1 Pandas 1: Introduction

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

Pandas Basics

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

Pandas Data Structures

Series

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

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

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

DataFrame

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

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

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

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

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

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

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

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

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

```
# View the DataFrame as a NumPy array
```

```
>>> grades.values
array([[ 52.,  73.],
       [ 10.,  39.],
       [ 35.,  nan],
       [ nan,  26.],
       [ nan,  99.],
       [ 81.,  68.]])
```

Data I/O

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

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

Table 1.1: Methods for exporting data in a pandas `Series` or `DataFrame`.

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

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

Another particularly useful function is `read_html()`, which is useful when scraping data. It takes in a url or html file and an optional argument `match`, a string or regex, and returns a list of the tables that match the `match` in a DataFrame. While the resulting data will probably need to be cleaned, it is frequently much faster than scraping a website.

Data Manipulation

Accessing Data

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

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

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

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

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

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

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

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

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

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

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

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

Other methods used for manipulating `DataFrame` and `Series` panda structures can be found in Table 1.2.

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

Table 1.2: Methods for managing or modifying data in a pandas `Series` or `DataFrame`.

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

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

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

Basic Data Manipulation

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

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

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

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

In addition to arithmetic, `Series` has a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 1.3.

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

Table 1.3: Numerical methods of the `Series` and `DataFrame` pandas classes.

Basic Statistical Functions

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

```
# Use describe to better understand the data
>>> grades.describe()
            Math   English  History
count   4.000000   5.00000      6.0
mean   44.500000  61.00000    100.0
std    29.827281  28.92231      0.0
min    10.000000  26.00000    100.0
25%    28.750000  39.00000    100.0
50%    43.500000  68.00000    100.0
75%    59.250000  73.00000    100.0
max    81.000000  99.00000    100.0
```

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

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

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

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

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

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

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

Dealing with Missing Data

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

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

```
Mark      81.0     68.0    100.0

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

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

ACHTUNG!

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

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

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

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

Complex Operations in Pandas

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

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

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

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

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
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)
```

Masks

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

```
# Get the ID column from studentInfo
>>> studentInfo.ID # or studentInfo['ID']
   ID
0   0
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
```

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

```
# Get the ID and Age columns.
>>> studentInfo[['ID', 'Age']]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   29
```

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

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

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

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

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

1. Identify the three crimes that have a mean yearly number of occurences over 1,500,000. Of these three crimes, which two are very correlated? Which of these two crimes has a greater maximum value? Save the title of this column as a variable to return as the answer.
2. Examine the data from 2000 and later. Sort this data (in ascending order) according to number of murders. Find the years where aggravated assault is greater than 850,000. Save the indices (the years) of the masked and reordered `DataFrame` as a NumPy array to return as the answer.
3. What year had the highest crime rate? In this year, which crime was committed the most? What percentage of the total crime that year was it? Save this value as a float.

Working with Dates and Times

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

```
>>> from datetime import datetime

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

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

The `datetime.datetime` object has a parser method, `strptime()`, that converts a string into a new `datetime.datetime` object. The parser is flexible so the user must specify the format that the dates are in. For example, if the dates are in the format `"Month/Day//Year::Hour"`, specify `format"=%m/%d//%Y::%H"` to parse the string appropriately. See Table 1.4 for formatting options.

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

Table 1.4: Formats recognized by `datetime.strptime()`

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

Converting Dates to an Index

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

```
>>> import pandas as pd

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

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

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

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

Generating Time-based Indices

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

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

Table 1.5: Parameters for `pd.date_range()`.

Exactly three of the parameters `start`, `end`, `periods`, and `freq` must be specified to generate a range of dates. The `freq` parameter accepts a variety of string representations, referred to as *offset aliases*. See Table 1.6 for a sampling of some of the options. For a complete list of the options, see `https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliasesl`.

Parameter	Description
`"D"`	calendar daily (default)
`"B"`	business daily (every business day)
`"H"`	hourly
`"T"`	minutely
`"S"`	secondly
`"MS"`	first day of the month (Month Start)
`"BMS"`	first business day of the month (Business Month Start)
`"W-MON"`	every Monday (Week-Monday)
`"WOM-3FRI"`	every 3rd Friday of the month (Week of the Month - 3rd Friday)

Table 1.6: Options for the `freq` parameter to `pd.date_range()`.

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

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

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


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

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

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

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

Elementary Time Series Analysis

Shifting

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

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

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

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

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

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

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

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

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

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

Return the DateTimeIndex of the day with the largest gain and the day with the largest loss.

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

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

Additional Material

SQL Operations in pandas

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

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

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

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

```
# SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   29

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

```
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4

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

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

```
# SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = mathInfo.ID
>>> pd.merge(studentInfo, mathInfo, on='ID') # 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]
```

More Datetime with Pandas

Periods

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

The `Period` class accepts a `value` and a `freq`. The `value` parameter indicates the label for a given `Period`. This label is tied to the **end** of the defined `Period`. The `freq` indicates the length of the `Period` and in some cases can also indicate the offset of the `Period`. The default value for `freq` is "M" for months. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 1.6.

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

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

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

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

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

# Get every three months form 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')
```

The bounds of a `PeriodIndex` object can be shifted by adding or subtracting an integer. `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. The `how` parameter can be `start` or `end` and determines if the timestamp is the beginning or the end of the period. Similarly, you can switch from timestamps to periods.

```
# Convert to timestamp (last day of each quarter)
>>> 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')

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

Operations on Time Series

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

Slicing

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

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

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

```
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
2012-01-03  1.436843  0.596143

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

Resampling

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

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

To downsample, use the `resample()` method of the `Series` or `DataFrame`. This method is similar to `groupby()` in that it groups different entries together. Then aggregation produces a new data set. The first parameter to `resample()` is an offset string from Table 1.6: `"D"` for daily, `"H"` for hourly, and so on.

```
>>> import numpy as np

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

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

```
Freq: A-DEC, dtype: float64

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

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

Elementary Time Series Analysis

Rolling Functions and Exponentially-Weighted Moving Functions

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

Rolling Functions (Moving Window Functions)

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

```
# 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.

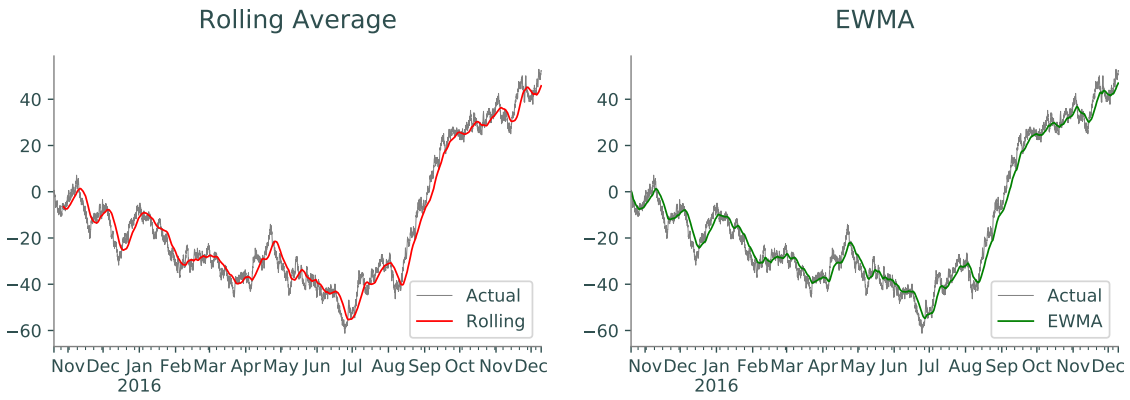
In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

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

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.



(a) (b)

Figure 1.1: Rolling average and EWMA.

```
ax2 = plt.subplot(122)
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")
```

1 Pandas 2: Plotting

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

Overview of Plotting Tools

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

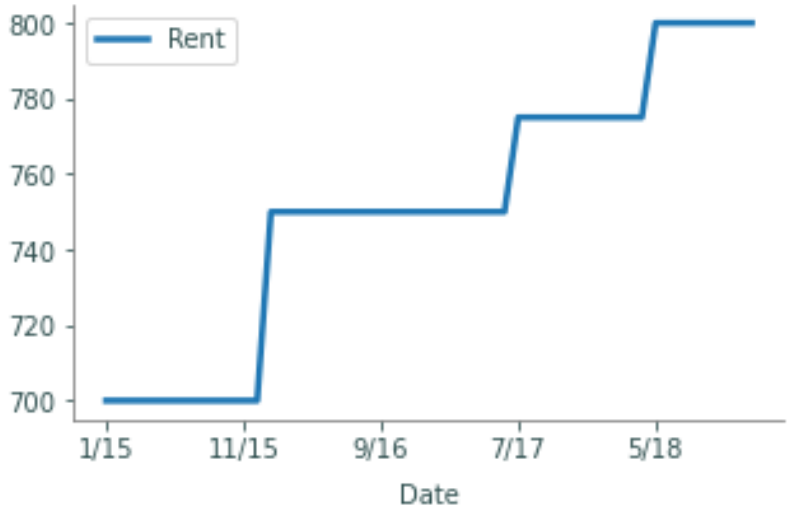
Plot Type	`plot()` ID	Uses and Advantages
Line plot	`"line"`	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	`"scatter"`	Compare exactly two data sets, independent of ordering
Bar plot	`"bar"`, `"barh"`	Compare categorical or sequential data
Histogram	`"hist"`	Show frequencies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

Table 1.1: Types of plots in pandas. The plot ID is the value of the keyword argument `kind`. That is, `df.plot(kind="scatter")` creates a scatter plot. The default `kind` is `"line"`.

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

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

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



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

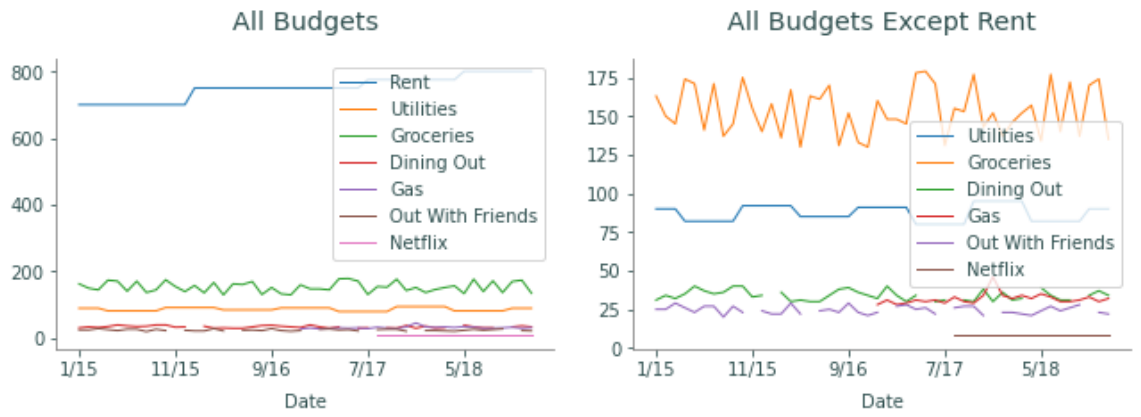
```
>>> plt.plot(budget.index, budget['Rent'], label='Rent')
>>> plt.xlabel(budget.index.name)
>>> plt.xlim(min(budget.index), max(budget.index))
>>> plt.legend(loc='best')
```

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

Visualizing an Entire Data Set

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

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



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

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

Figure 1.1

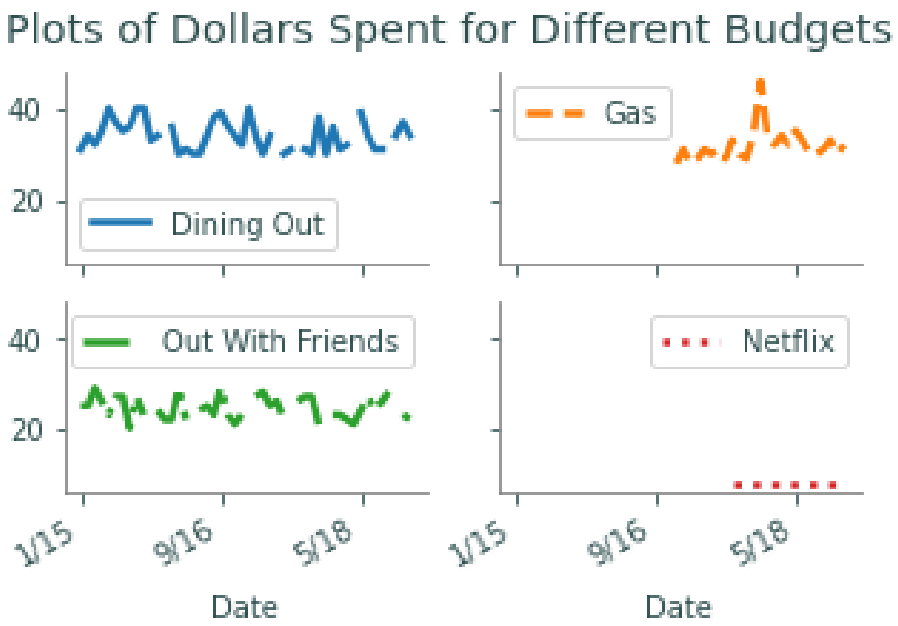
While plotting every `Series` at once can give an overview of all the data, the resulting plot is often difficult for the reader to understand. For example, the budget data set has 9 columns, so the resulting figure, Figure 1.1a, is fairly cluttered.

One way to declutter a visualization is to examine less data. Notice that `'Living Expenses'` has values much bigger than the other columns. Dropping this column, as well as `'Rent'`, gives a better overview of the data, shown in Figure 1.1b.

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

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

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



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

```
# Plot three histograms together.
>>> budget.plot(kind='hist',y=['Gas','Dining Out','Out With Friends'],
...     alpha=.7,bins=10,title="Frequency of Amount (in dollars) Spent")

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

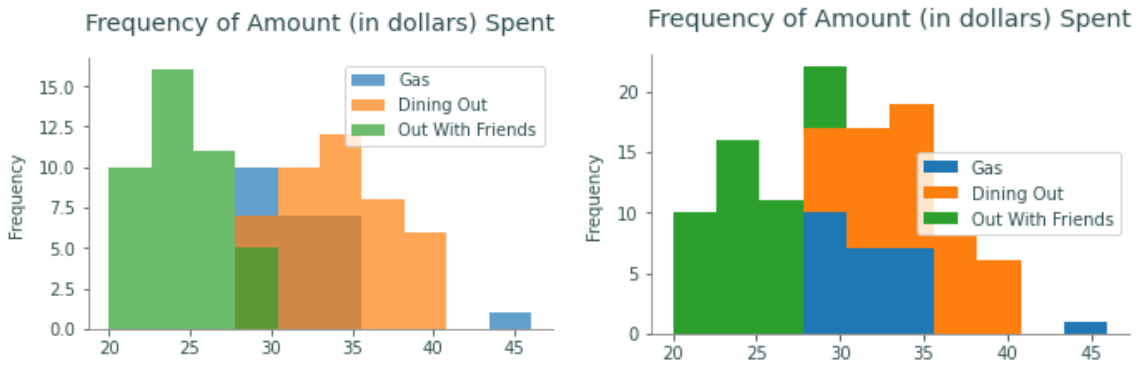


Figure 1.2: Two examples of histograms that are difficult to understand because multiple columns are plotted.

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

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

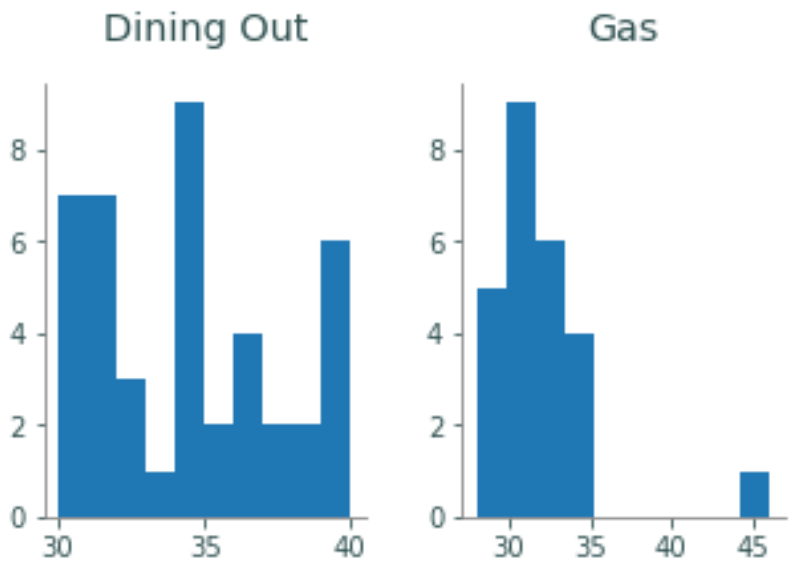


Figure 1.3: Histograms of `"Dining Out"` and `"Gas"`.

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

Patterns and Correlations

After visualizing the entire data set initially, a good next step is to closely compare related parts of the data. This can be done with different types of visualizations. For example, Figure 1.1b suggests that the `"Dining Out"` and `"Out With Friends"` columns are roughly on the same scale. Since this data is sequential (indexed by time), start by plotting these two columns against the index. Next, create a scatter plot of one of the columns versus the other to investigate correlations that are independent of the index. Unlike other types of plots, using `kind="scatter"` requires both `x` and `y` columns as arguments.

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

```
# Make a scatter plot of 'Dining Out' against 'Out With Friends'
>>> budget.plot(kind="scatter", x="Dining Out", y="Out With Friends",
...     alpha=.8,xlim=(0,max(budget['Dining Out'])+1),
...     ylim=(0,max(budget['Out With Friends'])+1))
```

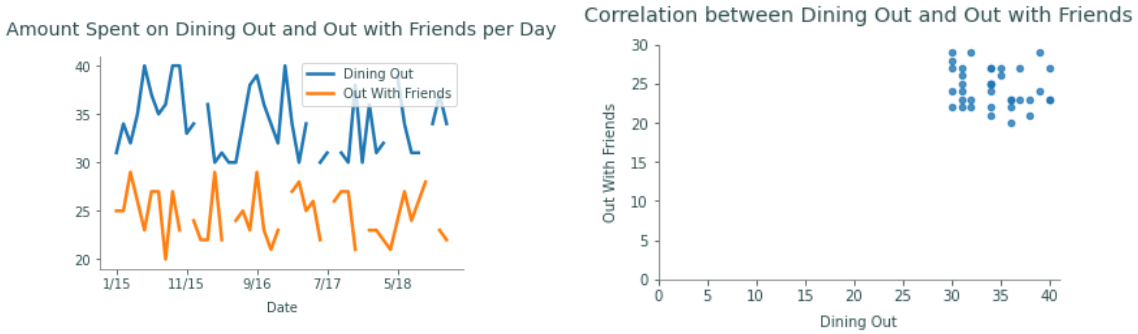


Figure 1.4: Correlations between `"Dining Out"` and `"Out With Friends"`.

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

Achtung!

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

The crime data set from Problem 1 is somewhat suspect in this regard. The 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.

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

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

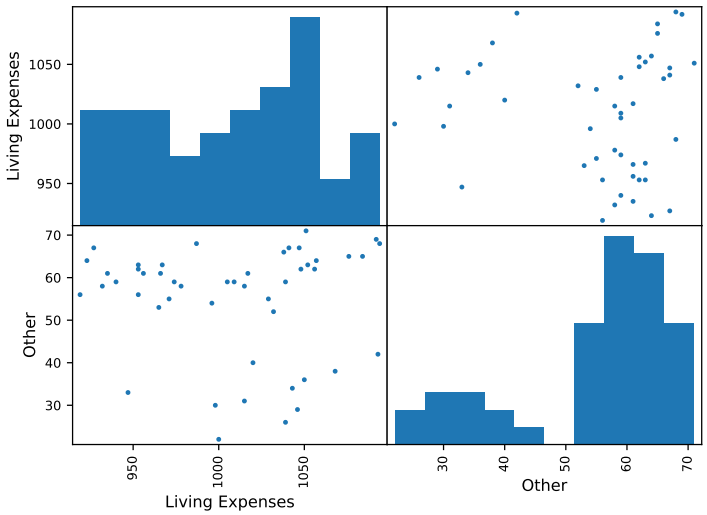


Figure 1.5: Scatter matrix comparing `"Living Expenses"` and `"Other"`.

Bar Graphs

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

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

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

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

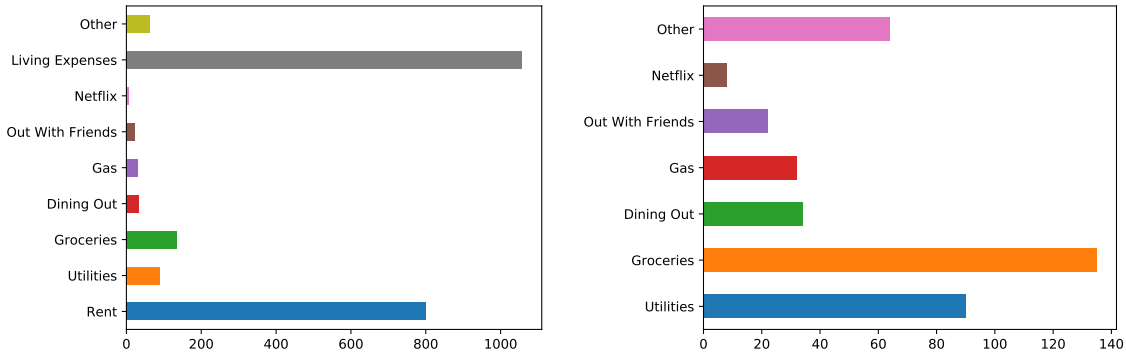


Figure 1.6: Bar graphs showing expenses paid in the last month of `budget`.

Problem 2. Using the crime data from the previous problem, identify if a trend exists between `Forcible Rape` and the following variables:

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

Make sure each graph is clearly labelled and readable. Return a tuple of booleans describing whether `Forcible Rape` correlates with each of the other variables.

Distributional Visualizations

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

```
# Compare the distributions of four columns.
>>> budget.plot(kind="box", y=["Gas","Dining Out","Out With Friends","Other"])

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

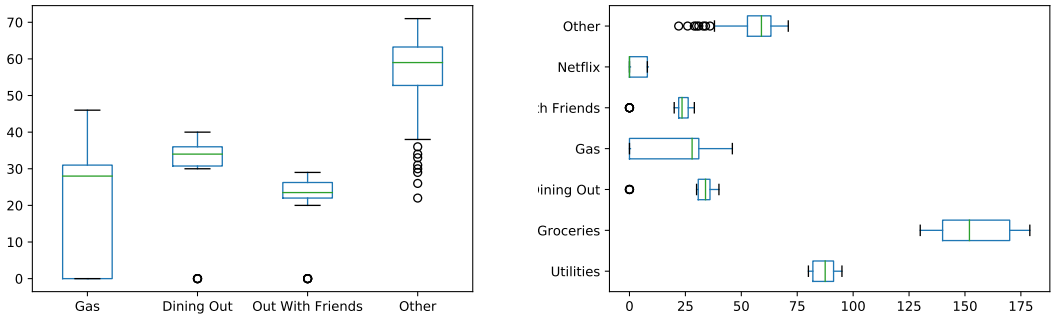


Figure 1.7: Vertical and horizontal box plots of `budget` dataset.

