. These files contain a lot of information which isn't of interest to us at the present time; all we would like to extract from them is the location of each earthquake, which appears in characters 21 through 33 of each line. Characters 21 through 26 contain the latitude of each epicenter, character 26 denoting North or South, and characters 27 through 33 contain the longitude of each epicenter, character 33 denoting East or West. We need to divide each value by 1,000 to represent these as degrees and decimals.

Problem 2. Load the earthquake data into a $n \times 2$ array, where each row gives the longitude and latitude of an earthquake in degrees. Multiply South latitudes and West longitudes by -1 . Create a scatter plot of the resulting data. You should be able to see the outlines of some of the continents and tectonic plates (since these are often areas of significant seismic activity). Your plot should match Figure 31.3.

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. We must recognize that latitude and longitude are best viewed as a variation of spherical coordinates in \mathbb{R}^3 , and we should interpret them as such. Since our *k-means* algorithm is based on Euclidean distance, we need to transform our data into 3-dimensional Euclidean coordinates.

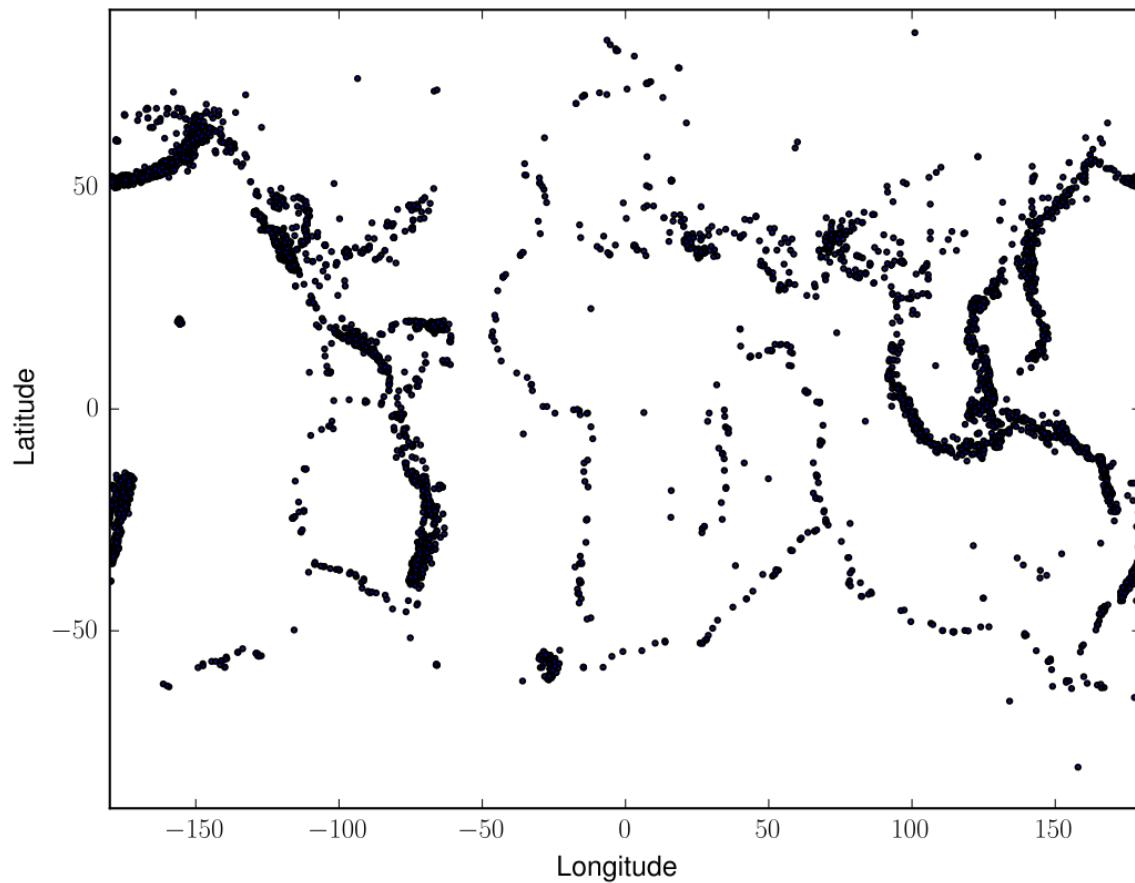


Figure 31.3: Earthquake epicenters over a 6 month period.