Hexbin Plots

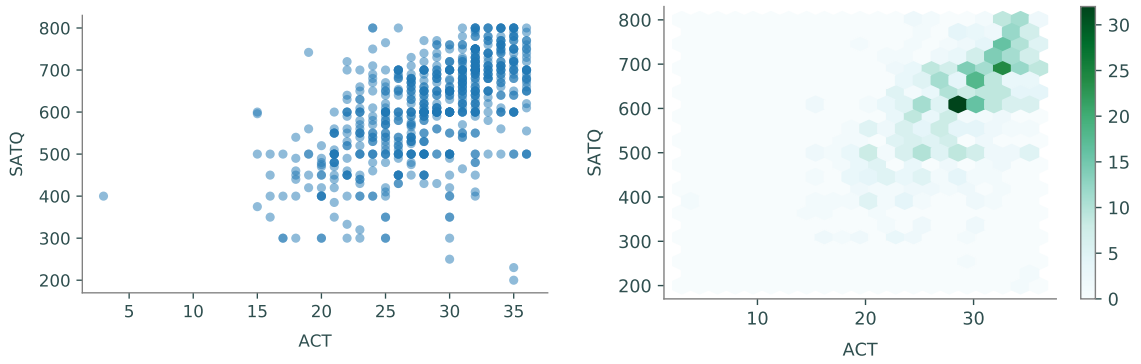
A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially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

The file `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 1.8a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 1.8b, 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)
```



(a) ACT vs. SAT Quant scores.

(b) Frequency of ACT vs. SAT Quant scores.

Figure 1.8: Scatter plots and hexbin plot of SAT and ACT scores.

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

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

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

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

1. The distributions of `Burglary`, `Violent`, and `Vehicle Theft`,
2. The distributions of `Vehicle Thefts` against the values of `Robbery`.

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

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

Principles of Good Data Visualization

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

Attention to Detail

Consider the plot in Figure 1.9. It is a scatter plot of positively correlated data of some kind, with `temp`–likely temperature–on the x axis and `cons` on the y axis. However, the picture is not really communicating anything about the dataset. It has not specified the units for the x or the y axis, nor does it tell what `cons` is. There is no title, and the source of the data is unknown.

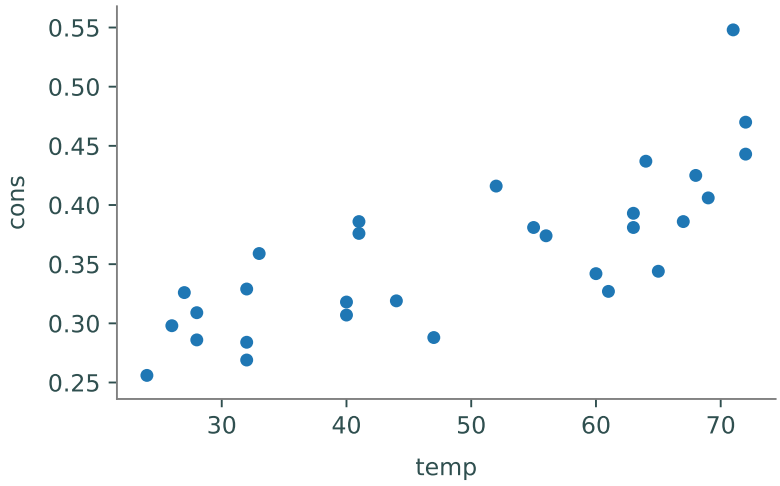


Figure 1.9: Non-specific data.

Labels and Citations

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

Consider again Figure 1.9. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> from pydataset import data
>>> icecream = data("Icecream")
```

```
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

This code produces the rather substandard plot in Figure 1.9. Examining the source of the dataset can give important details to create better plots. When plotting data, make sure to understand what the variable names represent and where the data was taken from. Use this information to create a more effective plot.

The ice cream data used in Figure 1.9 is better understood with the following information:

1. The dataset details ice cream consumption via 30 four-week periods from March 1951 to July 1953 in the United States.
2. `cons` corresponds to "consumption of ice cream per capita" and is measured in pints.
3. `income` is the family's weekly income in dollars.
4. `price` is the price of a pint of ice cream.
5. `temp` corresponds to temperature, degrees Fahrenheit.
6. The listed source is: "Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University."

This information gives important details that can be used in the following code. As seen in previous examples, pandas automatically generates legends when appropriate. Pandas also automatically labels the x and y axes, however our data frame column titles may be insufficient. Appropriate titles for the x and y axes must also list appropriate units. For example, the y axis should specify that the consumption is in units of *pints per 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 (Fahrenheit)")
>>> plt.ylabel("Consumption per head (pints)")
```

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

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

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

Finally, we have a clear and demonstrative graphic in Figure 1.10.

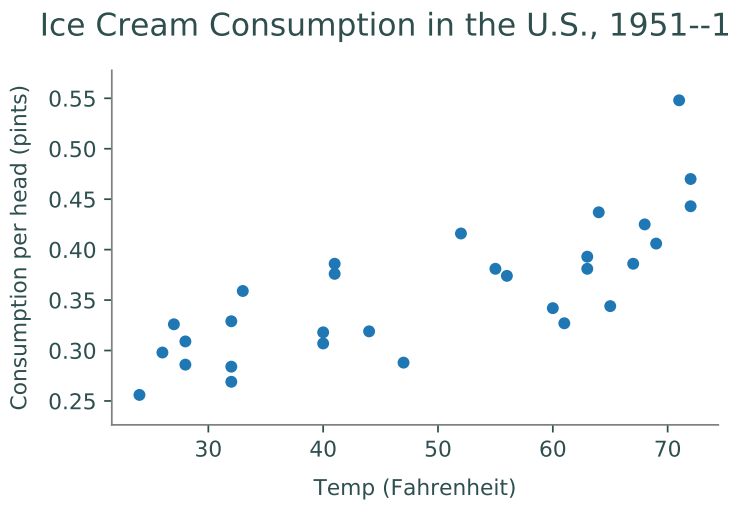


Figure 1.10: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

ACHTUNG!

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

Problem 4. The dataset `college.csv` contains information from 1995 on universities in the United States. To access information on variable names, go to `https://cran.r-project.org/web/packages/ISLR/ISLR.pdf`. Create 3 plots that compare variables or universities. These plots should answer questions about the data, e.g. what is the distribution of graduation rates or do schools with lower student to faculty ratios have higher tuition costs. These three plots should be easy to understand and have clear variable names and citations.

1 Pandas 3: Grouping

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

Groupby

The file `mammal_sleep.csv`[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")
```

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

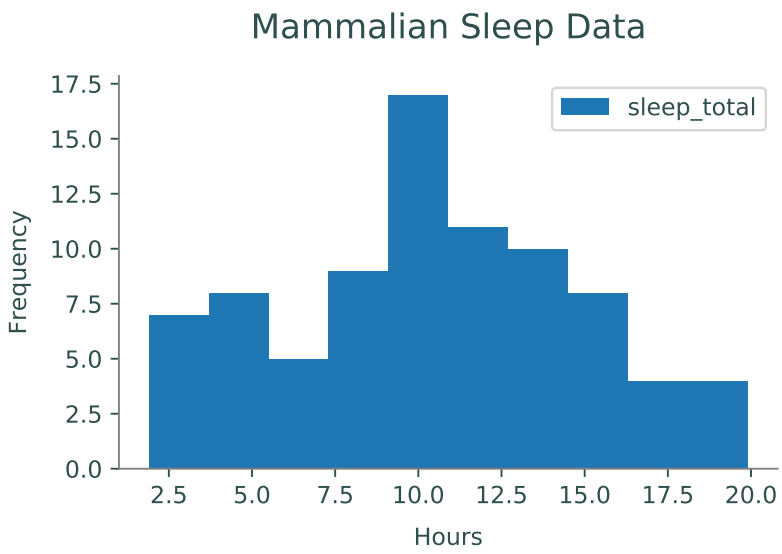


Figure 1.1: `"sleep_total"` frequencies from the mammalian sleep data set.

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

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

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

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

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

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

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

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

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

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

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

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

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

Determine whether private or public universities have a higher mean for each of these columns.

For the type of university with the higher mean, save the values of the describe function on said column as an array using `.values`. Return a tuple with these arrays in the order described above.

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

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

Visualizing Groups

There are a few ways that `groupby()` can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time using the `plot` method. The following visualization improves on Figure 1.1 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")
```

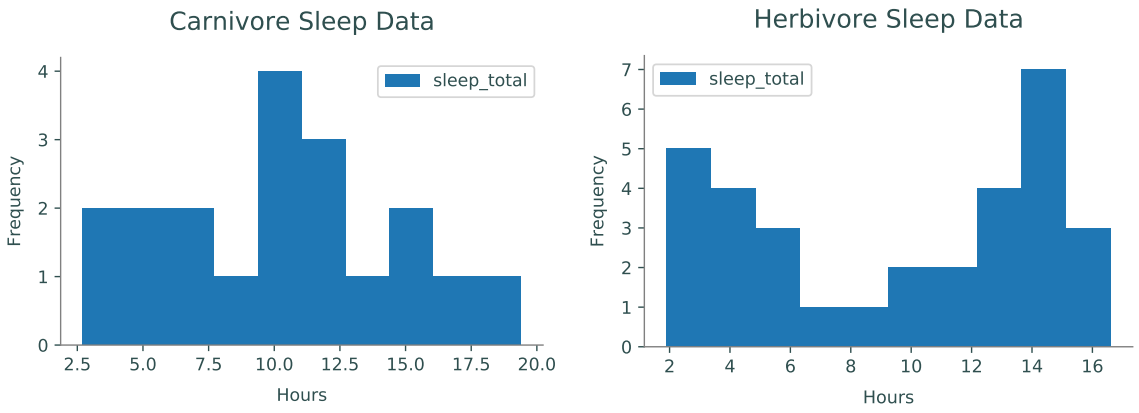
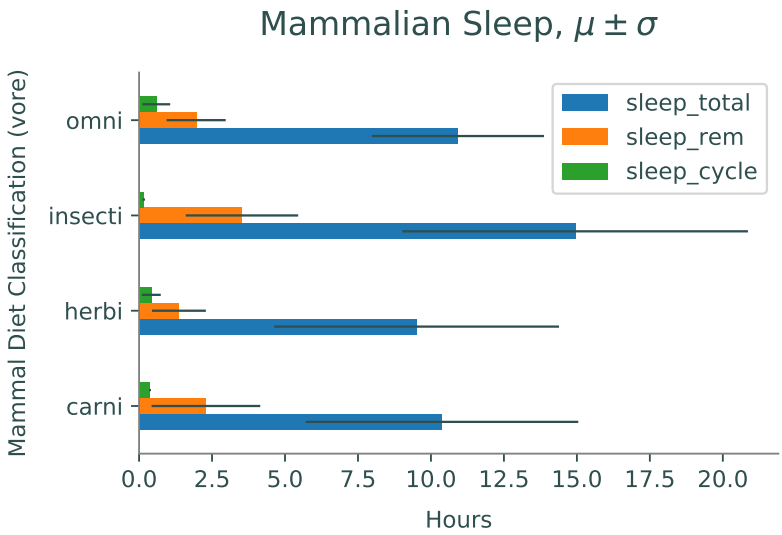


Figure 1.2: `"sleep_total"` histograms for two groups in the mammalian sleep data set.

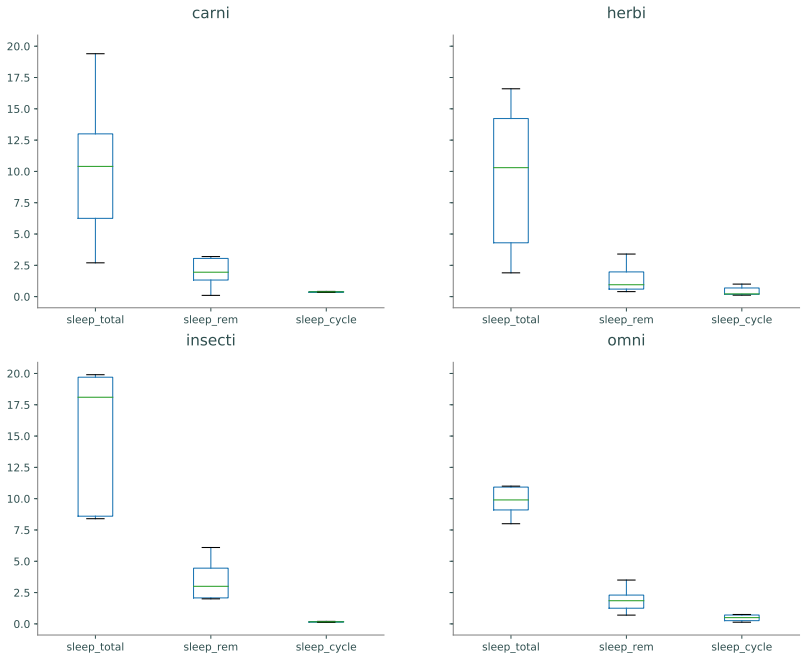
The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

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



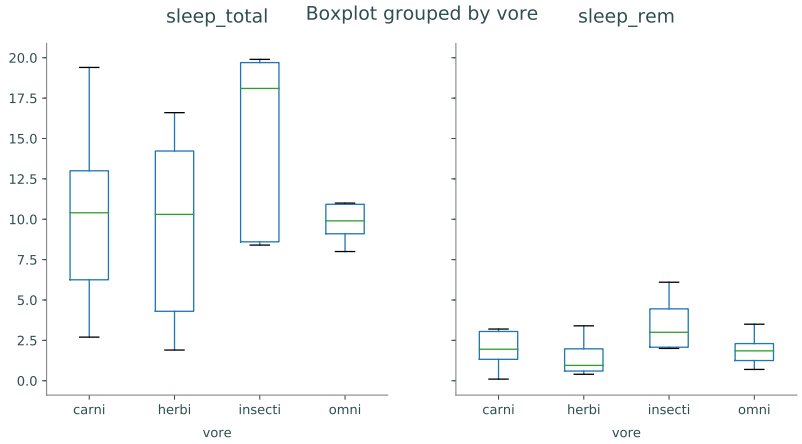
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

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



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

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



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

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

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

Pivot Tables

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

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

```
31    Red  Green  Female     7
21  Black   Blue  Female     9

>>> for col in ["Hair", "Eye", "Sex"]:      # Get unique values per column.
...     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. An example in the previous dataset would be if there were two or more rows in the original data for females with blond hair and blue eyes. To construct a pivot table, data of similar groups must be *aggregated* together in some way.

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

Consider the Titanic data set found in `titanic.csv`[2]. For this analysis, take only the "

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

Survived", "Pclass", "Sex", "Age", "Fare", and "Embarked" columns, replace null age values with the average age, then drop any rows that are missing data. To begin, we examine the average survival rate grouped by sex and passenger class.

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

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

Note

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

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

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

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

Return a tuple for each question (in order of the questions) where the first entry is the numerical code corresponding to the race and the second entry is corresponding to the sex.

Some useful keys in understand the data are as follows:

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

Discretizing Continuous Data

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

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

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(0, 4], (0, 4], (0, 4], (0,4], (0, 4], (4, 8], (4, 8], (4, 8]]
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 4. Use the employment data from Ohio in 1999 to answer the following questions:

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

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

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

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

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

1. Is there a correlation between the percent of alumni that donate and the amount the school spends per student in BOTH private and public universities?

2. Partition `Grad.Rate` into evenly spaced intervals of 20%. Is the partition with the greatest number of schools the same for private and public universities?
3. Does having a lower acceptance rate correlate with having more students from the top 10 percent of their high school class being admitted on average for BOTH private and public universities?
4. Why is the average percentage of students admitted from the top 10 percent of their high school class so high in private universities with very low acceptance rates? Use only the data to explain why; do not extrapolate.

1 GeoPandas

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

Installation

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

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

However, Geopandas can be notoriously difficult to install. This is especially the case if the python environment is not carefully maintained. Some of its dependencies can also be very difficult to install on certain systems. Because of this, using Colab for this lab is recommended; its environment is set up in a way that makes GeoPandas very easy to install. Otherwise, the GeoPandas documentation contains some additional options that can be used if installation difficulties occur: `https://geopandas.org/install.html`.

GeoSeries

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

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

A point is used to identify objects like coordinates, where there is one small instance of the object. A line could be used to describe objects such as roads. A polygon could be used to identify regions, such

as a country. Multipoints, multilines, and multipolygons contain lists of points, lines, and polygons, respectively.

Since each object in the GeoSeries is also a Shapely object, the GeoSeries inherits many methods and attributes of Shapely objects. Some of the key attributes and methods are listed in Table 1.1. These attributes and methods can be used to calculate distances, find the sizes of countries, and determine whether coordinates are within country's boundaries. The example below uses the attribute `bounds` to find the maximum and minimum coordinates of Egypt in a built-in GeoDataFrame.

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

Table 1.1: Attributes and Methods for GeoSeries

```
>>> import geopandas as gpd
>>> world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
# Get GeoSeries for Egypt
>>> egypt = world[world['name']=='Egypt']

# Find bounds of Egypt
>>> egypt.bounds
         minx    miny         maxx      maxy
47   24.70007    22.0     36.86623  31.58568
```

Creating GeoDataFrames

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

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

```
>>> import pandas as pd
>>> import geopandas as gpd
>>> from shapely.geometry import Point, Polygon

# Create a Pandas DataFrame
>>> df = pd.DataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
```

```
...                    'Latitude': [37.57, -12.05, -26.20],
...                    'Longitude': [126.98, -77.04, 28.04]})

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

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

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

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

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

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

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

```
# Both of these methods create the same GeoDataFrame as above
# Directly create the GeoDataFrame from the dictionary
>>> gdf = gpd.GeoDataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
...                    'Latitude': [37.57, -12.05, -26.20],
...                    'Longitude': [126.98, -77.04, 28.04]})
# Create geometry column and set as the geometry
>>> gdf['Coordinates'] = list(zip(df.Longitude, df.Latitude))
>>> gdf['Coordinates'] = df['Coordinates'].apply(Point)
# inplace=True modifies gdf itself rather than returning a copy
>>> gdf.set_geometry('Coordinates', inplace=True)

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

```
...                         geometry='Coordinates')
```

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

Plotting GeoDataFrames

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

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

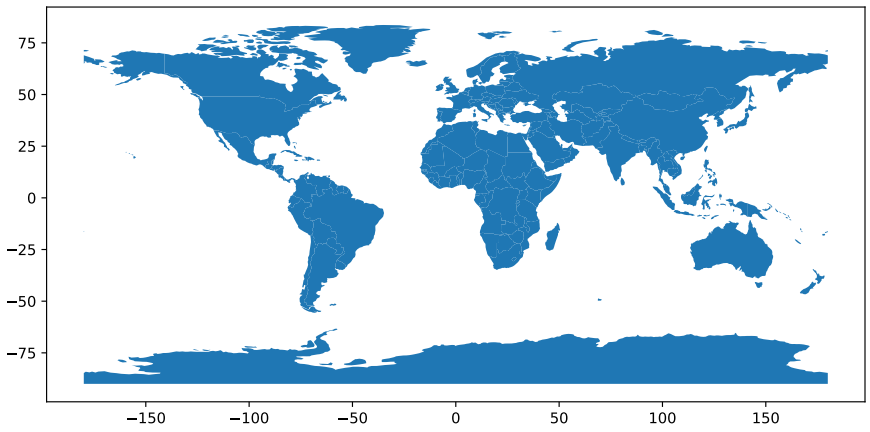


Figure 1.1: World map

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

```
# Set outline of world countries as base
>>> fig,ax = plt.subplots(figsize=(10,7), ncols=1, nrows=1)
>>> base = world.boundary.plot(edgecolor='black', ax=ax, linewidth=1)

# Load airport data and convert to a GeoDataFrame
>>> airports = pd.read_csv('airports.csv')
>>> airports['Coordinates'] = list(zip(airports.Longitude, airports.Latitude))
```

```
>>> airports['Coordinates'] = airports.Coordinates.apply(Point)
>>> airports = gpd.GeoDataFrame(airports, geometry='Coordinates')

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

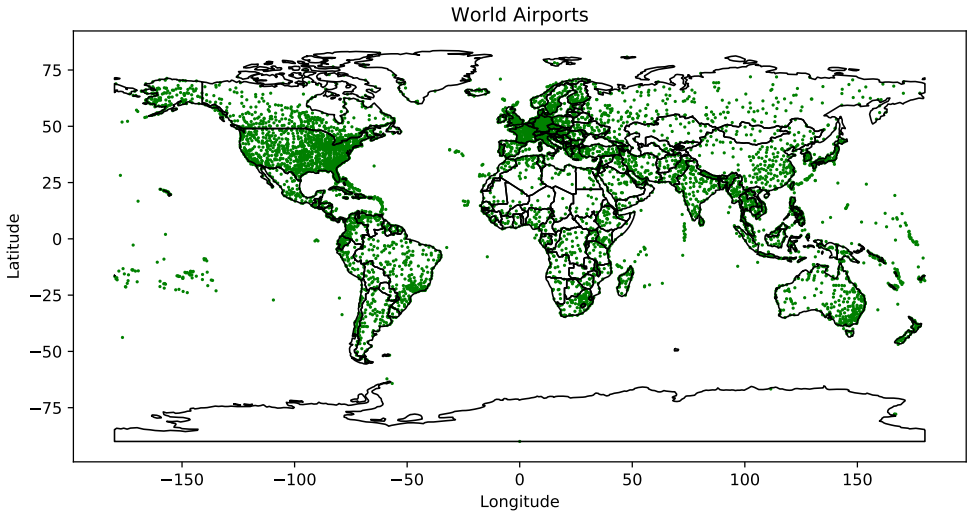


Figure 1.2: Airport map

Problem 1. Read in the file `airports.csv` as a pandas DataFrame. Create three convex hulls around the three sets of airports listed below. This can be done by passing in lists of the airports' coordinates to a `shapely.geometry.Polygon` object.

Create a new GeoDataFrame with these three Polygons as entries. Plot this GeoDataFrame on top of an outlined world map.

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

Working with GeoDataFrames

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

```
# Create column in the world GeoDataFrame for gdp_per_capita
>>> world['gdp_per_cap'] = world.gdp_md_est / world.pop_est
```

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

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

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

GeoSeries objects in a GeoDataFrame can also be dissolved, or merged, together into one GeoSeries based on their geometry data. For example, all countries on one continent could be merged to create a GeoSeries containing the information of that continent. The method designed for this is called `dissolve`. It receives two parameters, `by` and `aggfunc`. `by` indicates which column to dissolve along, and `aggfunc` tells how to combine the information in all other columns. The default `aggfunc` is `first`, which returns the first application entry. In the following example, we use `sum` as the `aggfunc` so that each continent is the combination of its countries.

```
>>> world = world[['continent', 'geometry', 'gdp_per_cap']]

# Dissolve world GeoDataFrame by continent
>>> continent = world.dissolve(by = 'continent', aggfunc='sum')
```

Projections and Coloring

When plotting, GeoPandas uses the CRS (coordinate reference system) of a GeoDataFrame. This reference system indicates how coordinates should be spaced on a plot. Two of the most commonly used CRSs are EPSG:4326 and EPSG:3395. EPSG:4326 is the standard latitude-longitude projection used by GPS. EPSG:3395, also known as the Mercator projection, is the standard navigational projection.

When creating a new GeoDataFrame, it is important to set the `crs` attribute of the GeoDataFrame. This allows any plots to be shown correctly. Furthermore, GeoDataFrames being layered need to have the same CRS. To change the CRS, use the method `to_crs()`.

```
# Check CRS of world GeoDataFrame
>>> print(world.crs)
epsg:4326
```

```
# Change CRS of world to Mercator
# inplace=True ensures that we modify world instead of returning a copy
>>> world.to_crs(3395, inplace=True)
>>> print(world.crs)
epsg:3395
```

GeoPandas accepts many different CRSs; a reference can be found at `www.spatialreference.org`. Additionally, inspecting a given CRS object in the terminal without using `print()` or `str()` can be used to get additional information about a specific CRS:[1]

```
>>> world.crs
<Projected CRS: EPSG:3395>
Name: WGS 84 / World Mercator
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: World between 80°S and 84°N.
- bounds: (-180.0, -80.0, 180.0, 84.0)
Coordinate Operation:
- name: World Mercator
- method: Mercator (variant A)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

GeoDataFrames can also be plotted using the values in the the other attributes of the GeoSeries. The map plots the color of each geometry object according to the value of the column selected. This is done by passing in the parameter `column` into the `plot()` method.

```
>>> fig, ax = plt.subplots(1, figsize=(10,4))
# Plot world based on gdp
>>> world.plot(column='gdp_md_est', cmap='OrRd', legend=True,  ax=ax)
>>> ax.set_title('World Map based on GDP')
>>> ax.set_xlabel('Longitude')
>>> ax.set_ylabel('Latitude')
>>> plt.show()
```

[1] This can also be accomplished using `print(repr(crs))`.

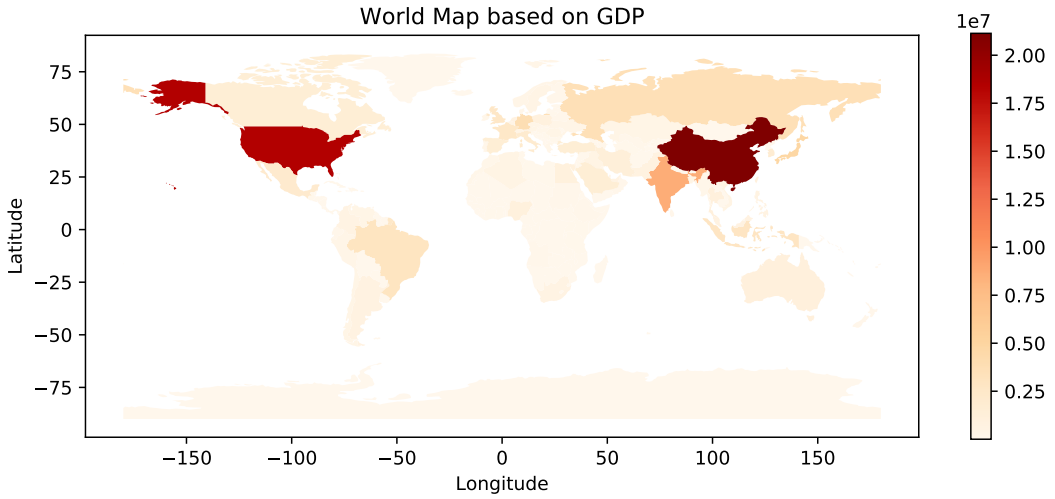


Figure 1.3: World Map Based on GDP

Problem 2. The file `county_data.gpkg.zip` contains information about US counties.[a] After unzipping, use the command `gpd.read_file('county_data.gpkg')` to create a GeoDataFrame of this information. Each county's shape is stored in the `geometry` column. Use this to plot all US counties two times, first using the default CRS and then using EPSG:5071.

Next, create a new GeoDataFrame that merges all counties within a single state. Drop regions with the following STATEFP codes: 02, 15, 60, 66, 69, 72, 78. Plot this GeoDataFrame to see an outline of all 48 contiguous states. Ensure a CRS of EPSG:5071.

[a] Source: `http://www2.census.gov/geo/tiger/GENZ2016/shp/cb_2016_us_county_5m.zip`

Note

`.gpkg` files are actually structured as a directory that contains several files that each contain parts of the data. For instance, `county_data.gpkg` consists of the files `county_data.cpg`, `county_data.dbf`, `county_data.prj`, `county_data.shp`, and `county_data.shx`. Be sure that these files are placed directly in the first level of folders, and not in further subdirectories.
To use this file in Google Colab, upload the zipped file and extract it with the following code:

```
county = files.upload()
!unzip county_data.gpkg.zip
```

It then can be loaded:

```
county_df = gpd.read_file('county_data.gpkg')
```

Merging GeoDataFrames

Just as multiple pandas DataFrames can be merged, multiple GeoDataFrames can be merged with attribute joins or spatial joins. An attribute join is similar to a merge in pandas. It combines two GeoDataFrames on a column (not the geometry column) and then combines the rest of the data into one GeoDataFrame.

```
>>> world = gpd.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
>>> cities = gpd.read_file(geopandas.datasets.get_path('naturalearth_cities'))

# Create subsets of the world and cities GeoDataFrames
>>> world = world[['continent', 'name', 'iso_a3']]
>>> cities = cities[['name', 'iso_a3']]

# Merge the GeoDataFrames on their iso_a3 code
>>> countries = world.merge(cities, on='iso_a3')
```

A spatial join merges two GeoDataFrames based on their geometry data. The function used for this is `sjoin`. `sjoin` accepts two GeoDataFrames and then direction on how to merge. It is imperative that two GeoDataFrames have the same CRS. In the example below, we merge using an `inner` join with the option `intersects`. The `inner` join means that we will only use keys in the intersection of both geometry columns, and we will retain only the left geometry column. `intersects` tells the GeoDataFrames to merge on GeoSeries that intersect each other. Other options include `contains` and `within`.

```
# Combine countries and cities on their geographic location
>>> countries = gpd.sjoin(world, cities, how='inner', op='intersects')
```

Problem 3. Load in the file `nytimes.csv`[a] as a DataFrame. This file includes county-level data for the cumulative cases and deaths of Covid-19 in the US, starting with the first case in Snohomish County, Washington, on January 21, 2020. Begin by converting the `date` column into a `DatetimeIndex`.

Next, use county FIPS codes to merge your GeoDataFrame from Problem 2 with the DataFrame you just created. A FIPS code is a 5-digit unique identifier for geographic locations. Ignore rows in the Covid-19 DataFrame with unknown FIPS codes as well as all data from Hawaii and Alaska.

Note that the `fips` column of the Covid-19 DataFrame stores entries as floats, but the county GeoDataFrame stores FIPS codes as strings, with the first two digits in the `STATEFP` column and the last three in the `COUNTYFP` column.

Once you have completed the merge, plot the cases from March 21, 2020 on top of your state outline map from Problem 2, using the CRS of EPSG:5071. Finally, print out the name of the county with the most cases on March 21, 2020 along with its case count.

[a] Source: `https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv`

Logarithmic Plotting Techniques

The color scheme of a graph can also help to communicate information clearly. A good list of available colormaps can be found at `https://matplotlib.org/3.2.1/gallery/color/colormap_reference.html`. Note also that you can reverse any colormap by adding `_r` to the end. The following example demonstrates some plotting features, using country GDP as in Figure 1.3.

```
>>> fig, ax = plt.subplots(figsize=(15,7), ncols=1, nrows=1)
>>> world.plot(column='gdp_md_est', cmap='plasma_r',
...         ax=ax, legend=True, edgecolor='gray')

# Add title and remove axis tick marks
>>> ax.set_title('GDP on Linear Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

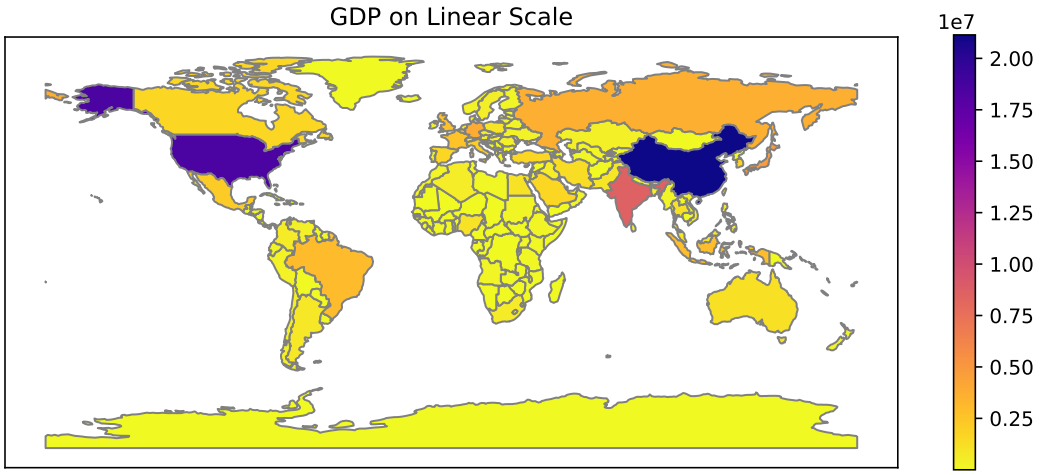


Figure 1.4: World map showing country GDP

Sometimes data can be much more informative when plotted on a logarithmic scale. See how the world map changes when we add a `norm` argument in the code below. Depending on the purpose of the graph, Figure 1.5 may be more informative than Figure 1.4.

```
>>> from matplotlib.colors import LogNorm
>>> from matplotlib.cm import ScalarMappable
>>> fig, ax = plt.subplots(figsize=(15,6), ncols=1, nrows=1)

# Set the norm using data bounds
>>> data = world.gdp_md_est
>>> norm = LogNorm(vmin=min(data), vmax=max(data))

# Plot the graph using the norm
>>> world.plot(column='gdp_md_est', cmap='plasma_r', ax=ax,
```

```
...         edgecolor='gray', norm=norm)

# Create a custom colorbar
>>> cbar = fig.colorbar(ScalarMappable(norm=norm, cmap='plasma_r'),
...         ax=ax, orientation='horizontal', pad=0, label='GDP')

>>> ax.set_title('Country Area on a Log Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

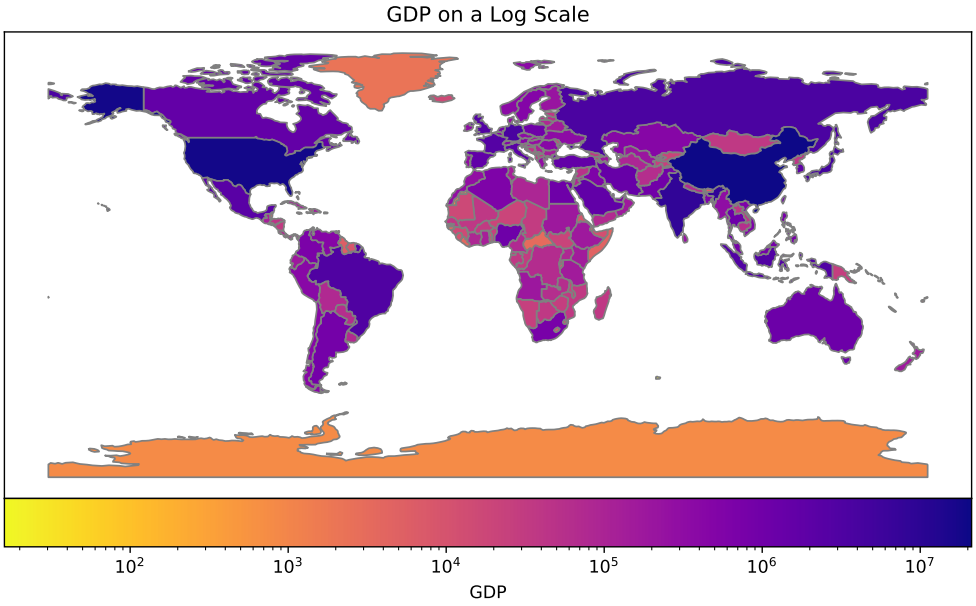


Figure 1.5: World map showing country GDP using a log scale

Problem 4. As in Problem 3, plot your state outline map from Problem 2 on top of a map of the Covid-19 cases from March 21, 2020. This time, however, use a log scale. Use EPSG:5071 for the CRS. Pick a good colormap (the counties with the most cases should generally be darkest) and be sure to display a colorbar.

Problem 5. In this problem, you will create an animation of the spread of Covid-19 through US counties from January 21, 2020 to June 21, 2020. Use a log scale and a good colormap, and be sure that you're using the same norm and colorbar for the whole animation. Use EPSG:5071 for the projection.

As a reminder, below is a summary of what you will need in order to animate this map. You may also find it helpful to refer to the animation section included with the Volume 4 lab

manual.

1. Set up your figure and norm. Be sure to use the highest case count for your `vmax` so that the scale remains uniform.
2. Write your `update` function. This should plot the cases from a given day.
3. Set up your colorbar. Do this outside the `update` function to avoid adding a new colorbar each day.
4. Create the animation and embed it.

1 Random Forests

Lab Objective: *Understand how to build and use a classification tree and a random forest.*

Classification Trees

Decision Classification trees are a class of decision trees used in a wide variety of settings where labeled training data is available. The desired outcome is a model that can accurately assign labels to unlabeled data. Decision trees are widely used because they have a fast run time, low computation cost, and can handle irrelevant, missing, and noisy data easily.

We begin with a data set of samples, such as information about customers from a certain store. Each sample contains a variety of features, such as if the individual is married or has children. The sample also has a classification label, such as whether or not the person made a specific purchase.

A classification tree is composed of many *nodes*, which ask a question (i.e. "Is income >= 85?") and then split the data based on the answers. If the response is `True`, then the sample is "pushed" down the tree to the left child node. If the response is `False`, then the sample is "pushed" down the tree to the right child node. A *leaf* node is a node that has no child node. Upon arrival at a leaf, an unlabeled sample is labeled with the classification that matches the majority of labeled samples at that leaf. Table 1.1 includes information about 10 individuals and then an indicator of whether or not they made a certain purchase. To simplify construction of the tree, all data is numeric, so 1=Yes and 0=No for yes/no questions.

Suppose we wanted to guess whether a single college student making under $30,000 would purchase this item. Starting at the top of the tree, we compare our sample to the question and first choose the right branch, and then we compare with the second question and choose the right branch again. Now we reach a leaf with the dictionary `{0:1}`. The key `0` corresponds to the label, and the value `1` means one of our original samples is at this leaf with that label. Since 100% of samples at this leaf are labeled with `0`, our new sample college student will be predicted to share the label `0`.

If we arrived instead at a leaf with the dictionary `{0:1, 1:4}`, then one of our original samples at this leaf would be labeled `0` and four would be labeled `1`, so the majority vote would assign the label `1` to our new sample.

Married (Y/N)	Children	Income ($1000)	Purchased (Y/N)
0	5	125	0
1	0	100	0
0	0	70	0
1	3	120	0
0	0	95	1
1	0	60	0
0	2	220	1
0	0	85	1
1	0	75	0
0	0	90	1

Table 1.1: Customer data with 3 features (Married, Children, Income) and a label (Purchase) indicating whether or not the customer bought the item.

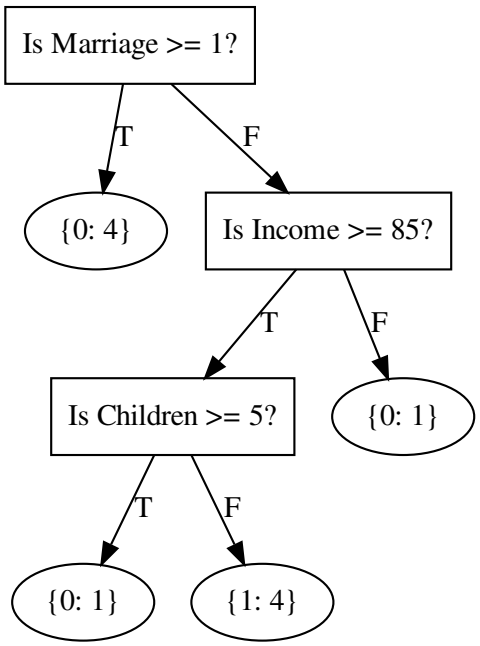


Figure 1.1: A classification tree built using Table 1.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

Problem 1. At each node in a classification tree, a question indicates which branch a sample belongs to. Write a `Match` method for the class `Question` that accepts a sample and returns `True` or `False` depending on how the sample's features compare to the question. For example, in the example above, a single college student making $20,000 would be a sample represented by the array `[0, 0, 20]`.

Next, write a `partition` function that partitions a data set for a given question. Return

the left and right regions of the partition in that order. If one region is empty, return it as `None`.

Measures

To use the `partition` function from Problem 1, we need to know which question to ask at each node. Usually, the question is determined by the split that maximizes either the Gini impurity or the information gain. Gini impurity measures how often a sample would be mislabeled based on the distribution of labels. It is a measure of homogeneity of labels, so it is 0 when all samples at a node have the same label.

Definition 1.1. *Let D be a data set with K different class labels and N different samples. Let N_k be the number of samples labeled class k for each $1 \leq k \leq K$, and let $f_k = \frac{N_k}{N}$. We define the* Gini impurity *to be*

$$G(D) = 1 - \sum_{k=1}^{K} f_k^2.$$

Information gain is based on the concept of Information Theory entropy. It measures the difference between two probability distributions. If the distributions are equal, then the information gain is 0. We will use a modified version of information gain for simplicity.

Definition 1.2. *Let $s_D(p, x) = D_1, D_2$ be a partition of data D. We define the* information gain *of this partition to be*

$$I(s_D(p, x)) = G(D) - \sum_{i=1}^{2} \frac{|D_i|}{|D|} \cdot G(D_i)$$

where $|D|$ represents the number of samples (or rows) in D.

Problem 2. Write a function `gini()` that computes the Gini impurity of an array of data with the class labels in the last column. Write another function `info_gain()` that computes the information gain for a given split of data. Make sure these functions account for the case of the data array containing only a single sample.

The file `animals.csv` contains information about 7 features for 100 animals. The last column, the class labels, indicates whether or not an animal lives in the ocean. You may use this file to test your functions. To test your functions, your values should match those below.

```
>>> import numpy as np
# Load in the data
>>> animals = np.loadtxt('animals.csv', delimiter=',')
# Load in feature names
>>> features = np.loadtxt('animal_features.csv', delimiter=',', dtype=str,
...                       comments=None)
# Load in sample names
>>> names = np.loadtxt('animal_names.csv', delimiter=',', dtype=str)
```

```
# Test your functions
>>> gini(animals)
0.4758
# split animals into two sets with fifty animals in each
>>> info_gain(animals[:50], animals[50:], gini(animals))
0.14579999999999999
```

Optimal Split

The optimal split of a data set can be chosen by maximizes either the Gini impurity or the information gain. We will optimize the information gain, so the optimal split is

$$s_D^* = s_D(p^*, x^*),$$

where

$$p^*, x^* = \text{argmax}_{p,x} I(s_D(p, x)).$$

Sometimes the partition to split on may separate the data into very small subsets with only a few samples each. This can make the classification tree vulnerable to overfitting and noisy data. For this reason, classification trees include an argument to specify the smallest allowable leaf size, or the minimum number of samples at the node. This number depends on the size of the whole data set; for example, data with 10,000 samples would have a larger minimum leaf than our first example using data with only 10 samples.

Problem 3. Write a function `find_best_split()` that computes the optimal split of a data set by checking through all possible `Question`s associated with the data (all values present for each feature (column)). Recall that the final column has the class label and will have no possible questions associated with it. Include a minimum leaf argument defaulting to 5. Do not allow the best split to include a leaf smaller than this size. Return the information gain and question associated with the best split. If two splits have the same information gain, choose the first split.

The output for the animals data set should be `(0.12259833679833688, Is # legs/tentacles >= 2.0?)`.

Building the Tree

Once the optimal split is determined, the node is defined to be a Leaf node or a Decision node. As described earlier, leaf nodes have no children nodes and is where the classification for a sample is made. If the optimal split returns a left and right tree, then the node is a decision node and has a question associated with it to determine which path a sample should follow. The next two problems will walk through building a classification tree using the functions and classes from the previous problems.

Problem 4. Write the class `Leaf`. It should have an attribute `prediction` that is the dictionary of how many samples at the leaf belong to each label, as shown in the leaves of Figure

1.1.

Next, write the class `Decision_Node`. This should have three attributes: an associated `Question`, a left branch, and a right branch. The branches will be `Leaf` or `Decision_Node` objects. Name these three attributes `question`, `left`, and `right`.

In addition to having a minimum leaf size, it's also important to have a maximum depth for trees. Without restricting the depth, the tree can become very large; if there is no minimum leaf size, it can be one less than the number of training samples. Limiting the depth can stop the tree from having too many splits, preventing it from becoming too complex and overfitting the training data. It's also important to not have too shallow of a tree because then the tree will underfit the data.

Problem 5. Write a function `build_trees()` that uses your previous functions to build a classification tree. Include a minimum leaf argument defaulting to 5 and a maximum depth argument defaulting to 4. Start counting depth at 0. For comparison, the tree in Figure 1.1 has depth 3.

You will probably want to build this tree recursively. If the remaining data has too few samples, if the depth is too much, or if the information gain is 0, make a `Leaf`. Otherwise, make a partition and build a new tree for each branch, returning those as `Decision_Nodes`.

The last column in the `animals.csv` file indicates whether or not the animal lives in the ocean; this is the class label for this data set. Test your classifier with this file and the function `draw_tree`. This will display and save a pdf of the graph. Examine the figure and test various parameters to check if your functions are working properly.