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, θ, φ) , where r is the distance from the origin, θ is the radial angle in the xy -plane from the x -axis, and φ is the angle from the z -axis. In our earthquake data, the longitude is already the appropriate θ value, and the φ value (in degrees) is simply 90° minus the latitude. 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:

$$\begin{array}{ll} r = \sqrt{x^2 + y^2 + z^2} & x = r \sin \varphi \cos \theta \\ \varphi = \arccos \frac{z}{r} & y = r \sin \varphi \sin \theta \\ \theta = \arctan \frac{y}{x} & z = r \cos \varphi \end{array}$$

Problem 3. Transform your earthquake data into three dimensional Euclidean coordinates. Be sure to consider if and when you need to transform your data from degrees to radians.

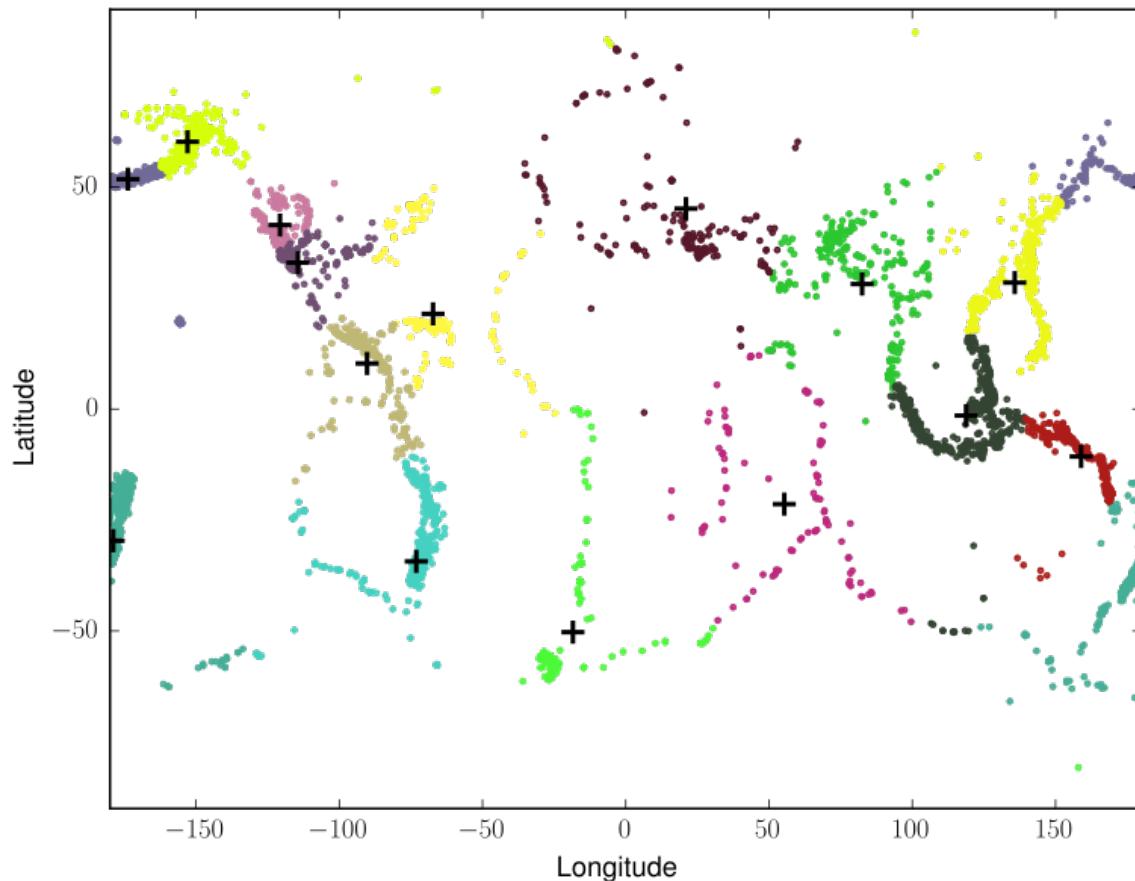


Figure 31.4: Earthquake epicenter clusters with $N = 15$.

We are now ready to cluster the earthquake data using the Euclidean coordinates. We need to address one further issue, however. Notice that each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. We also need to ensure that our cluster means have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth. Furthermore, 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. Thus, we need to add optional functionality to our `kmeans` function.

Problem 4. Add a keyword argument `normalize=False` to your `kmeans` function, and add code to normalize the means at each iteration, should this argument be set to `True`. Use your function to cluster the earthquake data into 15 clusters. Run this 10 times, keeping the best clustering. Transform the cluster means back to latitude and longitude coordinates (when calculating θ using the inverse tangent, use `numpy.arctan2` or `math.atan2`, so that that correct quadrant is chosen). Create a scatter plot showing each cluster mean, along with the earthquake epicenters color-coded according to their cluster. Your plot should resemble that of Figure 31.4.

Though plotting our results in two dimensions gives us a good picture, we can see that this is not entirely accurate. There are points that appear to be closer to a different cluster center than the one to which they belong. This comes from viewing the results in only two dimensions. When viewing in three dimensions, we can see more clearly the accuracy of our results.

Problem 5. Add a keyword argument `3d=False` to your `kmeans` function, and add code to show the three-dimensional plot instead of the two-dimensional scatter plot should this argument be set to `True`. Maintain the same color-coding scheme as before. Use `mpl_toolkits.mplot3d.Axes3D` to make your plot.

Spectral Clustering

We now turn to another method for solving a clustering problem, namely that of Spectral Clustering. As you can see in Figure ???, 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 σ .

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 ϵ 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`.

Problem 6. Implement the Spectral Clustering Algorithm by calling your `kmeans` function, using the following function declaration:

```
def specClus(measure,Laplacian,args,arg1=None,kiters=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.
    kiters : 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.

Problem 7. Create a function that will return the accuracy of your spectral clustering implementation, as follows:

```
def test_specClus(location,measure,Laplacian,args,arg1=None,kiters=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.
    kiters : 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
```


32

Bayesian Search

Lab Objective: *Understand how Bayesian methods can be used for search and rescue operations.*

In the past half century, statistical methods have seen a rise in popularity for search and rescue problems. *Bayesian search theory* has been successfully used to find lost sea vessels and aircraft, including the USS *Scorpion*, the MV *Derbyshire*, the SS *Central America*, and Air France Flight (AF) 447. It was also used to search for a lost nuclear bomb after a B-52 bomber crashed in Spain in 1966.

In this lab, we will simulate search procedures for the AF447 disaster, using our knowledge of Bayes' Theorem and estimations of ocean depth in the vicinity of the last point of radio contact with the flights captains.

On June 1st, 2009, AF447 departed Rio de Janeiro, Brazil, flying to Paris, France. Air traffic controllers lost contact with the aircraft at 2:10 AM, nearly halfway between South America and Western Africa. The location of final contact was approximately 2.98°N, 30.59°W. Figure ?? shows the flight path (and part of the planned flight path) of AF447.

The search operations proved unsuccessful for nearly two years. After an unsuccessful first year of searching, the consulting company Metron was asked to develop a probabilistic search procedure to find the flight. Using Bayesian search methods, a large portion of AF447's debris field was discovered within a week of resuming the search. Within a month, the black boxes were recovered.

Bayesian search theory consists of combining our belief of where a lost vessel should be located with our belief of our success of finding it in a location, should we search there. A simple way of expressing our belief of where a lost vessel should be found is with a probability distribution, centered at the last point of contact. Our belief of successfully finding the vessel in a given location can be a function of ocean depth (or, for a land based search, a function of density of foliage, difficulty of searching a mountainous region, etc.). We simplify this by discretizing the possible locations into a square grid.

Formally, let A_i be the event that we successfully find the vessel when we search in location i , and let B_i be the event that the vessel is in location i . Then let p be a probability distribution for the location of the vessel over the grid, so

$$p_i = \mathbb{P}(B_i).$$

Let q be a set of N Bernoulli parameters, each denoting the probability of success if we search in its location, i.e.

$$q_i = \mathbb{P}(A_i|B_i).$$



Figure 32.1: Flight path of AF447.

We also assume no chance of success in finding the vessel in location i if it is *not* located there.

This allows us to compute

$$\begin{aligned}
 r_i &= \mathbb{P}(A_i) \\
 &= \sum_j \mathbb{P}(A_i \cap B_j) \\
 &= \sum_j \mathbb{P}(A_i|B_j)\mathbb{P}(B_j) \\
 &= \mathbb{P}(A_i|B_i)\mathbb{P}(B_i) \\
 &= p_i q_i
 \end{aligned}$$

We proceed by searching in location k , where $k = \text{argmax}_i r_i$.

Problem 1. Write a function that accepts two 20×20 arrays, p and q , and returns the index pair i, j of the next search location.

If a search successful, then our search is over. If unsuccessful, we update our probabilities p_i given our recent data θ (our unsuccessful attempt) according to Bayes Rule, which in this case, is

$$\mathbb{P}(B_i | \theta) = \frac{\mathbb{P}(\theta | B_i)\mathbb{P}(B_i)}{\mathbb{P}(\theta | B_i)\mathbb{P}(B_i) + \mathbb{P}(\theta | B_i^c)\mathbb{P}(B_i^c)}$$

This has two different solutions, depending on where we searched, yielding the following for our posterior probability:

$$\tilde{p}_i = \begin{cases} \frac{(1-q_i)p_i}{(1-q_i)p_i + (1-p_i)} = p_i \frac{1-q_i}{1-p_i q_i} & \text{if we searched in location } i \\ \frac{p_i}{(1-q_i)p_i + (1-p_i)} = \frac{p_i}{1-p_i q_i} & \text{if we searched in location } j \neq i \end{cases}$$

Problem 2. Write a function that accepts two 20×20 arrays, p and q , as well as an index pair i, j of search coordinates for the most recent unsuccessful search, and returns the posterior probabilities $\tilde{p}_{k,l}$ for each possible location k, l , where $1 \leq k, l \leq 20$.

Using our computed posterior probability and our success probabilities, we compute r again, and choose a search location, continuing this procedure until success.

Problem 3. Write a function to simulate the search procedure. It should accept two 20×20 arrays, p and q , where p is our initial prior on the location of the vessel and q contains the probability of success for each grid location. It should also accept the actual location i, j of the vessel. During the simulation process, assume that q is correct, i.e. given that we search in location i, j , we will successfully find the vessel with probability $q_{i,j}$. The function should return the number of search iterations until success, as well as the location of the vessel (but don't hard code this!).

We have roughly examined the ocean depths in the vicinity of the crash, and compute probabilities of successfully finding it in a location, given that we search there (this is simply a function of depth). We have also computed an initial prior on the location, with locations nearer the final point of contact having higher probabilities than locations further away. These are represented as 20×20 arrays, and are located in the files `depthProbs` and `prior`, respectively.

Problem 4. Unpickle the two files mentioned above, and test your previous function. Once it's working properly, write a function that runs the previous function `n_sim` times, and prints out the shortest search length, longest search length, and average search length. Test it with `n_sim = 500` at locations 9, 10 and 12, 5.