```
# How to draw a tree
>>> my_tree = build_tree(animals, features)
>>> draw_tree(my_tree)
```

NOTE

The function `draw_tree` relies on the Graphviz package, which you can download by typing `conda install -c conda-forge python-graphviz` if you have the Anaconda distribution. If `draw_tree` returns an error about pdf being an unrecognized file type, try typing `dot -c` in your terminal.

Predicting

It's important to test your tree to ensure that it predicts class labels fairly accurately and so that you can adjust the minimum leaf and maximum depth parameters as needed. It is customary to randomly assign some of your labeled data to a training set that you use to fit your tree and then use the rest of your data as a testing set to check accuracy.

Problem 6. Write a function `predict_tree` that returns the predicted class label for a new sample given a trained tree. You will probably have to make this recursive in order to traverse the branches and reach a `Leaf` node with prediction information.

Next, write a function `analyze_tree` that accepts a labeled data set (with the labels in the last column, as in `animals.csv`) and a trained classification tree and returns the proportion of samples that the tree labels correctly.

Test your function with the `animals.csv` file. Shuffle the data set with `np.random.shuffle()` and use 80 samples to train your classification tree. Use the other 20 samples as the test set to see how accurately your tree classifies them. Your tree should be able to classify this set with roughly 80% accuracy on average, given the default parameters.

Random Forest

As noted, one of the main issues with Decision Trees is their tendency to overfit Random forests are a way of mitigating overfitting that cannot be fixed by restricting the depth and leaf size. A *random forest* is just what it sounds like–a collection of trees. Each tree is trained randomly, meaning that at each node, only a small, random subset of the features is available by which to determine the next split. The size of this subset should be small relative to the total number of features present. Let n be the total number of features in the data set. One common method, and the one we will use here, is to split on $\sqrt{n}$ features, rounding down where applicable.

When predicting the label of a new sample, each trained tree in the forest casts a vote, determined as above, and the sample is labeled according to the majority vote of the trees.

Problem 7. Add an argument `random_subset` to `build_tree()` and `find_best_split()`, defaulting to `False`, that indicates whether or not the tree should be trained randomly. When `True`, each node should be restricted to a random combination of $\sqrt{n}$ features to use in its split, where n is the total number of features (note that class labels are not features).

Next, write a function `predict_forest()` that accepts a new sample and a trained forest (as a list of trees). It should return the assigned label, found by majority vote of the trees.

Finally, write a function `analyze_forest()` that accepts a labeled data set and a trained forest and analyzes the accuracy of the forest's predictions.

Test your functions out on the `animals.csv` file. Examine the graphs of the individual trees to see how they compare to the non-randomized versions.

Scikit-Learn

Next, we'll compare our implementation to scikit-learn's `RandomForestClassifier`. Rather than accepting all the data as a single array, as in our implementation, this package accepts the feature data as the first argument and all of the labels as the second argument.

```
>>> from sklearn.ensemble import RandomForestClassifier

# Create the forest with the appropriate arguments and 200 trees
>>> forest = RandomForestClassifier(n_estimators=200, max_depth=4,
...                                 min_samples_leaf=5)
```

```
# Shuffle the data
>>> shuffled = np.random.permutation(animals)
>>> train = shuffled[:80]
>>> test = shuffled[80:]

# Fit the model to your data, passing the labels in as the second argument
>>> forest.fit(train[:,:-1], train[:,-1])

# Test the accuracy with the testing set
>>> forest.score(test[:,:-1], test[:,-1])
0.85
```

Problem 8. The file `parkinsons.csv` contains annotated speech data from people with and without Parkinson's Disease. The first column is the subject ID, columns 2-27 are various features, and the last column is the label indicating whether or not the subject has Parkinson's. You will need to remove the first column so your forest doesn't use participant ID to predict class labels. Feature names are contained in the file `parkinsons_features.csv`.

Write a function to compare your forest implementation to the package from scikit-learn. Because of the size of this data set, we will only use a small portion of the samples and build a very simple forest. Randomly select 130 samples. Use 100 in training your forest and 30 more in testing it. Include 5 trees in the forest and use `min_samples_leaf=15`. Time how long it takes to train and analyze your forest.

Repeat this with scikit-learn's package, using the same 100 training samples and 30 test samples. Set `n_estimators=5` and `min_samples_leaf=15`.

Next, using scikit-learn's package, run the whole data set, using the default parameters. Use 80% of the data to train the forest and the other 20% to test it.

Return three tuples, where each tuple contains the accuracy and time for each variation.

1 Data Cleaning

Lab Objective: *The quality of a data analysis or model is limited by the quality of the data used. In this lab we learn techniques for cleaning data, creating features, and determining feature importance.*

Almost every dataset has problems that make it unsuitable for regression or other modeling. At a basic level, these problems might cause simple functions to error out. More substantially, data problems could significantly change the result of your model or analysis.

Data cleaning is the process of identifying and correcting bad data. This could be data that is missing, duplicated, irrelevant, inconsistent, incorrect, in the wrong format, or does not make sense. Though it can be tedious, data cleaning is the most important step of data analysis. Without accurate and legitimate data, any results or conclusions are suspect and may be incorrect.

We will demonstrate common issues with data and how to correct them using the following dataset. It consists of family members and some basic details.

```
# Example dataset
>>> df = pd.read_csv('toy_dataset.csv')

>>>
          Name  Age        name         DOB Marital_Status
0     John Doe   30        john  01/01/2010       Divorcee
1     Jane Doe   29        jane  12/02/1990       Divorced
2   Jill smith   40         NaN  03/04/1980        married
3   Jill smith   40        jill  03/04/1980        married
4   jack smith  100        jack    4/4/1980       marrieed
5  Jenny Smith    5         NaN  05/05/2015            NaN
6  JAmes Smith    2         NaN  20/06/2018         single
7        Rover    2         NaN  05/05/2018            NaN



   Height  Weight    Marriage_Len      Spouse
0    72.0     175               5         NaN
1     5.5     125               5    John Doe
2    64.0     120              10  Jack Smith
```

```
3    64.0     120             NaN  jack smith
4     1.8     220              10  jill smith
5   105.0      40             NaN         NaN
6    27.0      25  Not Applicable         NaN
7    36.0      50             NaN         NaN
```

Inspection

The first step of data cleaning is to analyze the quality of the data. If the quality is poor, the data might not be worth using. Knowing the quality of the data will also give you an idea of how long it will take to clean it. A quality dataset is one in which the data is valid, accurate, complete, consistent, and uniform. Some of these issues, like uniformity, are fairly easy to fix during cleaning, while other aspects like accuracy are more difficult, if not impossible.

Validity is the degree that the data conforms to given rules. If a column corresponds to the temperature in Salt Lake City, measured in degrees Farenheit, then a value over 110 or below 0 should make you suspicious, since those would be extreme values for Salt Lake City. In fact, checking the all-time temperature records for Salt Lake shows that the values in this column should never be more than 107 and never less than -30. Any values outside that range are almost certainly errors and should probably be reset to NaN, unless you have special information that allows you to impute more accurate values.

Some standard rules are

- **data type**: The data types of each column should all be the same.
- **data range**: The data of a column, typically numbers or dates, should all be in the same range.
- **mandatory constraints**: Certain columns cannot have missing data.
- **unique constraint**: Certain columns must be unique.
- **regular expression patterns**: A text column must be in the same format, like phone numbers must in the form 999-999-9999.
- **cross-field validation**: Conditions must hold across multiple columns, a hospital discharge date can't be earlier than the admittance date.
- **duplicated data**: Rows or columns that are repeated. In some cases, they may not be exact.

We can check the data type in Pandas using `dtype`. A `dytpe` of `object` means that the data in that column contains either strings or mixed `dtypes`. These fields should be investigated to determine if they contain mixed datatypes. In our toy example, we would expect that `Marriage_Len` is numerical, so an object `dtype` is suspicious. Looking at the data, we see that `James` has `Not Applicable`, which is a string.

```
# Check validity of data
# Check Data Types
>>> df.dtypes
Name               object
Age                 int64
```

```
name               object
DOB                object
Marital_Status     object
Height            float64
Weight              int64
Marriage_Len       object
Spouse             object
dtype: object
```

Duplicates can be easily identified in Pandas using the `duplicated()` function. When no parameters are passed, it returns a DataFrame of the first duplicates. We can identify rows that are duplicated in only some columns by passing in the column names. The `keep` parameter has three possible values, first, last, and False. False keeps all duplicated values, while first and last only keep one of the duplicated values, the first and last ones respectively.

```
# Display duplicated rows
>>> df[df.duplicated()]
Empty DataFrame
Columns: [Name, Age, name, DOB, Marital_Status, Height, Weight, Marriage_Len, ↩
    Spouse]
Index: []

# Display rows that have duplicates in some columns
>>> df[df.duplicated(['Name','DOB','Marital_Status'],keep=False)]
         Name  Age  name         DOB Marital_Status  Height  Weight ↩
            Marriage_Len      Spouse
2  Jill smith   40   NaN  03/04/1980        married    64.0     120           ↩
    10  Jack Smith
3  Jill smith   40  jill  03/04/1980        married    64.0     120          ↩
    NaN  jack smith
```

We can check the range of values in a numeric column using the `min` and `max` attributes. Other options for looking at the values include line plots, histograms, and boxplots. Some other useful Pandas commands for evaluating data include `pd.unique()` and `df.nunique()`, which identify and count unique values. `value_counts()` counts the number of values in each item of a column, like a histogram.

```
# Count the number of unique values in each row
>>> df.nunique()
Name              7
Age               6
name              2
DOB               7
Marital_Status    5
Height            7
Weight            7
Marriage_Len      4
Spouse            4
dtype: int64
```

```
# Print the unique Marital_Status values
>>> pd.unique(df['Marital_Status'])
array(['Divorcee', 'Divorced', 'married', 'marrieed', nan, 'single'],
      dtype=object)


# Count the number of each Marital_Status values
>>> df['Marital_Status'].value_counts()
married     2
single      1
marrieed    1
Divorcee    1
Divorced    1
Name: Marital_Status, dtype: int64
```

The accuracy of the data, how close the data is to reality, is harder to confirm. Just because a data point is valid, doesn't mean that it is true. For example, a valid street address doesn't have to exist, or a person might lie about their weight. The first case could be checked using mapping software, but the second could be unverifiable.

The percentage of missing data is the completeness of the data. All uncleaned data will have missing values, but datasets with large amounts of missing data, or lots of missing data in key columns, are not going to be as useful. Pandas has several functions to help identify and count missing values. In Pandas, all missing data is considered a `NaN` and does not affect the dtype of a column. `df.isna()` returns a boolean DataFrame indicating whether each value is missing. `df.notnull()` returns a boolean DataFrame with `True` where a value is not missing.

```
# Count number of missing data points in each column
>>> df.isna().sum()
Name              0
Age               0
name              6
DOB               0
Marital_Status    2
Height            0
Weight            0
Marriage_Len      2
Spouse            4
dtype: int64
```

Consistency measures how consistent the data is in the dataset and across multiple datasets. For example, in our toy dataset, `Jack Smith` is 100 years old, but his birth year is 1980. Data is inconsistent across datasets when the data points should be the same and are different. This could be due to incorrect entries or syntax. An example is using multiple finance dataset to build a predictive model. The dates in each dataset should have the same format so that they can all be used equally in the model. Any columns that have monetary data should all be in the same unit, like dollars or pesos.

Lastly, uniformity is the measure of how similarly the data is formatted. Data that has the same units of measure and syntax are considered uniform. Looking at the `Height` column in our dataset, we see values ranging from 1.8 to 105. This is likely the result of different units of measure.

When looking at the quality of the data, there are no set rules on how to measure these concepts or at what point the data is considered bad data. Sometimes, even if the data is bad, it is the only data available and has to be used. Having an idea of the quality of the data will help you know what cleaning steps are needed and help with analyzing the results. Creating a `summary statistics`, also known as `data profiling` is a good way to get a general idea of the quality of the data. The `summary statistics` should be specific to the dataset and describe aspects of the data discussed in this section. It could also include visualizations and basic statistics, like the mean and standard deviation.

Visualization is an important aspect of the inspection phase. Using histograms, box plots, and hexbins can identify outliers in the data. Outliers should be investigated to determine if they are accurate. Removing outliers will improve your model, but you should only remove an outlier if you have a legitimate reason. Columns that have a small distribution or variance, or consist of one value, could be worth removing since they might contribute little to the model.

Problem 1. The `g_t_results.csv` file is a set of parent-reported scores on their child's Gifted and Talented tests. The two tests, `OLSAT` and `NNAT`, are used by NYC to determine if children are qualified for gifted programs. The OLSAT Verbal has 16 questions for Kindergartners and 30 questions for first, second, and third graders. The NNAT has 48 questions. Using this dataset, answer the following questions.

1) What column has the highest number of null values and what percent of its values are null? Print the answer as a tuple with (column name, percentage). Make sure the second value is a percent.

2) List the columns that should be numeric that aren't. Print the answer as a tuple.

3) How many third graders have scores outside the valid range for the OLSAT Verbal Score? Print the answer

4) How many data values are missing (NaN)? Print the number.

Each part is one point.

Cleaning

After the data has been inspected, it's time to start cleaning. There are many aspects and methods of cleaning; not all of them will be used in every dataset. Which ones you choose should be based on your dataset and the goal of the project.

Unwanted Data

Removing unwanted data typically falls into two categories, duplicated data and irrelevant data. Duplicated observations usually occur when data is scraped, combined from multiple datasets, or a user submits the data twice. Irrelevant data consists of observations that don't fit the specific problem you are trying to solve or don't have enough variation to affect the model. We can drop duplicated data using the `duplicated()` function described above with `drop()` or using `drop_duplicates`, which has the same parameters as `duplicated`.

Validity Errors

After moving unwanted data, we correct any validity errors found during inspection. All features should have a consistent type, standard formatting (like capitalization), and the same units. Syntax errors should be fixed, and white space at the beginning and ends of strings should be removed. Some data might need to be padded so that it's all the same length.

Method	Description
series.str.lower()	Convert to all lower case
series.str.upper()	Convert to all upper case
series.str.strip()	Remove all leading and trailing white space
series.str.lstrip()	Remove leading white space
series.str.replace(" ","")	Remove all spaces
series.str.pad()	Pad strings

Table 1.1: Pandas String Formatting Methods

Validity also includes correcting or removing contradicting values. This might be two values in a row or values across datasets. For example, a child shouldn't have a marital status of married. Another example is if two columns should sum to a third but don't for a specific row.

Missing Data

There will always be missing data in any uncleaned dataset. Some commonly suggested methods for handling data are removing the missing data and setting the missing values to some value based on other observations. However, missing data can be informative and removing or replacing missing data erases that information. Also, removing missing values from a dataset might result in significant amounts of data being lost. Removing missing data could also make your model less accurate if you need to predict on data with missing values, so retaining the missing values can help increase accuracy.

So how can we handle missing data? Dropping missing data is the easiest method. Dropping rows should only be done if the are a small number of missing data points in a column or if the row is missing a significant amount of data. If a column is very sparse, consider dropping the entire column. Another option is to estimate the missing data's value and replace it. There are many ways to do this, including mean, mode, median, randomly choosing from the distribution, linear regression, and hot-decking.

Hot-deck is when you fill in the data based on similar observations. It can be applied to numerical and categorical data, unlike most of the other options listed above. The easiest hot-deck method is to fill in the data with random numbers after dividing the data into groups based on some characteristic, like gender. Sequential hot-deck sorts the column with missing data based on an auxiliary column and then fills in the data with the value from the next available data point. K-Nearest Neighbors can also be used to identify similar data points.

The last option is to flag the data as missing. This retains the information from missing data and removes the missing data (by replacing it). For categorical data, simply replace the data with a new category. For numerical data, we can fill the missing data with 0, or some value that makes sense, and add an indicator variable for missing data. This allows the algorithm to estimate the constant for missing data instead of just using the mean.

```
## Replace missing data
import numpy as np
```

```
# Add an indicator column based on missing Marriage_Len
>>> df['missing_ML'] =df['Marriage_Len'].isna()

# Fill in all missing data with 0
>>> df['Marriage_Len'] = df['Marriage_Len'].fillna(0)

# Change all other NaNs to missing
>>> df = df.fillna('missing')

# Change Not Applicable row to NaNs
>>> df = df.replace('Not Applicable',np.nan)

# Drop rows will NaNs
>>> df = df.dropna()


>>> df
          Name  Age         DOB Marital_Status
0     JOHN DOE   30  01/01/2010       divorcee
1     JANE DOE   29  12/02/1990       divorced
2   JILL SMITH   40  03/04/1980        married
3   JACK SMITH   40   4/4/1980         married
4  JENNY SMITH    5  05/05/2015        missing

   Height  Weight Marriage_Len      Spouse  missing_ML
0    72.0     175            5     missing       False
1    68.0     125            5    John Doe       False
2    64.0     120           10  Jack Smith       False
3    71.0     220           10  jill smith       False
4    41.0      40            0     missing        True
```

Nonnumerical Values Misencoded as Numbers

More dangerous, in many ways, than numerical errors, are entries that are recorded as a numerical value (`float` or `int`) when they should be recorded as nonnumerical data, that is, in a format that cannot be summed, multiplied, or averaged. One example is missing data recorded as 0. Missing data should always be stored in a form that cannot accidentally be incorporated into the model. Typically this is done by storing NaN as the value. However, the above method of using `missing` as the value is more valuable since some algorithms will not run on data with NaN. Unfortunately, many datasets have recorded missing values with a 0 or some other number. You should verify that this does not occur in your dataset. Similarly, a survey with a scale from 1 to 5 will sometimes have the additional choice of "N/A" (meaning "not applicable"), which could be coded as 6, not because the value 6 is meaningful, but just because that is the next thing after 5. Again, this should be fixed so that the "N/A" choice cannot accidentally be used for any computations.

Categorical data are also often encoded as numerical values. These values should not be left as numbers that can be computed with. For example, postal codes are shorthand for locations, and there is no numerical meaning to the code. It makes no sense to add, subtract, or multiply

postal codes, so it is important not to let those accidentally be added, subtracted, or multiplied, for example by inadvertently including them in the design matrix (unless they are one-hot encoded or given some other meaningful numerical value). It is good practice to convert postal codes, area codes, ID numbers, and other non-numeric data into strings or other data types that cannot be computed with.).

Ordinal Data

Ordinal data is data that has a meaningful order, but the differences between the values aren't consistent, or maybe aren't even meaningful at all. For example, a survey question might ask about your level of education, with 1 being high-school graduate, 2 bachelor's degree, 3 master's degree, and 4 doctoral degree. These values are called ordinal data because it is meaningful to talk about an answer of 1 being less than an answer of 2. However, the difference between 1 and 2 is not necessarily the same as the difference between 3 and 4, and it would not make sense to compute an average answer—the average of a high school diploma and a masters degree is not a bachelor's degree, despite the fact that the average of 1 and 3 is 2. Treating these like categorical data loses the information of the ordering, but treating it like regular numerical data implies that a difference of 2 has the same meaning whether it comes as $3-1$ or $4-2$. If that last assumption is approximately true, then it may be ok to treat these data as numerical in your model, but if that assumption is not correct, it may be better to treat the variable as categorical.

Problem 2. `imdb.csv` contains a small set of information about 99 movies. Clean the data set by doing the following in order:

1. Remove duplicate rows by dropping the first or last. Print the shape of the dataframe after removing the rows.
2. Drop all rows that contain missing data. Print the shape of the dataframe after removing the rows.
3. Remove rows that have data outside valid data ranges and explain briefly how you determined your ranges for each column.
4. Identify and drop columns with three or fewer different values. Print a tuple with the names of the columns dropped.
5. Convert the titles to all lower case.

Print the first five rows of your dataframe.

Feature Engineering

One often needs to construct new columns, commonly referred to as `features` in the context of machines learning, for a dataset, because the dependent variable is not necessarily a linear function of the features in the original dataset. Constructing new features is called *feature engineering*. Once new features are created, we can analyze how much a model depends on each feature. Features with low importance probably do not contributed much and could potentially be removed.

Fognets are fine mesh nets that collect water that condenses on the netting. These are used in some desert cities in Morocco to produce drinking water. Consider a dataset measuring the amount of

water Y collected from fognets, where one of the features WindDir is the wind direction, measured in degrees. This feature is not likely to contribute meaningfully in a linear model because the direction 359 is almost the same as the direction 0, but no nonzero linear multiple of WindDir will reflect this relation. One way to improve the situation is to replace the WindDir with two new (engineered) features: $\sin\left(\frac{\pi}{180}\text{WindDir}\right)$ and $\cos\left(\frac{\pi}{180}\text{WindDir}\right)$.

Discrete Fourier transforms and wavelet decomposition often reveal important properties of data collected over time (*called time-series*), like sound, video, economic indicators, etc. In many such settings it is useful to engineer new features from a wavelet decomposition, the DFT, or some other function of the data.

Problem 3. `basketball.csv` contains data for all NBA players between 2001 and 2018. Each row represents a player's stats for a year. The features in this data set are

- player (str): the player's name
- age (int): the player's age
- team_id (cat): the player's team
- per (float): player efficiency rating, how much a player produced in one minute of play
- ws (float): win shares, an estimate of how much the player contributed to
- bpm (float): box plus/minus is the estimated number of points a player contributed to over 100 possessions
- year (int): the year

(float):

Create two new features:

- career_length (int): number of years player has been playing (start at 0).
- target (str): The target team if the player is leaving. If the player is retiring, the target should be 'retires'. A player is retiring if their name doesn't exist the next year. (Set the players in 2019 to NaN).

Remove all duplicates of a player in each year. Remove all rows except those where a player changes team, that is, target is not null nor 'retires'. Drop the player, year, and team_id columns.

Return the first ten lines of the dataframe.

Engineering for Categorical Variables

Categorical features are those that take only a finite number of values, and usually no categorical value has a numerical meaning, even if it happens to be number. For example in an election dataset, the names of the candidates in the race are categorical, and there is no numerical meaning (neither ordering nor size) to numbers assigned to candidates based soley on their names.

Consider the following election data.

Ballot number	For Governor	For President
001	Herbert	Romney
002	Cooke	Romney
003	Cooke	Obama
004	Herbert	Romney
005	Herbert	Romney
006	Cooke	Stein

A common mistake occurs when someone assigns a number to each categorical entry (say 1 for Cooke, 2 for Herbert, 3 for Romney, etc.). While this assignment is not, in itself, inherently incorrect, it is incorrect to use the value of this number in a statistical model. Any such model would be fundamentally wrong because a vote for Cooke cannot, in any reasonable way, be considered half of a vote for Herbert or a third of a vote for Romney. Many researchers have accidentally used categorical data in this way (and some have been very publicly embarrassed) because their categorical data was encoded numerically, which made it hard to recognize as categorical data.

Whenever you encounter categorical data that is encoded numerically like this, immediately change it either to non-numerical form ("Cooke," "Herbert," "Romney,"...) or apply a one-hot encoding as described below.

In order to construct a meaningful model with categorical data, one normally applies a *one-hot encoding* or *dummy variable encoding*. [1] To do this construct a new feature for every possible value of the categorical variable, and assign the value 1 to that feature if the variable takes that value and zero otherwise. Pandas makes one-hot encoding simple:

```
# one-hot encoding
df = pd.get_dummies(df, columns=['For President']])
```

The previous dataset, when the presidential race is one-hot encoded, becomes

Ballot number	Governor	Romney	Obama	Stein
001	Herbert	1	0	0
002	Cooke	1	0	0
003	Cooke	0	1	0
004	Herbert	1	0	0
005	Herbert	1	0	0
006	Cooke	0	0	1

Note that the sum of the terms of the one-hot encoding in each row is 1, corresponding to the fact that every ballot had exactly one presidential candidate.

When the gubernatorial race is also one-hot encoded, this becomes

Ballot number	Cooke	Herbert	Romney	Obama	Stein
001	0	1	1	0	0
002	1	0	1	0	0
003	1	0	0	1	0
004	0	1	1	0	0
005	0	1	1	0	0
006	1	0	0	0	1

[1] Yes, these are silly names, but they are the most common names for it. Unfortunately, it is probably too late to change these now.

Now the sum of the terms of the one-hot encodings in each row is 2, corresponding to the fact that every ballot had two names—one gubernatorial candidate and one presidential candidate.

Summing the columns of the one-hot-encoded data gives the total number of votes for the candidate of that column. So the numerical values in the one-hot encodings are actually numerically meaningful, and summing the entries gives meaningful information. One-hot encoding also avoids the pitfalls of incorrectly using numerical proxies for categorical data.

The main disadvantage of one-hot encoding is that it is an inefficient representation of the data. If there are C categories and n datapoints, a one-hot encoding takes an $n \times 1$-dimensional feature and turns it into an $n \times C$ sparse matrix. But there are ways to store these data efficiently and still maintain the benefits of the one-hot encoding.

Achtung!

When performing linear regression, it is good practice to add a constant column to your dataset and to remove one column of the one-hot encoding of each categorical variable. (Adding a constant column should only be done in linear regression).

To see why, notice that summing terms in one row corresponding to the one-hot encoding of a specific categorical variable (for example the presidential candidate) always gives 1. If the dataset already has a constant column (which you really always should add if it isn't there already), then the constant column is a linear combination of the one-hot encoded columns. This cause the matrix to fail to be invertible and can cause identifiability problems.

The standard way to deal with this is to remove one column of the one-hot embedding for each categorical variable. For example, with the elections dataset above, we could remove the Cooke and Romney columns. Doing that means that in the new dataset a row sum of 0 corresponds to a ballot with a vote for Cooke and a vote for Romney, while a 1 in any column indicates how the ballot differed from the base choice of Cooke and Romney.

When using pandas, you can drop the first column of a one-hot encoding by passing in `drop_first=True`.

Problem 4. Load `housing.csv` into a dataframe with index=0. Descriptions of the features are in `housing_data_description.txt`. The goal is to construct a regression model that predicts SalePrice using the other features of the dataset. Do this as follows:

1. Identify and handle the missing data. Hint: Dropping every row with some missing data is not a good choice because it gives you an empty dataframe. What can you do instead?
2. Identify the variable with nonnumerical values that are misencoded as numbers. One-hot encode it. Hint: don't forget to remove one of the encoded columns to prevent collinearity with the constant column).
3. Add a constant column to the dataframe.
4. Save a copy of the dataframe.
5. Choose four categorical features that seem very important in predicting SalePrice. One-hot encode these features, and remove all other categorical features.

6. Run an OLS regression on your model.

   Print the ten features that have the highest coef in your model and the summary.

   To run an OLS model in python, use the following code.

```
import statsmodels.api as sm

>>> results = sm.OLS(y, X).fit()

# Print the summary
>>> results.summary()

# Convert the summary table to a dataframe
>>> results_as_html = a.tables[1].as_html()
>>> result_df = pd.read_html(results_as_html, header=0, index_col=0)[0]
```

Problem 5. Using the copy of the dataframe you created in Problem 4, one-hot encode all the categorical variables. Print the shape of you database, and Run OLS.

Print the ten features that have the highest `coef` in your model and the summary. Write a couple of sentences discussing which model is better and why.

1 Finding Patterns in Data: LSI and more about Scikit-Learn

Lab Objective: *Understand the basics of principal component analysis and latent semantic indexing. Learn more about scikit-learn and implement a machine learning pipeline.*

Principal Component Analysis

Principal Component Analysis (PCA) is a multivariate statistical tool used to change the basis of a set of samples from the basis of original features (which may be correlated) into a basis of uncorrelated variables called the *principal components*. It is a direct application of the singular value decomposition (SVD). The first principal component will account for the greatest variance in the samples, the second principal component will be orthogonal to the first and account for the second greatest variance, etc. By projecting the samples onto the space spanned by the first few principal components, we can reduce the dimensionality of the data while preserving most of the variance.

Take a matrix X with samples as rows and features as columns. The first step in PCA is to pre-process the data, which usually includes translating the columns of X to have mean 0. Some datasets require additional scaling based on variance and units of measurement. Call the new pre-processed matrix Y.

We next compute the truncated SVD of our centered data, $Y = U\Sigma V^T$, where the columns of V are the principal components and form an orthonormal basis for the space spanned by the samples. The variance captured by each principal component can be calculated by the equation below, where σ_i is the i-th nonzero singular value and there are k total singular values.

$$\frac{\sigma_i^2}{\sum_{j=1}^{k} \sigma_j^2} \tag{1.1}$$

In general, we are only interested in the first several principal components. But just how many principal components should we keep? One method is to keep the first two principal components so that we can project the data into 2-dimensional space. Another is to only keep the set of principal components accounting for a certain percentage of the variance, using the equation above.

Once we have decided how many principal components to keep (say the first l), we can project the samples from the original feature space onto the principal component space by computing

$$\widehat{Y} = U_{:,:l}\Sigma_{:l,:l} = YV_{:,:l}$$

Problem 1. The breast cancer dataset from scikit-learn has 569 samples with 30 features each. Each sample is labeled as `0` (malignant) or `1` (benign). With 30 features, this data can't be directly visualized, so we will use PCA to graph the first two principal components, which account for nearly all of the variance in the data.

You can load this data using the following code.

```
>>> cancer = sklearn.datasets.load_breast_cancer()
>>> X = cancer.data
>>> y = cancer.target # Class labels (0 or 1)
```

Write a function that performs PCA on the breast cancer dataset. Graph the first two principal components, with the first along the x-axis. Your graph should resemble Figure 1.1 below. Include in the graph title the amount of variance captured by the first two principal components, calculated with Equation 1.1.

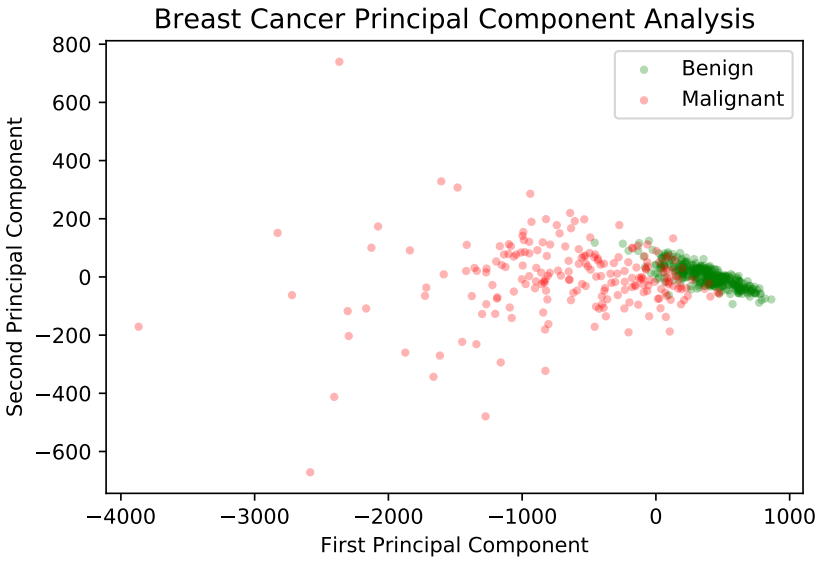


Figure 1.1: First two principal components of the transformed breast cancer data

Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an application of PCA to the realm of natural language processing. In particular, LSI employs the SVD to reduce the dimensionality of a large corpus of text documents in order to enable us to evaluate the similarity between two documents. Many information-retrieval systems used in government and in industry are based on LSI.

To motivate the problem, suppose we have a large collection of documents about various topics. How can we find an article about BYU? We might consider simply choosing the article that contains the acronym the greatest number of times, but this is a crude method. A better way is to use a form of PCA on the collection of documents.

In order to do so, we need to represent the documents as numerical vectors. A standard way of doing this is to define an ordered set of words occurring in the collection of documents

(called the *vocabulary*) and then to represent each document as a vector of word counts from the vocabulary. More formally, let our vocabulary be $V = \{w_1, w_2, \ldots, w_m\}$. Then a document is a vector $x = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$ such that x_i is the number of occurrences of word w_i in the document. In this setup, we represent the entire collection of m documents as an $n \times m$ matrix X, where m is the number of vocabulary words and n is the number of documents in our collection, so each row is a document vector. As expected, we let $X_{i,j}$ be the number of times term j occurs in document i. Note that X is often a sparse matrix, as any single document likely does not contain most of the vocabulary words. This mode of representation is called the *bag of words* model for documents.

We calculate the SVD of X *without* centering or scaling the data so that we may retain the sparsity. This is unique to this particular problem. We now have $X = U\Sigma V^T$. If we are keeping l principal components, we can represent the corpus of documents by the matrix

$$\widehat{X} = U_{:,:l}\Sigma_{:l,:l} = XV_{:,:l}$$

Note that $\widehat{X}$ will no longer be a sparse matrix, but will have dimension $n \times l$.

Now that we have our documents represented in terms of the first l principal components, we can find the similarity between two documents. Our measure for similarity is simply the cosine of the angle between the vectors; a small angle (large cosine) indicates greater similarity, while a large angle (small cosine) indicates greater dissimilarity. Recall that we can use the inner product to find the cosine of the angle between two vectors. Under this metric, the similarity between document i and document j (represented by the i-th and j-th row of $\widehat{X}$, 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

$$\text{argmax}_{j \neq i} \frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

Problem 2. Create a function `similar` that takes in a sparse matrix `Xhat` and an index `i` and returns the indices of the most similar and the least similar documents.

Application: State of the Union

We now discuss some practical issues involved in creating the bag of words representation X from the raw text. Our dataset will consist of the US State of the Union addresses from 1945 through 2013, each contained in a separate text file in the folder `Addresses`. We would like to avoid loading in all of the text into memory at once, and so we will *stream* the documents one at a time.

The first thing we need to establish is the vocabulary set, i.e. the set of unique words that occur throughout the collection of documents. A Python set object automatically preserves the uniqueness of the elements, so we will create a set and then iteratively read through the documents, adding the unique words of each document to the set. As we read in each document, we will remove punctuation and numerical characters and convert everything to lower case. The following code, found in the function `document_converter()`, will accomplish this task.

```
# Get list of file paths to each text file in the folder
```

```
>>> folder = "./Addresses/"
>>> paths = [folder+p for p in os.listdir(folder) if p.endswith(".txt")]

# Helper function to get list of words in a string
>>> def extractWords(text):
...     ignore = string.punctuation + string.digits
...     cleaned = "".join([t for t in text.strip() if t not in ignore])
...     return cleaned.lower().split()

# Initialize vocab set, then read each file and add to the vocab set.
>>> vocab = set()
>>> for p in paths:
...     with open(p, 'r') as infile:
...         for line in infile:
...             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 with key-value pairs of the form (word, index).

```
# Load stopwords.
>>> with open("stopwords.txt", 'r') as f:
...     stops = set([w.strip().lower() for w in f.readlines()])

# Remove stopwords from vocabulary, create ordering.
>>> vocab = {w:i for i, w in enumerate(vocab.difference(stops))}
```

We are now ready to create the word count vectors for each document, and we store these in a sparse matrix X. It is convenient to use the `Counter` object from the `collections` module, as this object automatically counts the occurrences of each distinct element in a list.

```
>>> from collections import Counter
>>> counts = [] # holds the entries of X
>>> doc_index = [] # holds the row index of X
>>> word_index = [] # holds the column index of X

# Iterate through the documents.
>>> for doc, p in enumerate(paths):
...     with open(p, 'r') as f:
...         # create the word counter
...         ctr = Counter()
...         for line in f:
...             ctr.update(extractWords(line))
...         # Iterate through the word counter, storing counts
...         for word, count in ctr.items():
...             if word in vocab:
...                 word_index.append(vocab[word])
```

```
...                 counts.append(count)
...                 doc_index.append(doc)

# Create sparse matrix holding these word counts.
>>> X = sparse.csr_matrix((counts, [doc_index, word_index]),
...                           shape=(len(paths), len(vocab)), dtype=np.float)
```

Problem 3. Applying the techniques of LSI discussed above to the word count matrix X, and keeping the first 7 principal components, write a function that takes in the path to a single State of the Union address `speech` and returns a tuple of the addresses that are most and least similar to `speech`. For Ronald Reagan's 1984 speech, the input would be './Addresses/1984-Reagan.txt', and your output should be ('1988-Reagan', '1946-Truman'). Be sure to format the strings properly.

Since X is a sparse matrix, you will need to use the SVD method found in `scipy.sparse.linalg`. This method operates slightly differently than the SVD method found in `scipy.linalg`, so be sure to read the documentation.

The simple bag of words representation is a bit crude, as it fails to consider how some words may be more important than others in determining the similarity of documents. Words appearing in few documents tend to provide more information than words occurring in every document. For example, while the word *war* might not be considered a stop word, it is likely to appear in many more addresses than the word *Afghanistan*. Two speeches sharing the word *Afghanistan* are probably more closely related than two speeches sharing the word *war*. So while $X_{i,j}$ is a good measure of the importance of term j in document i, we also need to consider some kind of global weight for each term j, indicating how important the term is over the entire collection. There are a number of different weights we could choose; we will employ the following approach. Define

$$p_{i,j} = \frac{X_{i,j}}{\sum_j X_{i,j}}.$$

We then let

$$g_j = 1 + \sum_{i=1}^{m} \frac{p_{i,j} \log(p_{i,j} + 1)}{\log m},$$

where m is the number of documents in the collection. We call g_j the *global weight* of term j. We replace each term frequency in the matrix X by weighting it globally. Specifically, we define a matrix A with entries

$$A_{i,j} = g_j \log(X_{i,j} + 1).$$

We can now perform LSI on the matrix A, whose entries are both locally and globally weighted.

Problem 4. Use the equation above to edit the function `weighted_document_converter()` to calculate the sparse matrix A. Similar to the function `document_converter()`, this function should return A and a list of file paths.

Scikit-Learn

Scikit-learn is one of the fundamental tools Python offers for machine learning. It includes classifiers, such as `RandomForestClassifier` and `KNeighborsClassifier`, as well as transformers, which preprocess data before classification. In the remainder of this lab, we will discuss transformers, validation tools, how to find optimal hyperparameters, and how to build a machine learning pipeline.

Transformers

A scikit-learn *transformer* processes data to make it better suited for classification. This may involve shifting or scaling data, dropping columns, replacing missing values, and so on. The function from Problem 4 is an example of a transformer, as is PCA.

> **Note**
>
> A *hyperparameter* is not dependent on data. Hyperparameters are declared in the constructor `__init__()`, before data is even passed in. Parameters set during the `fit()` method are often called *model parameters* and do depend on specific data. For example, a `StandardScaler` transformer shifts and scales data to have a mean of 0 and a standard deviation of 1.
>
> Scikit-learn's transformers have three main methods: `fit_transform()`, which fits model parameters and also transforms given data; `fit()`, which sets model parameters but does not perform a transformation; and `transform()`, which transforms data according to pre-fitted model parameters. Model parameters are fitted according to training data, and they are not refitted to testing data, so a `StandardScaler` will shift and scale testing data according to the mean and variance of the training data; the transformed test data likely will not have mean 0 and variance 1.

Scikit-learn has a built-in PCA package. Its hyperparameters include the desired number of principal components and the type of SVD solver to use. Its `fit_transform()` method takes in an array of data and returns the decomposition with `n_components`.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=5) # Create the PCA transformer with hyperparameters
>>> Xhat = pca.fit_transform(X) # Fit the transformer and transform the data
```

Problem 5. Repeat Problem 3 using your weighted document converter function and scikit-learn's built-in PCA decomposition. Do your answers seem more reasonable than before? For Bill Clinton's 1993 speech, your code should return ('./Addresses/1994-Clinton.txt', './Addresses/1951-Truman.txt').

Hint: Scikit-learn's PCA does not accept sparse matrices.

Validation Tools

We now turn our attention from transformers to classifiers. A *classifier* is trained to predict how a new sample should be classified or labeled. Knowing how to determine whether or not a classifier

performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the possible labels are only `0` or `1`.

The `score()` method of a scikit-learn classifier returns the *accuracy* of the model, or the percent of labels predicted correctly. However, accuracy isn't always the best measure of success. Consider the *confusion matrix* for a classifier, the matrix where the (i, j)th entry is the number of samples with actual label i but that are classified with label j. Call the class with label 0 the *negatives* and the class with label 1 the *positives*. Then the confusion matrix is as follows.

$$\begin{array}{c} \\ \text{Actual: } 0 \\ \text{Actual: } 1 \end{array} \begin{array}{c} \text{Predicted: } 0 \qquad\qquad \text{Predicted: } 1 \\ \begin{bmatrix} \text{True Negatives } (TN) & \text{False Positives } (FP) \\ \text{False Negatives } (FN) & \text{True Positives } (TP) \end{bmatrix} \end{array}$$

With this terminology, we define the following metrics.

- *Accuracy*: $\dfrac{TN+TP}{TN+FN+FP+TP}$, the percent of labels predicted correctly.
- *Precision*: $\dfrac{TP}{TP+FP}$, the percent of predicted positives that are actually correct.
- *Recall*: $\dfrac{TP}{TP+FN}$, the percent of actual positives that are predicted correctly.

Precision is useful in situations where false positives are dangerous or costly, while recall is important when avoiding false negatives takes priority. For example, an email spam filter should avoid filtering out an email that isn't actually spam; here a false positive is more dangerous, so precision is a valuable metric for the filter. On the other hand, recall is more important in disease detection: it is better to test positive and not have the disease than to test negative when the disease is actually present. Focusing on a single metric often leads to skewed results, so the following metric is also common.

$$F_\beta \text{ Score} : (1+\beta^2)\frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + FP + \beta^2 FN}.$$

Choosing $\beta < 1$ weighs precision more than recall, while $\beta > 1$ prioritizes recall over precision. The choice of $\beta = 1$ yields the common F_1 score, which weighs precision and recall equally. This is an important alternative to accuracy when, for example, the training set is heavily unbalanced with respect to the class labels.

Scikit-learn implements all of these metrics in `sklearn.metrics`. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. We will be using the function `classification_report()`, which returns precision, recall, and F_1 scores for each label. Each row in the report corresponds to a specific label and gives the scores with its label as the "positive" classification. For example, in binary classification, the row corresponding to `1` gives the scores as they would normally be calculated, with `1` as "positive."

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.metrics import confusion_matrix, classification_report
```

```
>>> from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)

# Fit the esimator to training data and predict the test labels.
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> knn.fit(X_train, y_train)
>>> knn_predicted = knn.predict(X_test)

# Compute the confusion matrix by comparing actual labels to predicted labels.
>>> CM = confusion_matrix(y_test, knn_predicted)
>>> CM
array([[44,  5],
       [10, 84]])

# Get precision, recall, and F1 scores all at once.
# The row labeled 1 gives these scores as we normally calculate them.
>>> print(classification_report(y_test, knn_predicted))
              precision    recall  f1-score   support

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

    accuracy                           0.90       143
   macro avg       0.88      0.90      0.89       143
weighted avg       0.90      0.90      0.90       143
```

Problem 6. For this problem, use the cancer dataset from Problem 1 to compare a `RandomForestClassifier` and a `KNeighborsClassifier`, using the default parameters for each.

Use `train_test_split()` with `random_state=2` to split up the data. Fit the classifiers with the training set and predict the labels for the testing set. Print out a classification report for each classifier, making sure to clearly label which report corresponds to which classifier.

Write a few sentences explaining which of these classifiers would be better to use in this situation and why, using the information from the report as evidence. Remember that in this dataset, the label `1` means benign and `0` means malignant.

Grid Search

Finding the optimal hyperparameters for a given model is a challenging and active area of research.[1] However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with a classifier, a dictionary of hyperparameters, and some validation parameters. When its `fit()` method is called, it tests the given classifier with every possible hyperparameter combination.

[1] Intelligent hyperparameter selection is sometimes called *metalearning*.

For example, a `KNeighborsClassifier` has a few important hyperparameters that can have a significant impact on the speed and accuracy of the model. These include `n_neighbors`, the number of nearest neighbors allowed to vote, and `weights`, which specifies a strategy for weighting the distances between points. The code box below tests various combinations of these hyperparameters.

The cost of a grid search rapidly increases as the hyperparameter space grows. However, the outcomes of each trial are completely independent of each other, so the problem of training each classifier is *embarassingly parallel*, meaning the trials can easily be computed simultaneously. To parallelize the grid search over n CPU cores, set the `n_jobs` parameter to n, or set it to -1 to divide the labor between as many cores as are available.

```
>>> from sklearn.model_selection import GridSearchCV

>>> knn = KNeighborsClassifier()
# Specify values for certain hyperparameters
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...               "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, scoring="f1", n_jobs=-1)

# Run the actual search. This may take some time.
>>> knn_gs.fit(X_train, y_train)

# After fitting, you can access data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_, sep='\n')
{'n_neighbors': 5, 'weights': 'uniform'}
0.9532526583188765

# Access the model
>>> knn_gs.best_estimator_
KNeighborsClassifier(weights='distance')
```

In some circumstances, the parameter grid can be organized in a way that eliminates redundancy. For example, with a `RandomForestClassifier`, you could test each `max_depth` argument with entirely different sets of values for `min_samples_leaf`. To specify certain combinations of parameters, enter the parameter grid as a list of dictionaries.

Problem 7. Do a grid search on the breast cancer dataset using a `RandomForestClassifier`. Modify at least three parameters in your grid. Use `scoring="f1"` for the `GridSearchCV` object. Fit your model with the same train-test split as in Problem 6. Print out the best parameters and the best score.

Next, use the `GridSearchCV` object to predict labels for your test set. Print out a confusion matrix using these values.

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* chains together one or more transformers and one estimator (such as a classifier) into a single object, complete with `fit()` and `predict()` methods.

This simplifies and automates the machine learning process so that when you get new data or make changes to various functions and features, you can easily rerun the new version from beginning to end.

The following example demonstrates how to use a pipeline with a `StandardScaler` transformer and a `KNeighborsClassifier`. Like classifiers, pipelines have `fit()`, `predict()`, and `score()` methods. Each member of the pipeline is declared as a tuple where the first element is a string naming the step and the second is the actual transformer or classifier.

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.pipeline import Pipeline

# Chain together a StandardScaler transformer and a KNN classifier.
>>> pipe = Pipeline([("scaler", StandardScaler()), # "scaler" is the step name
...                  ("knn", KNeighborsClassifier())]) # "knn" is the step name
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972
```

Since `Pipeline` objects follow `fit()` and `predict()` conventions, they can be used with tools like `GridSearchCV`. To specify which hyperparameters belong to which steps of the pipeline, precede each hyperparameter name with `<stepname>__`. For example, `knn__n_neighbors` corresponds to the `n_neighbors` hyperparameter of the pipeline step labeled `knn`.

```
# Create the Pipeline, labeling each step.
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("knn", KNeighborsClassifier())])

# Specify the hyperparameters to test for each step.
>>> pipe_param_grid = {"scaler__with_mean": [True, False],
...                    "scaler__with_std": [True, False],
...                    "knn__n_neighbors": [2,3,4,5,6],
...                    "knn__weights": ["uniform", "distance"]}

# Pass the Pipeline object to the GridSearchCV and fit it to the data.
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        n_jobs=-1).fit(X_train, y_train)

>>> print(pipe_gs.best_params_, pipe_gs.best_score_, sep='\n')
{'knn__n_neighbors': 6, 'knn__weights': 'distance',
 'scaler__with_mean': True, 'scaler__with_std': True}
0.971830985915493
```

Pipelines can also be used to compare different transformations or estimators. For example, a pipeline can end in either a `KNeighborsClassifier()` or a classifier called `SVC()`, even though they have different hyperparameters. Like before, you can use a list of dictionaries to specify the specific combinations of the hyperparameter space.

```
# Create the pipeline, using any classifier as a placeholder
>>> pipe = Pipeline([("scaler", StandardScaler()),
```

```
                    ("classifier", KNeighborsClassifier())])

# Create the grid
>>> pipe_param_grid = [
...     {"classifier": [KNeighborsClassifier()],    # Try a KNN classifier...
...      "classifier__n_neighbors": [2,3,4,5],
...      "classifier__weights": ["uniform", "distance"]},
...     {"classifier": [SVC(kernel="rbf")],         # ...and an SVM classifier.
...      "classifier__C": [.001, .01, .1, 1, 10, 100],
...      "classifier__gamma": [.001, .01, .1, 1, 10, 100]}]

# Fit using training data
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        scoring="f1", n_jobs=-1).fit(X_train, y_train)

# Get the best hyperparameters
>>> params = pipe_gs.best_params_
>>> print("Best classifier:", params["classifier"])
Best classifier: SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

# Check the best classifier against the test data
>>> confusion_matrix(y_test, pipe_gs.predict(X_test))
array([[48,  1],                        # Near perfect!
       [ 1, 93]])
```

Problem 8. The breast cancer dataset has 30 features. By using PCA, we can drastically reduce the dimensionality while still retaining predictive power.

Create a pipeline with a `StandardScaler`, `PCA`, and a `KNeighborsClassifier`. Use the same train-test split as before. Do a grid search on this pipeline, modifying at least six hyperparameters and using `scoring="f1"`. Use no more than 5 principal components. Print out your best parameters and best score. Attain a score of at least .96.

Hint: The documentation for `StandardScaler`, `PCA`, and `KNeighborsClassifier` can be found at these links.

1 Data Augmentation

Lab Objective: *Explore different methods of extending data sets to create more robust classifiers.*

Data Augmentation

It is not hard to find amusing examples of deep neural networks or other machine learning systems that are brittle and respond poorly to inputs that are only slightly different than the data that the systems were trained on. One way to address brittleness in machine learning systems is to train them on a much wider range of examples. Adult humans, for example, have seen many images of stop signs in a wide variety of settings. If a machine learning system had seen as many different images of stop signs in as many different settings, it would be much more robust. But all this requires large amounts of data, and good labeled data is hard to come by.

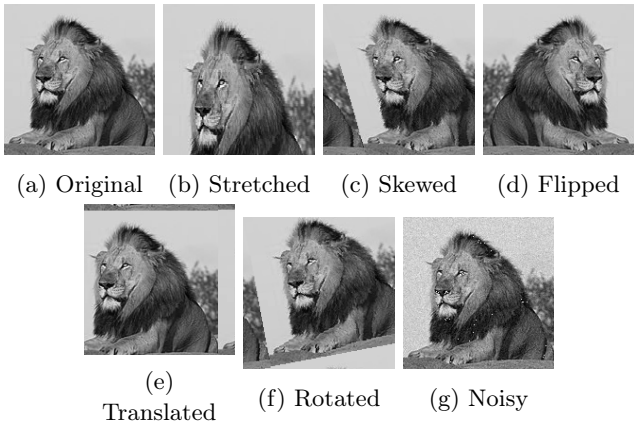
One common approach for generating new data is *data augmentation*, that is, generating new data from old data by applying various transformations that we know should not change the label. For example, an image of a stop sign can be slightly translated, rotated, skewed, or cropped and still be a legitimate image of a stop sign. It may even be blurred or partially obscured, so a classifier for identifying a stop sign must be robust in the face of this sort of interference. Images that don't contain text can often be flipped horizontally, and depending on the image, can also sometimes be flipped vertically. We can use these techniques to generate a larger data set and create more robust classifiers.

As shown in Figure 1 below, these transformations still accurately depict a lion while providing slight modifications to the original image. Image transformations are comprised of three steps. First, create a $2 \times (d1 * d2)$ coordinate representation of the image ($d1$ and $d2$ are the dimensions of the image). For example, if the image is 3 pixels by 3 pixels, the coordinate representation would be

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$

where each column represents the coordinate point of a pixel. Next, perform a linear transformation on the coordinate matrix. Note that some transformations will return float values; however, they need to be integers because the matrix represents coordinate points so you will need to round the values. Finally, use the transformed coordinates to create a new image. (The function `np.take()` may be useful for this.)

Visualizing the code below would give you the second image in Figure 1.



(a) Original (b) Stretched (c) Skewed (d) Flipped

(e) Translated (f) Rotated (g) Noisy

Figure 1.1: Different image transformations

```
>>> import numpy as np
>>> from imageio import imread
>>> import matplotlib.pyplot as plt

>>> lion = imread('sq_lion.png')
>>> d1, d2 = lion.shape
>>> # Get the coordinate points for each pixel in the image
>>> coords = np.mgrid[0:d1, 0:d2].reshape((2,d1*d2))
>>> # Create a linear transformation matrix. (This one will stretch the matrix)
>>> stretch_matrix = np.array([[1, 0], [0, .7]])
>>> # apply the linear transformation to the coordinate matrix
>>> new_coords = stretch_matrix@coords
>>> # some transformations will return entries as floats, but we need them to ↩
    be integers because they are coordinates
>>> new_coords = new_coords.astype(int)
>>> # the next two steps apply the transformation to the image
>>> x, y = new_coords.reshape((2, d1, d2), order='F')
>>> stretched_lion = np.take(lion, x+d1*y, mode='wrap').reshape((d1, d2))
```

When augmenting a data set, it is also important to consider what types of augmentation would provide useful samples. For example, if training a dataset to recognize lowercase letters, flipping an image upside down may not be a useful transformation (consider the letters b and p). It is important to ensure that the transformed data is still an reasonable example of the object it is classified as.

Problem 1. Code from scratch the following simple black-and-white image augmenters that take as inputs the data X (a $d_1 \times d_2$ array that contains an images) and parameters controlling the transformation. It should return the transformed data $f(X)$. Note that each image should receive its own random treatment; for example, if the images are being translated, then each image should be translated by a different (randomly drawn) amount. Your functions should have the the following names and perform the corresponding transform:

1. `translate(X,A,B)`, with parameters A, B. Returns an image translated by a random amount (a, b), where $a \sim unif(-A, A)$, and $b \sim unif(-B, B)$. The resulting image should be cropped to be of size $d_1 \times d_2$. Note that this translation will leave a border on two sides of the image. Fill the empty border with the parts that were cropped off the opposite sides.

2. `rotate(X,T)`, with parameter Θ. Returns each image rotated by a random amount $\theta \sim unif(-\Theta, \Theta)$. The resulting image should be cropped to be the same size as the original, and any blank parts should be filled with one of the parts cropped off the other side. HINT: The rotation matrix is:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

3. `skew(X,A)`, with parameter A. Returns each image with the linear transformation $\left[\begin{smallmatrix} 1 & a \\ 0 & 1 \end{smallmatrix}\right]$ applied, where $a \sim unif(0, A)$. Crop parts that go outside the image boundaries and fill missing areas with the appropriate cropped piece.

4. `flip_horizontal(X)`. Returns a horizontally-flipped version of each image.

5. `gauss_noise(X,s)`, with parameter σ^2. For each image draw $d_1 \times d_2$ random noise values from Normal$(0, \sigma^2)$ and add those to the original image.

Show that each transformation works by displaying a transformed version of lion.png.

Problem 2. Create a function called `image_augment` that will augment your data set using each of the transformations created in problem 1. This function should accept the parameters X (the images), Y (labels), and a list of the parameters for each transformation. The function should return an augmented data set with 6 times the number of images N and an array containing the appropriate label for each image.

Take the sklearn digits dataset, make an 80-20 train-test split, and then apply each of your transformations to the entire training set using `image_augment`. You must decide good values of each of the parameters to use. This should give you a larger (augmented) training set with roughly 8,600 training points. Fit a random forest to the augmented training set and to the original training set and score it using the test set. Return the average score for both classifiers (be sure to label the scores).

Your augmented score should be better than your original score.

Audio Augmentation

For audio data, adding various forms of noise is still a reasonable augmentation choice, but many of the other augmentation methods used for images aren't really suitable. Some useful methods include dropping data at certain time steps, blocking certain frequencies, and changing the pitch or speed.

When adding noise, it may be useful to consider the types of noise most likely to be encountered when the method is in use. For example, if the method will be used to identify voice commands in an outdoor environment, then adding in typical outdoor noises would probably be more useful than

adding white noise, which may not occur much in everyday life. Be thoughtful in choosing how to augment your data, as some types of manipulations may change or obscure the data to the point where it is no longer recognizable.

Audio Packages

There are many different python packages that provide different analysis and audio manipulation tools. Some common packages include `pyAudioAnalysis`, `PYO`, and `ffmpeg-python`. For the purposes of this lab, we will be using `LibROSA`. `LibROSA` is a python package that allows users to read in, write, analyze, and alter .wav files. It can be used to augment an audio dataset and extract features that can be used to distinguish between different sounds or types of music. (`LibROSA` is not included in Anaconda, so you may need to install it with `pip install librosa`.)

```
>>> import matplotlib.pyplot as plt
>>> import librosa
>>> import librosa.display  #this needs to be imported separately,
                            #but is included in the librosa package
>>> import numpy as np
# load the audio time series and sampling rate
>>> chopin, sample_rate = librosa.load('chopin.wav', sr = 22050)
>>> plt.figure(figsize = (15,5))
>>> librosa.display.waveplot(chopin,sample_rate)            #generate plot
>>> plt.show()
```

The `LibROSA` library heavily relies on NumPy arrays. If you already have a NumPy array and it's sample rate, you can skip the `librosa.load()`.

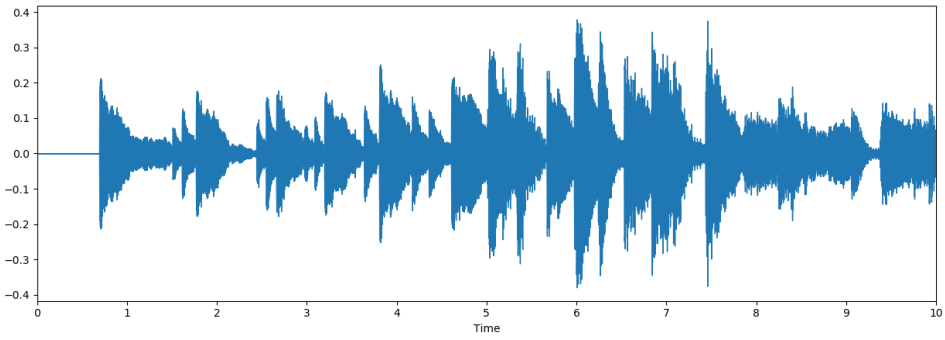


Figure 1.2: Visualization of an audio file

When you load in a `.wav` file, `LibROSA` returns the audio as an `ndarray` and a sample rate.

Depending on the type of audio you are analyzing, different functions may provide better distinguishing characteristics than others. It is important to take these characteristics into account when augmenting data. One possible feature that could be used for classifying music is the "predominant local pulse estimation" or PLP, as shown below. (PLP essentially takes the pulse of the music, just like you can take your own pulse in your wrist.)

```
>>> pulse = librosa.beat.plp(chopin)
>>> plt.figure(figsize = (15,5))
>>> plt.plot(np.linspace(0,10,len(pulse)),pulse)
>>> plt.show()
```

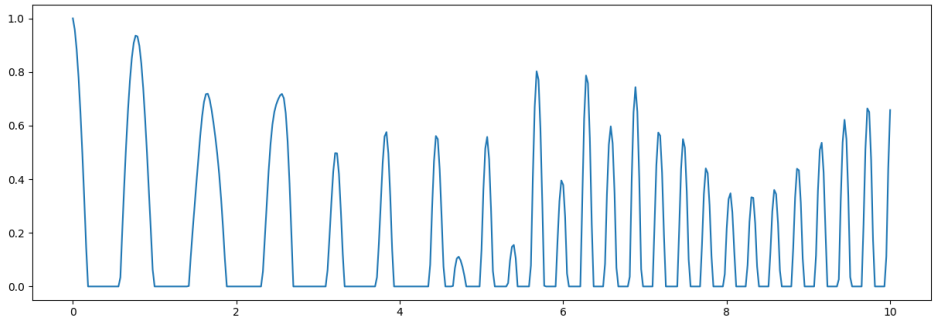


Figure 1.3: Predominant local pulse estimation of audio from figure 1.2

The `LibROSA` package contains several different functions that can be used to manipulate audio data, several of which are described in the table below.

Function	Returns
`time_stretch()`	slows or speeds up audio series by a fixed rate
`pitch_shift()`	shifts the pitch by n_steps semitones
`harmonic()`	extracts the harmonic elements from an audio time-series
`percusive()`	extracts percussive elements from an audio time-series
`split()`	splits an interval into non-silent intervals
`remix()`	re-orders time intervals

Table 1.1: These descriptions were taken directly from `librosa.org/librosa/effects.html`

Problem 3. The file `music.npy` contains the audio time series data of 10 second clips from 150 different songs, with `styles.npy` describing the associated style of ballroom dance. The styles included are Chacha, Foxtrot, Jive, Samba, Rumba, and Waltz. Use `train_test_split` from `sklearn.model_selection` with `test_size=.5` to create train and test sets.

Create two training sets by augmenting this original training set. Each new augmented training set will include the original data and the augmented data. For the first, add ambient noise from the file `restaurant-ambience.wav`. For the second, use `time_stretch`.

HINT: Since the ambient noise clip is much longer than the other music clips, you will have to select a sample of the ambient noise to add to the other clips. It may also benefit you to randomize which ambient noise sample you add to each clip, you can do this by choosing a random index to start from, and sampling starting at that index.

Problem 4. Do the following steps 5 times:

- Use the original data set and the augmented data sets to fit three RandomForestClassifiers, one only on the original data, one on the original data and the data with ambient noise added, and one on the original data and the time stretched data.
- Score each classifier.

Print the mean score for each of the classifiers and print the standard deviation for the scores.

HINT: Use the PLP as a feature you use to fit and classify. This example may be helpful for printing your results nicely.

```
print('\t\t Mean \t STD')
print('Original','\t',np.round(orig.mean(),3),'\t',np.round(orig.std(),3))
print('Ambient Noise','\t',np.round(amb.mean(),3),'\t',np.round(amb.std()↩
    ,3))
print('Time Stretch:','\t',np.round(time.mean(),3),'\t', np.round(time.std↩
    (),3))
```

Synthetic Minority Oversampling

Another situation where generating data can be helpful is in a classification problem where one class is rare compared to the others. For example the problem of identifying glioblastoma (a rare malignant brain tumor) is difficult in part because this cancer only occurs in 3 out of every 100,000 people. A classifier that predicts "no cancer" in every case performs very well (99.997% accurate).

If the training set has 100,000 total cases, only three of which are positive, then undersampling (taking the same number of negatives as positives) gives a dataset with only six total instances, and this is not enough to make a good classifier. Naïve oversampling (repeatedly drawing, with replacement, from the three positive cases to get the same number of positives as negatives) works better than undersampling the negatives, but does not perform very well, because the oversampled dataset just has (roughly) 33,332 repeated instances of each of the three positve instances, and the resulting classifier is likely to be overfit on those three instances.

In the special case where the features are all continuous, we can partially address this class-imbalance problem by synthetically generating new positive instances from the minority class samples (in this case the three positive cases). The *synthetic minority oversampling technique (SMOTE)*[1] works by randomly choosing points along the line segments connecting this point to each (or some) of its k nearest minority neighbors.

SMOTE tends to work better on low-dimensional data than on high-dimensional data. For example if the minority class training examples are images (one dimension per pixel, so, high dimensional) of the subject (say possible tumor cells) that are not centered and not uniformized to be of similar size, then many points along the line connecting two of these images could look nothing like the two endpoints. In such cases SMOTE is not usually very helpful.

[1] See `https://jair.org/index.php/jair/article/view/10302`

The Algorithm

The purpose of this section is to provide a high-level understanding of how Synthetic Minority Oversampling works and will heavily reference the SMOTE paper mentioned earlier.

The goal in creating synthetic observations is to increase the accuracy of the classifier. This means that the synthetic data generated needs to be similar to the minority class data, while creating moderate variation. To do this, select a member of the minority class and find its k nearest neighbors. Randomly select one. Now, for each feature select a random point on the line between the points.

We will demonstrate this using data with two features, represented by x and y coordinates. Consider the points $(0, 0)$, $(1, 3)$, $(2, 1)$, and $(3, 2)$, as shown in figure 1.4.

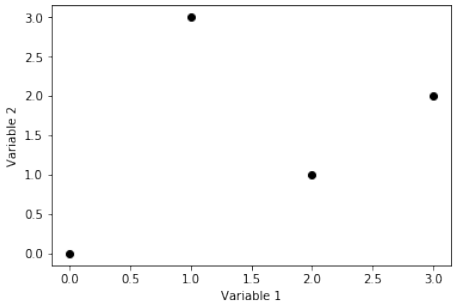


Figure 1.4: Data before SMOTE

To keep things simple, we will use $k = 1$. The nearest neighbor for $(0, 0)$ is $(2, 1)$. Choose a random point between the x values (shown in red) and a random point between the y values (shown in blue). For data with n features, a random point between the two feature values would be chosen for *each* feature. The intersection of these lines gives us the coordinate for the synthetic point (purple).

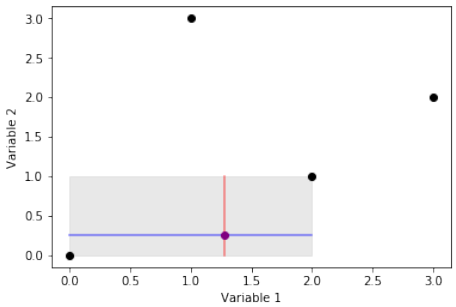
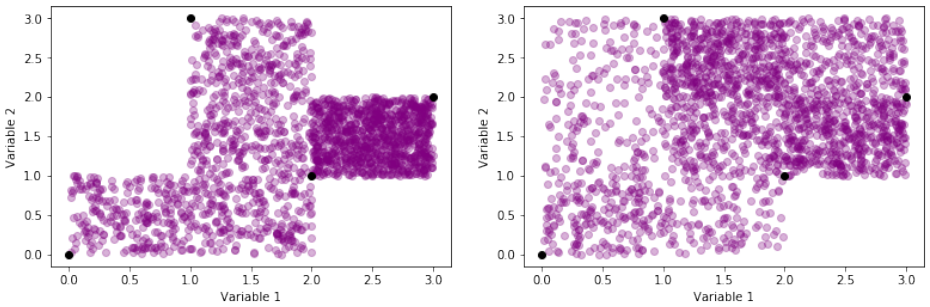


Figure 1.5: SMOTE process

Running this algorithm 500 times per original point, with $k = 1$, returns a graph like figure 1.6a. Running the algorithm 500 times per original point and increasing k to 2, returns the graph like figure 1.6b.



(a) $k = 1$ (b) $k = 2$

Figure 1.6: After SMOTE

Problem 5. Write a function that uses the synthetic minority oversampling technique to augment an imbalanced data set. Your function should:

- Take arguments: `X` a matrix of minority class samples, `N` the number of samples to generate per original point, and `k` the number of nearest neighbors.
- For each original point in the sample, randomly pick one of the `k` nearest neighbors and randomly generate a new point that lies between the two original values. You may use `sklearn.neighbors.KDTree` to find the k nearest neighbors.
- Return an array containing the synthetic samples.

Problem 6. The dataset found in `creditcard.npy` contains information about credit card purchases made over a two day period. Of the approximately 285,000 observations, 492 are fraudulent purchases. The last column indicates if the purchase was valid (0) or fraudulent (1).

Do the following steps 10 times:

- Create a training and test set from the data using `train_test_split` from `sklearn.model_selection` with `test_size=.7`.
- Use `smote` with $N = 500$ and $k = 2$ to augment the training set.
- Create two Gaussian Naïve Bayes classifiers (from `sklearn.naive_bayes.GaussianNB`), one which wil be trained on only the original data and the other on the SMOTE augmented data and the original data.
- Fit each classifier and find the recall and accuracy of each model.

Print the mean recall and mean accuracy of each model and and describe the findings. HINT: Recall $= \frac{tp}{tp+fn}$. This example may be helpful for printing your results nicely.

```
>>> print('\t\t Recall \t Accuracy')
```

```
>>> print('Original','\t',np.round(mean_orig_recall,5),'\t', np.round(↩
    mean_orig_score,5))
>>> print('SMOTE', '\t\t', np.round(mean_smote_recall,5), '\t', np.round(↩
    mean_smote_score,5))
```

1 Introduction to Parallel Computing

Lab Objective: *Many modern problems involve so many computations that running them on a single processor is impractical or even impossible. There has been a consistent push in the past few decades to solve such problems with parallel computing, meaning computations are distributed to multiple processors. In this lab, we explore the basic principles of parallel computing by introducing the cluster setup, standard parallel commands, and code designs that fully utilize available resources.*

Parallel Architectures

Imagine that you are in charge of constructing a very large building. You could, in theory, do all of the work yourself, but that would take so long that it simply would be impractical. Instead, you hire workers, who collectively can work on many parts of the building at once. Managing who does what task takes some effort, but the overall effect is that the building will be constructed many times faster than if only one person was working on it. This is the essential idea behind parallel computing.

A *serial* program is executed one line at a time in a single process. This is analogous to a single person creating a building. Since modern computers have multiple processor cores, serial programs only use a fraction of the computer's available resources. This is beneficial for smooth multitasking on a personal computer because multiple programs can run at once without interrupting each other.

For smaller computations, running serially is fine. However, some tasks are large enough that running serially could take days, months, or in some cases years. In these cases it is beneficial to devote all of a computer's resources (or the resources of many computers) to a single program by running it in *parallel*. Each processor can run part of the program on some of the inputs, and the results can be combined together afterwards. In theory, using N processors at once can allow the computation to run N times faster. Even though communication and coordination overhead prevents the improvement from being quite that good, the difference is still substantial.

A *computer cluster* or *supercomputer* is essentially a group of regular computers that share their processors and memory. There are several common architectures that are used for parallel computing, and each architecture has a different protocol for sharing memory, processors, and tasks between *computing nodes*, the different simultaneous processing areas. Each architecture offers unique advantages and disadvantages, but the general commands used with each are very similar.

In this lab, we will explore the usage and capabilities of parallel computing using Python's iPyParallel package. iPyParallel can be installed with either pip or conda:

```
$ pip install ipyparallel
```

```
$ conda install ipyparallel
```

The iPyParallel Architecture

There are three main parts of the iPyParallel architecture:

- *Client*: The main program that is being run.
- *Controller*: Receives directions from the client and distributes instructions and data to the computing nodes. Consists of a *hub* to manage communications and *schedulers* to assign processes to the engines.
- *Engines*: The individual processors. Each engine is like a separate Python terminal, each with its own namespace and computing resources.

Essentially, a Python program using iPyParallel creates a `Client` object connected to the cluster that allows it to send tasks to the cluster and retreive their results. The engines run the tasks, and the controller manages which engines run which tasks.

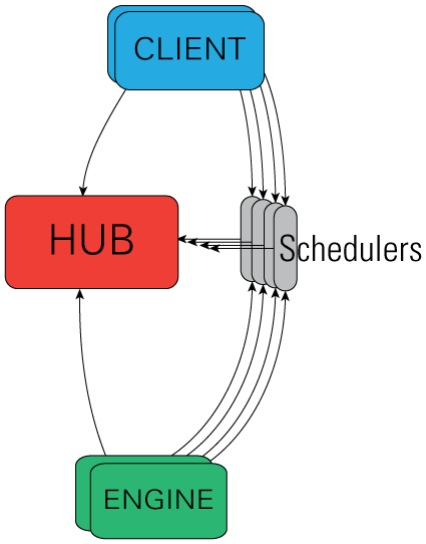


Figure 1.1: An outline of the iPyParallel architecture.

Setting up an iPyParallel Cluster

Before being able to use iPyParallel in a script or interpreter, it is necessarty to start an iPyParallel cluster. We demonstrate here how to use a single machine with multiple processor cores as a cluster. Establishing a cluster on multiple machines requires additional setup, which is detailed in the

Additional Material section. The following commands initialize parts or all of a cluster when run in a terminal window:

Command	Description
`ipcontroller start`	Initialize a controller process.
`ipengine start`	Initialize an engine process.
`ipcluster start`	Initialize a controller process and several engines simultaneously.

Each of these processes can be stopped with a keyboard interrupt (`Ctrl+C`). By default, the controller uses JSON files in `UserDirectory/.ipython/profile-default/security/` to determine its settings. Once a controller is running, it acts like a server, listening connections from clients and engines. Engines will connect automatically to the controller when they start running. There is no limit to the number of engines that can be started in their own terminal windows and connected to the controller, but it is recommended to only use as many engines as there are cores to maximize efficiency.

Achtung!

The directory that the controller and engines are started from matters. To facilitate connections, navigate to the same folder as your source code before using `ipcontroller`, `ipengine`, or `ipcluster`. Otherwise, the engines may not connect to the controller or may not be able to find auxiliary code as directed by the client.

Starting a controller and engines in individual terminal windows with `ipcontroller` and `ipengine` is a little inconvenient, but having separate terminal windows for the engines allows the user to see individual errors in detail. It is also actually more convenient when starting a cluster of multiple computers. For now, we use `ipcluster` to get the entire cluster started quickly.

```
$ ipcluster start           # Assign an engine to each processor core.
$ ipcluster start --n 4     # Or, start a cluster with 4 engines.
```

Note

Jupyter notebooks also have a **Clusters** tab in which clusters can be initialized using an interactive GUI. To enable the tab, run the following command. This operation may require root permissions.

```
$ ipcluster nbextension enable
```

The iPyParallel Interface

Once a controller and its engines have been started and are connected, a cluster has successfully been established. The controller will then be able to distribute messages to each of the engines, which will

compute with their own processor and memory space and return their results to the controller. The client uses the `ipyparallel` module to send instructions to the controller via a `Client` object.

```
>>> from ipyparallel import Client

>>> client = Client()       # Only works if a cluster is running.
>>> client.ids
[0, 1, 2, 3]                # Indicates that there are four engines running.
```

Once the client object has been created, it can be used to create one of two classes: a `DirectView` or a `LoadBalancedView`. These views allow for messages to be sent to collections of engines simultaneously. A `DirectView` allows for total control of task distribution while a `LoadBalancedView` automatically tries to spread out the tasks equally on all engines. The remainder of the lab will be focused on the `DirectView` class.

```
>>> dview = client[:]   # Group all engines into a DirectView.
>>> dview2 = client[:2] # Group engines 0,1, and 2 into a DirectView.
>>> dview2.targets      # See which engines are connected.
[0, 1, 2]
```

Since each engine has its own namespace, modules must be imported in every engine. There is more than one way to do this, but the easiest way is to use the `DirectView` object's `execute()` method, which accepts a string of code and executes it in each engine.

```
# Import NumPy in each engine.
>>> dview.execute("import numpy as np")
```

Problem 1. Write a function that initializes a `Client` object, creates a `DirectView` with all available engines, and imports `scipy.sparse` as `sparse` on all engines. Return the `DirectView`.

Managing Engine Namespaces

Before continuing, set the `DirectView` you are using to use blocking:

```
>>> dview.block = True
```

This affects the way that functions called using the `DirectView` return their values. Using blocking makes the process simpler, so we will use it initially. What blocking is will be explained later.

Push and Pull

The `push()` and `pull()` methods of a `DirectView` object manage variable values in the engines. Use `push()` to set variable values and `pull()` to get variables. Each method also has a shortcut via indexing.

```
# Initialize the variables 'a' and 'b' on each engine.
>>> dview.push({'a':10, 'b':5})         # OR dview['a'] = 10; dview['b'] = 5
[None, None, None, None]                # Output from each engine

# Check the value of 'a' on each engine.
>>> dview.pull('a')                     # OR dview['a']
[10, 10, 10, 10]

# Put a new variable 'c' only on engines 0 and 2.
>>> dview.push({'c':12}, targets=[0, 2])
[None, None]
```

Problem 2. Write a function `variables(dx)` that accepts a dictionary of variables. Create a `Client` object and a `DirectView` and distribute the variables. Pull the variables back and make sure they haven't changed.

Scatter and Gather

Parallelization almost always involves splitting up collections and sending different pieces to each engine for processing. The process is called *scattering* and is usually used for dividing up arrays or lists. The inverse process of pasting a collection back together is called *gathering* and is usually used on the results of processing. This method of distributing a dataset and collecting the results is common for processing large data sets using parallelization.

```
>>> import numpy as np

# Send parts of an array of 8 elements to each of the 4 engines.
>>> x = np.arange(1, 9)
>>> dview.scatter("nums", x)
>>> dview["nums"]
[array([1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]

# Scatter the array to only the first two engines.
>>> dview.scatter("nums_big", x, targets=[0,1])
>>> dview.pull("nums_big", targets=[0,1])
[array([1, 2, 3, 4]), array([5, 6, 7, 8])]

# Gather the array again.
>>> dview.gather("nums")
array([1, 2, 3, 4, 5, 6, 7, 8])

>>> dview.gather("nums_big", targets=[0,1])
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Executing Code on Engines

Execute

The `execute()` method is the simplest way to run commands on parallel engines. It accepts a string of code (with exact syntax) to be executed. Though simple, this method works well for small tasks.

```
# 'nums' is the scattered version of np.arange(1, 9).
>>> dview.execute("c = np.sum(nums)")   # Sum each scattered component.
<AsyncResult: execute:finished>
>>> dview['c']
[3, 7, 11, 15]
```

Apply

The `apply()` method accepts a function and arguments to plug into it, and distributes them to the engines. Unlike `execute()`, `apply()` returns the output from the engines directly.

```
>>> dview.apply(lambda x: x**2, 3)
[9, 9, 9, 9]
>>> dview.apply(lambda x,y: 2*x + 3*y, 5, 2)
[16, 16, 16, 16]
```

Note that the engines can access their local variables in either of the execution methods.

Map

The built-in `map()` function applies a function to each element of an iterable. The iPyParallel equivalent, the `map()` method of the `DirectView` class, combines `apply()` with `scatter()` and `gather()`. Simply put, it accepts a dataset, splits it between the engines, executes a function on the given elements, returns the results, and combines them into one object.

```
>>> num_list = [1, 2, 3, 4, 5, 6, 7, 8]
>>> def triple(x):                  # Map a function with a single input.
...     return 3*x
...
>>> dview.map(triple, num_list)
[3, 6, 9, 12, 15, 18, 21, 24]

>>> def add_three(x, y, z):         # Map a function with multiple inputs.
...     return x+y+z
...
>>> x_list = [1, 2, 3, 4]
>>> y_list = [2, 3, 4, 5]
>>> z_list = [3, 4, 5, 6]
>>> dview.map(add_three, x_list, y_list, z_list)
[6, 9, 12, 15]
```

Blocking vs. Non-Blocking

Parallel commands can be implemented two ways. The difference is subtle but extremely important.

- *Blocking*: The main program sends tasks to the controller, and then waits for all of the engines to finish their tasks before continuing (the controller "blocks" the program's execution). This mode is usually best for problems in which each node is performing the same task.
- *Non-Blocking*: The main program sends tasks to the controller, and then continues without waiting for responses. Instead of the results, functions return an `AsyncResult` object that can be used to check the execution status and eventually retrieve the actual result.

Whether a function uses blocking is determined by default by the `block` attribute of the `DirectView` The execution methods `execute()`, `apply()`, and `map()`, as well as `push()`, `pull()` , `scatter()`, and `gather()`, each have a keyword argument `block` that can instead be used to specify whether or not to using blocking. Alternatively, the methods `apply_sync()` and `map_sync()` always use blocking, and `apply_async()` and `map_async()` always use non-blocking.

```
>>> f = lambda n: np.sum(np.random.random(n))

# Evaluate f(n) for n=0,1,...,999 with blocking.
>>> %time block_results = [dview.apply_sync(f, n) for n in range(1000)]
CPU times: user 9.64 s, sys: 879 ms, total: 10.5 s
Wall time: 13.9 s

# Evaluate f(n) for n=0,1,...,999 with non-blocking.
>>> %time responses = [dview.apply_async(f, n) for n in range(1000)]
CPU times: user 4.19 s, sys: 294 ms, total: 4.48 s
Wall time: 7.08 s

# The non-blocking method is faster, but we still need to get its results.
# Both methods produced a list, although the contents are different
>>> block_results[10]  # This list holds actual result values from each engine.
[3.833061790352166,
4.8943956129713335,
4.268791758626886,
4.73533677711277]

>>> responses[10]           # This list holds AsyncResult objects.
<AsyncResult: <lambda>:finished>
# We can get the actual results by using the get() method of each AsyncResult
>>> %time nonblock_results = [r.get() for r in responses]
CPU times: user 3.52 ms, sys: 11 mms, total: 3.53 ms
Wall time: 3.54 ms          # Getting the responses takes little time.

>>> nonblock_results[10]    # This list also holds actual result values
[5.652608204341693,
4.984164642641558,
4.686288406810953,
5.275735658763963]
```

When non-blocking is used, commands can be continuously sent to engines before they have finished their previous task. This allows them to begin their next task without waiting to send their calculated answer and receive a new command. However, this requires a design that incorporates checkpoints to retrieve answers and enough memory to store response objects.

Class Method	Description
`wait(timeout)`	Wait until the result is available or until `timeout` seconds pass.
`ready()`	Return whether the call has completed.
`successful()`	Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.
`get(timeout)`	Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `TimeoutError` is raised.

Table 1.1: All information from `https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult`.

Table 1.1 details the methods of the `AsyncResult` object.

There are additional magic methods supplied by `iPyParallel` that make some of these operations easier. These methods are explained in the Additional Material section. More information on `iPyParallel` architecture, interface, and methods can also be found at `https://ipyparallel.readthedocs.io/en/latest/index.html`.

Problem 3. Write a function that accepts an integer n. Instruct each engine to make n draws from the standard normal distribution, then hand back the mean, minimum, and maximum draws to the client. Return the results in three lists.

If you have four engines running, your results should resemble the following:

```
>>> means, mins, maxs = problem3(1000000)
>>> means
[0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
>>> mins
[-4.1508589, -4.3848019, -4.1313324, -4.2826519]
>>> maxs
[4.0388107, 4.3664958, 4.2060184, 4.3391623]
```

Problem 4. Use your function from Problem 3 to compare serial and parallel execution times. For $n = 1000000, 5000000, 10000000, 15000000$,

1. Time how long it takes to run your function.
2. Time how long it takes to do the same process serially. Make n draws and then calculate and record the statistics, but use a `for` loop with N iterations, where N is the number of engines running.

Plot the execution times against n. You should notice an increase in efficiency in the parallel version as the problem size increases.

Applications

Parallel computing, when used correctly, is one of the best ways to speed up the run time of an algorithm. As a result, it is very commonly used today and has many applications, such as the following:

- Graphic rendering
- Facial recognition with large databases
- Numerical integration
- Calculating discrete Fourier transforms
- Simulation of various natural processes (weather, genetics, etc.)
- Natural language processing

In fact, there are many problems that are only feasible to solve through parallel computing because solving them serially would take too long. With some of these problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack simple encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in such a way that running them on multiple processors results in a significant run time improvement. Manipulating an algorithm to be run with parallel computing is called *parallelizing* the algorithm. When a problem only requires very minor manipulations to parallelize, it is often called *embarrassingly parallel*. Typically, an algorithm is embarrassingly parallel when there is little to no dependency between results. Algorithms that do not meet this criteria can still be parallelized, but there is not always a significant enough improvement in run time to make it worthwhile. For example, calculating the Fibonacci sequence using the usual formula, $\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$, is poorly suited to parallel computing because each element of the sequence is dependent on the previous two elements.

Problem 5. The *trapeziod rule* is a simple technique for numerical integration:

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{k=1}^{N}(f(x_k) + f(x_{k+1})),$$

where $a = x_1 < x_2 < \ldots < x_N = b$ and $h = x_{n+1} - x_n$ for each n. See Figure 1.2.

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered embarrassingly parallel.

Write a function that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Parallelize the trapezoid rule in order to estimate the integral of f. That is, evenly divide the points among all available processors and run the trapezoid 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.

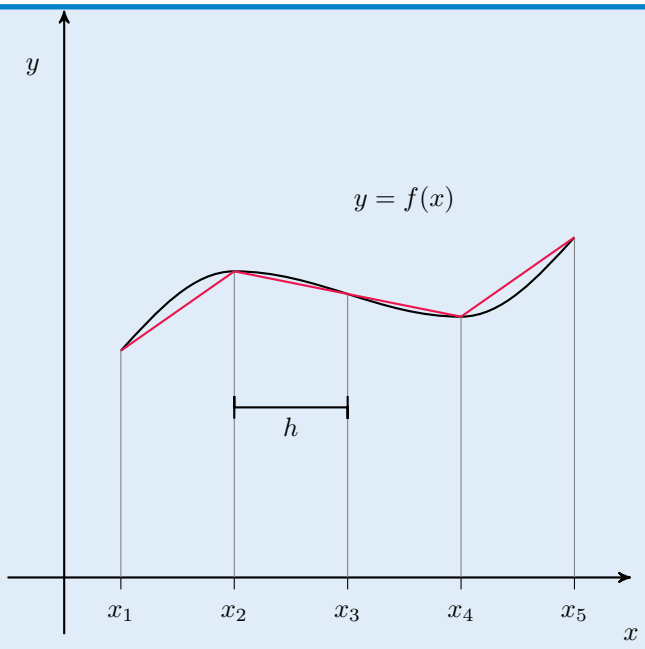


Figure 1.2: A depiction of the trapezoid rule with uniform partitioning.

Intercommunication

The phrase *parallel computing* refers to designing an architecture and code that makes the best use of computing resources for a problem. Occasionally, this will require nodes to be interdependent on each other for previous results. This contributes to a slower result because it requires a great deal of communication latency, but is sometimes the only method to parallelize a function. Although important, the ability to effectively communicate between engines has not been added to iPyParallel. It is, however, possible in an MPI framework and will be covered in the MPI lab.

Additional Material

Clusters of Multiple Machines

Though setting up a computing cluster with `iPyParallel` on multiple machines is similar to a cluster on a single computer, there are a couple of extra considerations to make. The majority of these considerations have to do with the network setup of your machines, which is unique to each situation. However, some basic steps have been taken from `https://ipyparallel.readthedocs.io/en/latest/process.html` and are outlined below.

SSH Connection

When using engines and controllers that are on separate machines, their communication will most likely be using an SSH tunnel. This *Secure Shell* allows messages to be passed over the network.

In order to enable this, an SSH user and IP address must be established when starting the controller. An example of this follows.

```
$ ipcontroller --ip=<controller IP> --user=<user of controller> --enginessh=<↩
    user of controller>@<controller IP>
```

Engines started on remote machines then follow a similar format.

```
$ ipengine --location=<controller IP> --ssh=<user of controller>@<controller IP↩
    >
```

Another way of affecting this is to alter the configuration file in `UserDirectory/.ipython/profile-default/security/ipcontroller-engine.json`. This can be modified to contain the controller IP address and SSH information.

All of this is dependent on the network feasibility of SSH connections. If there are a great deal of remote engines, this method will also require the SSH password to be entered many times. In order to avoid this, the use of SSH Keys from computer to computer is recommended.

Magic Methods & Decorators

To be more easily usable, the `iPyParallel` module has incorporated a few magic methods and decorators for use in an interactive iPython or Python terminal.

Magic Methods

The `iPyParallel` module has a few magic methods that are very useful for quick commands in iPython or in a Jupyter Notebook. The most important are as follows. Additional methods are found at `https://ipyparallel.readthedocs.io/en/latest/magics.html`.

%px - This magic method runs the corresponding Python command on the engines specified in `dview.targets`.

%autopx - This magic method enables a boolean that runs any code run on every engine until `%autopx` is run again.

Examples of these magic methods with a client and four engines are as follows.

```
# %px
In [4]: with dview.sync_imports():
   ...:     import numpy
   ...:
importing numpy on engine(s)
In [5]: \%px a = numpy.random.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]]
```

1 Naive Bayes

Lab Objective: *Create a Naïve Bayes Classifier. Use this classifier, and Sklearn's premade classifier to make an SMS spam filter.*

About Naïve Bayes

Naïve Bayes classifiers are a family of machine learning classification methods that use Bayes' theorem to probabilistically categorize data. They are called naïve because they assume independence between the features. The main idea is to use Bayes' theorem to determine the probability that a certain data point belongs in a certain class, given the features of that data. Despite what the name may suggest, the naïve Bayes classifier is not Bayesian method. This is because naïve Bayes is based on likelihood rather than Bayesian inference.

While naïve Bayes classifiers are most easily seen as applicable in cases where the features have, ostensibly, well defined probability distributions (such as classifying sex given physical characteristics), they are applicable in many other cases. While it is generally a bad idea to assume independence naïve Bayes classifiers are still very effective, even when we can be confident there is nonzero covariance between features.

The Classifier

You are likely already familiar with Bayes' Theorem, but we will review how we can use Bayes' Theorem to construct a robust machine learning model.

Given the feature vector of a piece of data we want to classify, we want to know which of all the classes is most likely. Essentially, we want to answer the following question

$$\text{argmax}_{k \in K} P(C = k|\mathbf{x}), \tag{1.1}$$

where C is the random variable representing the class of the data. Using Bayes' Theorem, we can reformulate this problem into something that is actually computable. We find that for any $k \in K$ we have

$$P(C = k|\mathbf{x}) = \frac{P(C = k)P(\mathbf{x}|C = k)}{P(\mathbf{x})}.$$

Now we will examine each feature individually and use the chain rule to expand the following expression

$$\begin{aligned} P(C=k)P(\mathbf{x}|C=k) &= P(x_1,\dots,x_n,C=k) \\ &= P(x_1|x_2,\dots,x_n,C=k)P(x_2,\dots,x_n,C=k) \\ &= \dots \\ &= P(x_1|x_2,\dots,x_n,C=k)P(x_2|x_3,\dots,x_n,C=k)\cdots P(x_n|C=k)P(C=k), \end{aligned}$$

and applying the assumption that each feature is independent we can drastically simplify this expression to the following

$$P(x_1|x_2,\dots,x_n,C=k)\cdots P(x_n|C=k) = \prod_{i=1}^{n} P(x_i|C=k).$$

Therefore we have that

$$P(C=k|\mathbf{x}) = \frac{P(C=k)}{P(\mathbf{x})}\prod_{i=1}^{n} P(x_i|C=k),$$

which reforms Equation 1.1 as

$$\text{argmax}_{k\in K} P(C=k)\prod_{i=1}^{n} P(x_i|C=k). \tag{1.2}$$

We drop the $P(\mathbf{x})$ in the denominator since it is not dependent on k.

This problem is approximately computable, since we can use training data to attempt to find the parameters which describe $P(x_i|C=k)$ for each i,k combination, and $P(C=k)$ for each k. In reality, a naïve Bayes classifier won't often find the actual correct parameters for each distribution, but in practice the model does well enough to be robust. Something to note here is that we are actually computing $P(C=k|\mathbf{x})$ by finding $P(C=k,\mathbf{x})$. This means that naïve Bayes is a generative classifier, and not a discriminative classifier.

Spam Filters

A spam filter is just a special case of a classifier with two classes: spam and not spam (or ham). We can now describe in more detail how we are to calculate Equation 1.2 given that we know what the features are. We can use a labeled training set to determine $P(C=\text{spam})$ the probability of spam and $P(C=\text{ham})$ the probability of ham. To do this we will assume that the training set is a representative sample and define

$$P(C=\text{spam}) = \frac{N_{\text{spam}}}{N_{\text{samples}}}, \tag{1.3}$$

and

$$P(C=\text{ham}) = \frac{N_{\text{ham}}}{N_{\text{samples}}}. \tag{1.4}$$

Using a bag of words model, we can create a simple representation of $P(x_i|C=k)$ where x_i is the i^{th} word in a message, and therefore $\mathbf{x}$ is the entire message. This results in the simple definition of

$$P(x_i|C=k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k}}{N_{\text{words in class } k}}. \tag{1.5}$$

Note that the denominator in Equation 1.5 is not the number of unique words in class k, but the total number of occurrences of any word in class k. In the case we have some word x_u that is not found in the training set, we can may choose $P(x_u|C=k)$ so that the computation is not effected, i.e. letting $P(x_u|C=k)=1$ for unique words.

A First Model

When building a naïve Bayes classifier we need to choose what probability distribution we believe our features to have. For this first model, we will assume that the words are a categorically distributed random variable. This means the random variable may take on say N different values, each value has a certain probability of occurring. This distribution can be thought of as a Bernoulli trial with N outcomes instead of 2.

In our situation we may have N different words which we expect may occur in a spam or ham message, so we need to use the training data to find each word and its associated probability. In order to do this we will make a DataFrame that will allow us to calculate the probability of the occurrence of a certain word x_i based on what percentage of words in the training set were that word x_i. This DataFrame that will allow us to more easily compute Equation 1.5, assuming the words are categorically distributed. While we are creating this DataFrame, it will also be a good opportunity to compute Equations 1.3 and 1.4.

Throughout the lab we will use an SMS spam dataset contained in `sms_spam_collection.csv`. We will use portions of data to check our progress. This codes example makes full test and train sets, but we will provide you opportunity to check against certain subsets.

```
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split

>>> # load in the sms dataset
>>> df = pd.read_csv('sms_spam_collection.csv')

>>> # separate the data into the messages and labels
>>> X = df.Message
>>> y = df.Label

>>> # split the data into test and train sets
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7)
```

Training The Model

Problem 1. Create a class `NaiveBayesFilter`, with an `__init__()` method that be may empty. Add a `fit()` method which takes as arguments `X`, the training data, and `y` the training labels. In this case `X` is a `pandas.Series` containing strings that are SMS messages. For each message in `X` count the number of occurrences of each word and record this information in a DataFrame.

The final form of the DataFrame should have a column for each unique word that appears in any message of `X` as well as a `Label` column, you may also include any other columns you think you'll need. Each row of the DataFrame corresponds to a row of `X`, and records the number of occurrences of each word in a given message. The `Label` column records the label of the message. e.g., `df.loc[5,'red']`) gives the number of times the word 'red' appears in message 5 (assuming that 'red' appears in any of the messages).

Save this DataFrame as `self.data`.

HINT: Ensure that the index of the DataFrame matches the index of `X` and `y`, by setting

`index=X.index` (or `index=y.index`) when creating the DataFrame.

HINT: Be sure that you are counting the number of occurrences of a word and not a string, for example: when searching the string `'find it in there'` for the word `'in'`, make sure your get 1 and not 2 (becuause of the `'in'` in `'find'`). `pd.Series.str.split()`, and the `count()` methods may be helpful.

Predictions

Now that we have implemented the `fit()` method, we can begin to classify new data. We will do this with two methods, the first will be a method that calculates $P(S|\mathbf{x})$ and $P(H|\mathbf{x})$, and the other will determine the more likely of the two and assign a label. While it may seem like we should have $P(C = S|\mathbf{x}) = 1 - P(C = H|\mathbf{x})$, we do not. This would only be true if we assume the S and H are independent of $\mathbf{x}$. It is clear that we shouldn't make this assumption, because we are trying to determine the likelihood of S and H based on what $\mathbf{x}$ tells us. Therefore we must compute both $P(C = S|\mathbf{x})$ and $P(C = H|\mathbf{x})$.

Problem 2. Implement the `predict_proba()` method in your naïve Bayes classifier. This should take as an argument `X`, the data that needs to be classified. For each message `x` in `X` compute $P(S|\mathbf{x})$ and $P(H|\mathbf{x})$ using Equations 1.3, 1.4, and 1.5.

The method should return an (`Nx2`) array, where `N` is the length of `X`. The first column corresponds to $P(C = H|\mathbf{x})$, and the second to $P(C = S|\mathbf{x})$.

Problem 3. Implement the `predict()` method in your naïve Bayes classifier. This should take as an argument `X`, the data that needs to be classified. Implement equation 1.2 and return an array that classifies each message in `X`.

```
>>> # create the filter
>>> NB = NaiveBayesFilter()

>>> # fit the filter to the first 300 data points
>>> NB.fit(X[:300], y[:300])

>>> # test the predict function
>>> NB.predict(X[530:535])
array(['ham', 'spam', 'ham', 'ham', 'ham'], dtype=object)

>>> # score the filter on the last 300 data points
>>> # score will use your predict() method
>>> NB.score(X[-300:], y[-300:])
0.9233333333333333
```

Underflow

There are some issues that we encounter given this implementation. Notice that in the following example, the likelihoods for both spam and ham are 0 for each message.

```
>>> # find the likelihoods for messages 1085 and 2010
>>> NB.predict_proba(X[[1085,2010]])
array([[0., 0.],
       [0., 0.]])
```

This is because the messages are long, and thus involve the product of many numbers that are between 0 and 1. Because of this, we have encountered what is called underflow, where a number becomes so small it is not machine representable. Therefore, we should work in logspace, as to avoid inevitable underflow caused by long messages. If we take the log of Equation 1.2 have

$$\text{argmax}_{k\in K} \ln\left(P(C=k)\right) + \sum_{i=1}^{n} \ln\left(P(x_i|C=k)\right), \tag{1.6}$$

and this problem is still valid because log is monotone increasing.

Problem 4. Implement `predict_log_proba()` and `predict_log()` using equation 1.6. Notice how `X[[1085,2010]]` is now classifiable.

Optimizing the Model

As you may have noticed that while the DataFrame model is rather quick to train, it is very slow to classify. This is because of the in depth lookup that must be done for every word in every message of the testing data. While there are some ways to speed up this lookup process, like finding repeated words in an unclassified message, ultimately looking up the word frequencies and summing them is expensive. What we can do instead is do these lookups ahead of time. This will result in a DataFrame that is significantly smaller (of size $2 \times N_{\text{vocabulary}}$) and computation time that is around 50 times faster.

Problem 5. Implement the two following optimizations to the DataFrame filter

- Reduce the lookup time by altering the DataFrame created in the `fit()` method. The new DataFrame will have two rows, and $N_{\text{vocabulary}}$ columns, with `'spam'` and `'ham'` being the index. Each entry will be the number of times a word appears in spam or ham messages. E.g. `self.data.loc['ham','red']` is the number of times the word "red" appeard in ham messages. Alter the `predict_proba()` and `predict_log_proba()` to appropriately utilize this improvement.
- If not already implemented during Problem 2, instead of computing $\prod_{i=1}^{n} P(x_i|C)$ for a message with n words, find $\prod_{i=1}^{l} P(x_i|C)^{n_i}$ where l is the number of unique words in the message and n_i is number of times the i^{th} word occurs. (Since $P(x_i|C)$ is the same for any word that is repeated in a message, the lookup should only need to be done once.)

```
>>> # checkout what the new DataFrame looks like after the changes
>>> NB = NaiveBayesFilter()
>>> NB.fit(X[:300], y[:300])
>>> NB.data.loc['ham','i']
184
>>> NB.data.loc['spam','i']
4
```

The Poisson Model

Now that we've examine one way to constructing a naïve Bayes classifier, let us look at one more method. In the Poisson model we assume that each word is Poisson random variable, occurring with potentially different frequencies among spam and ham messages. Because each of the messages is a different length, we can reparameterize the Poisson PMF to the following

$$P(n_i = x) = \frac{(rn)^x e^{-rn}}{x!} \tag{1.7}$$

where n_i is the number of times word i occurs in a message, n is the length of the message, and $\lambda = rn$ is the classical Poisson rate. In this case r represents the number of events per unit time/space/etc.

While we will use maximum likelihood estimation to determine r, we could easily refactor this model to use Bayesian inference, allowing greater control over the model. This would create a condition where the training data doesn't completely determine the model's viability. I encourage you to do this refactor once you cover Bayesian inference and see how much better your results can be. (Try a beta prior with $a = 2, b = 5$)

Training the Model

Similar to the other classifier that we made, training the model amounts to using the training data to determine how $P(x_i|C = k)$ is computed, as well as compting $P(C = k)$. As stated earlier, we will attempt to find the most likely value of r for each word that appears in the training set. To do this we will use maximum likelihood estimation. The parameter we choose is the one that maximizes the likelihood function

$$\hat{r} = \text{argmax}_r \mathcal{L}(r|\mathbf{x}) = \text{argmax}_r P(\mathbf{x}|r).$$

In this case, since we are using a Poisson distribution (1.7) for each word, we will solve the following problem for both the spam class and the ham classes

$$r_{i,k} = \text{argmax}_{r\in[0,1]} \frac{(rN_k)^{n_i} e^{-rN_k}}{n_i!}, \tag{1.8}$$

where $r_{i,k}$ is the Poisson rate for word i in class k, N_k is the total number of words in class k (either spam or ham), and n_i is the number of times word i occurs in that class. We have $r \in [0, 1]$ because a word cannot occur more than once per word in the message.

Problem 6. Create a new class called `PoissonBayesFilter` with an `__init__()` method that may be empty. Add a `fit()` method which takes as arguments `X`, the training data, and `y` the training labels. Implement `fit()` by implementing the MLE algorithm found in 1.8 to predict r for each word in both the spam and ham classes, and therefore train the model. Store these computed rates in dictionaries called `self.spam_rates` and `self.ham_rates`, where the key is the word and the value is the associated r. (E.g. `self.ham_rates['i']` will give the computed r value for the word "i" in ham messages)

```
>>> #create a poisson bayes object to examine it
>>> PB = PoissonBayesFilter()
>>> PB.fit(X[:300], y[:300])

>>> # check spam and ham rate of 'i'
>>> PB.ham_rates['i']
0.04830483048304831
>>> PB.spam_rates['i']
0.0033003300330033004
```

Predictions

Making predictions with this model is exactly the same as we did earlier. To clarify the calculation, lets reformulate 1.6 to fit the Poisson case better. This gives

$$\text{argmax}_{k\in K} \ln\left(P(C=k)\right) + \sum_{i=1}^{l} \ln\left(\frac{(r_{i,k}n)^{n_i}e^{-r_i n}}{n_i!}\right), \tag{1.9}$$

with l being the number of unique words in a message, n_i the number of times the i^{th} word occurs, n the number of words in the message, and $r_{i,k}$ the Poisson rate of the i^{th} word in class k.

Problem 7. Implement the `predict_proba()` and `predict()` methods using Equation 1.9. These methods will expect the same arguments and return the same types as the previous problems. You may use `scipy.stats.poisson.pmf` if you wish.

Naive Bayes with Sklearn

Now that we have explored a few ways to implement out own naïve Bayes classifier, we can examine the tools from the sklearn library. Sklearn provides robust tools that will accomplish all the things that we've coded up so far.

First we will want to use `CountVectorizer` from `sklearn.feature_extraction.text`. This tool will essentially do the work of the first `fit()` method we wrote. The vectorizer must learn a dictionary as well as transform the training set, this can be done with the `fit_transform()` method. This will fit the vectorizer, i.e. create the dictionary, and transform the data.

Now we can use the transformed training data to fit a `MultinomialNB` model from `sklearn.naive_bayes`. Any data that we want to classify, we must first transform with the vectorizer (using

the `transform()` method, not the `fit_transform()` method), then we can classify it using the `predict()` method of the `MultinomialNB` model. This naïve Bayes model uses the multinomial distribution where we have used the categorical and poisson distributions. Multinomial is very similar to the categorical implementation, as the multinomial distribution models the outcome of n categorical trials (in the same way that the binomial distribution models n Bernoulli trials).

Problem 8. Write a function that will classify messages. It will take as arguments training data `X_train` and `y_train`, and test data `y_test`. In this function use the `CountVectorizer` and `MultinomialNB` from sklearn and return the predicted classification of the model.

References

Rish, Irina. (2001). An Empirical Study of the Naïve Bayes Classifier. IJCAI 2001 Work Empir Methods Artif Intell. 3.

Data from: http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/

1 K-Means Clustering

Lab Objective: *Clustering is the one of the main tools in unsupervised learning—machine learning problems where the data comes without labels. In this lab we implement the k-means algorithm, a simple and popular clustering method, and apply it to geographic clustering and color quantization.*

Jupyter Notebooks

Unlike previous labs where the python file submitted was a normal `.py` file, this lab among others will be done in a Jupyter Notebook (`.ipynb` or an iPython Notebook). Jupyter Notebooks is a powerful tool in visualizing data. If you have used Google Colab, this works in a similar manner but it is run on your personal machine.

Once Jupyter Notebook is installed, there are several ways of starting a Jupyter Notebook. The easiest way is to open a new terminal window and navigate to the directory with your .ipynb file, once in the desired directory, type Jupyter notebook. This should automatically open a web browser to the Jupyter Notebook dashboard, from there you can select the .ipynb file and open and edit it.

The Python kernel will keep running in the background until told to stop. So when you are done, to close the Jupyter Notebook, you need to go to file-> Close and Halt, or in the terminal window press ctrl+c (cmd+c for Mac).

> ACHTUNG!
>
> Before you push this file to Bitbucket to be graded, be sure to run each cell. When you push a .ipynb file, the current state of the file is pushed. This means what you see is exactly what the graders will see.

Clustering

In this lab, we will analyze a few different datasets from Scikit-Learn's library and use the K-means algorithm. Figure 1.1 is a graph of the iris dataset. As a human, it is easy to identify the two distinct groups of data. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*. The K-means algorithm is a simple way of helping computers see the group distinctions.

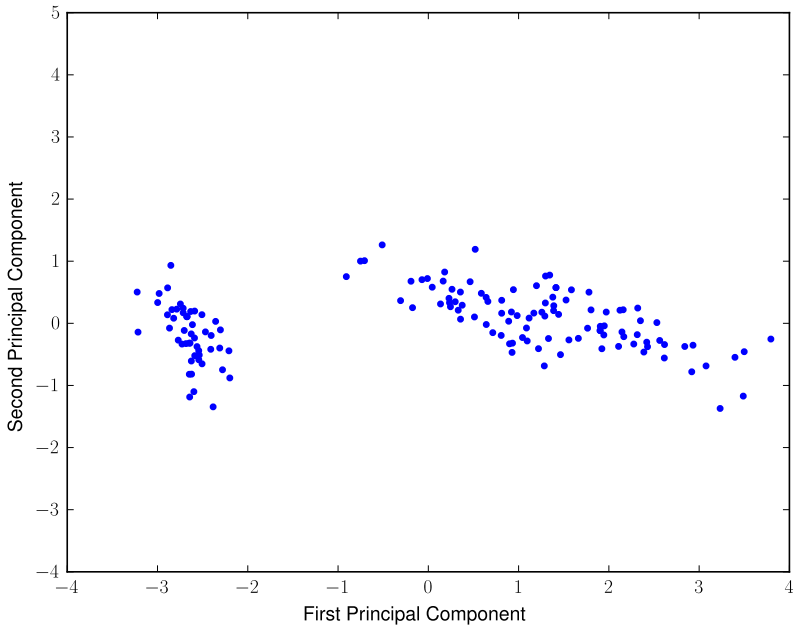


Figure 1.1: The first two principal components of the iris dataset.

The objective of clustering is to find a partitions of the data such that points in the same subset will be "close" according to some metric. The metric used will likely depend on the data, but some obvious choices include Euclidean distance and angular distance. Throughout this lab, we will use the metric $d(x, y) = \|x - y\|_2$, the Euclidean distance between x and y, unless we specify a different metric to be used.

More formally, suppose we have a collection of $\mathbb{R}^K$-valued observations $X = \{x_1, x_2, \ldots, x_n\}$. 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 $S = \{S_1, S_2, \ldots, S_N\}$, where

$$X = \bigcup_{i=1}^{N} S_i$$

and

$$|S_i| > 0, \qquad i = 1, 2, \ldots, N.$$

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.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

The K-Means Algorithm

Finding the global minimizing partition S^* is generally intractable since the set of partitions can be very large indeed, but the *k-means* algorithm is a heuristic approach that can often provide reasonably

accurate results.

We begin by specifying an initial cluster mean $\mu_i^{(1)}$ for each $i = 1, \cdots, N$. 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

$$S_i^{(t)} = \{x_j \ : \ \|x_j - \mu_i^{(t)}\|_2^2 \leq \|x_j - \mu_l^{(t)}\|_2^2, \ \ l = 1, \cdots, N\}.$$

We then update our cluster means by computing for each $i = 1, \cdots, N$. We continue to iterate in this manner until the partition ceases to change.

Figure 1.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

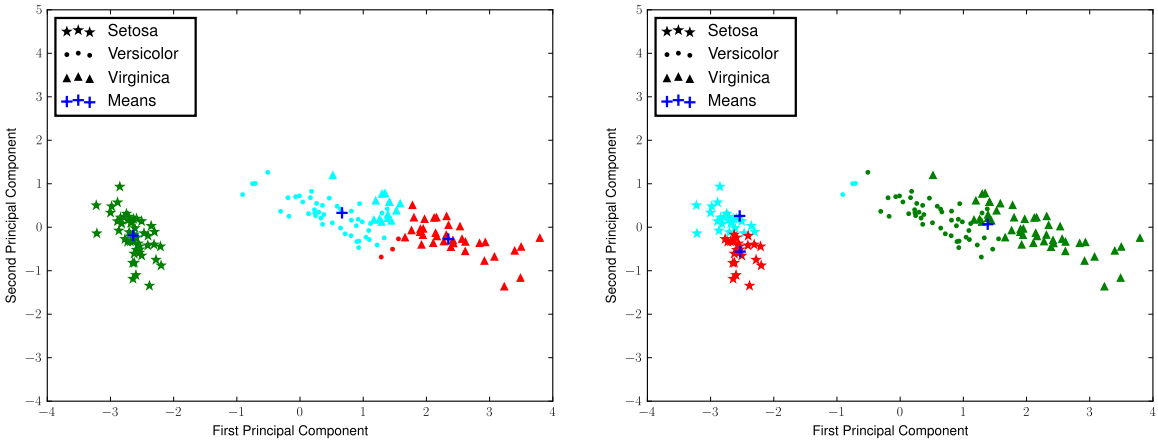


Figure 1.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

The algorithm can be summarized as follows.

1. From the data points, choose k initial cluster centers.
2. For $i = 0, \ \ldots, \ \texttt{max\_iter}$,
   (a) Assign each data point to the cluster center that is closest, forming k clusters.
   (b) Recompute the cluster centers as the means of the new clusters.
   (c) If the old cluster centers and the new cluster centers are sufficiently close, terminate early.

Problem 1. Write a `KMeans` class for doing basic k-means clustering. Implement the following methods, following `sklearn` class conventions.

1. `__init__()`: Accept a number of clusters k, a maximum number of iterations, and a convergence tolerance. Store these as attributes.
2. `fit()`: Accept an $m \times n$ matrix X of m data points with n features. Choose k random rows of X as the initial cluster centers. Run the k-means iteration until consecutive centers are within the convergence tolerance, or until iterating the maximum number of times. Save the cluster centers as attributes.

   If a cluster is empty, reassign the cluster center as a random row of X.
3. `predict()`: Accept an $l \times n$ matrix X of data. Return an array of l integers where the ith entry indicates which cluster center the ith row of X is closest to.

Test your class on the iris data set after reducing the data to two principal components. Plot the data, coloring by cluster.

Fire Station Placement

When urban planners are making plans for a city, there are many city elements to consider. One of which is the locations of the fire stations that will service the city. When choosing a suitable location for the city, urban planners look at the current building locations, the roads nearby each location, prior traffic history and the areas of potential growth. We will simplify this complex problem by only taking into account the distances from each building to the nearest fire station (see Additional Material for a harder version of this problem).

Using another data set from SKLearn, we can get the data from the 1990 US Census for California housing based on the blocks of the residents. This has been saved in `sacramento.npy` and can be accessed by using the np.load() function. This file contains demographic data for each block in Sacramento and nearby cities. The eight columns in the file are: median block income, median house age in the block, average number of rooms, average number of bedrooms, average house occupancy, latitude and longitude.

There are couple ways for a fire station to be optimally placed. The stations could be placed to minimize the average distance to each house. Another option is to minimize the distance to the farthest house in each group. For this problem, minimize the distance to the farthest house in each group.

Problem 2. Using the Methods you wrote in Problem 1, add a parameter, p, to your class that denotes the norm and defaults to 2. Save p as an attribute to be used in your fit() and predict() functions. Using the data in `sacramento.npy` find the optimal placement for the fire stations. Plot the longitude and latitudes, the centers, and color them by cluster. Try different values for `p` to find the optimal locations for the fire stations. As the initial centers are chosen at random, make sure to run the `predict()` function several times. In a Markdown cell report which norm was the best at keeping the maximum distance small.

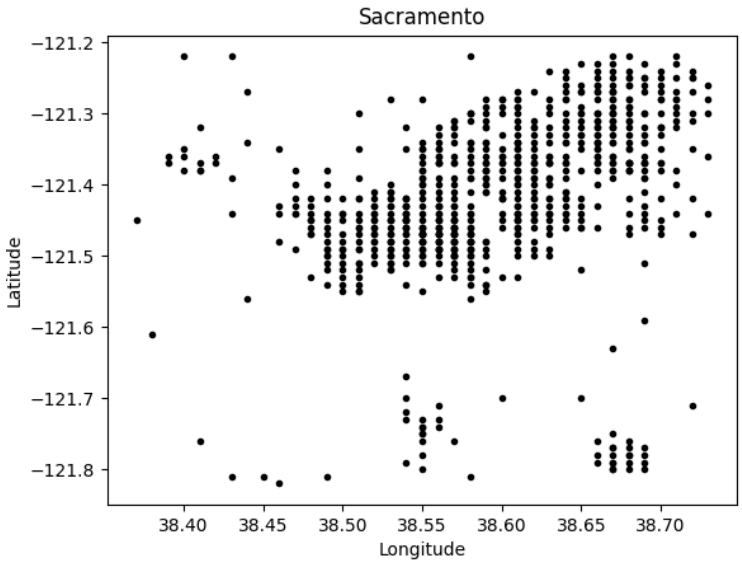


Figure 1.3: Sacramento Housing Data (1990 US Census).

Detecting Active Earthquake Regions

Suppose we are interested in learning about which regions are prone to experience frequent earthquake activity. We could make a map of all earthquakes over a given period of time and examine it ourselves, but this, as an unsupervised learning problem, can be solved using our k-means clustering tool.

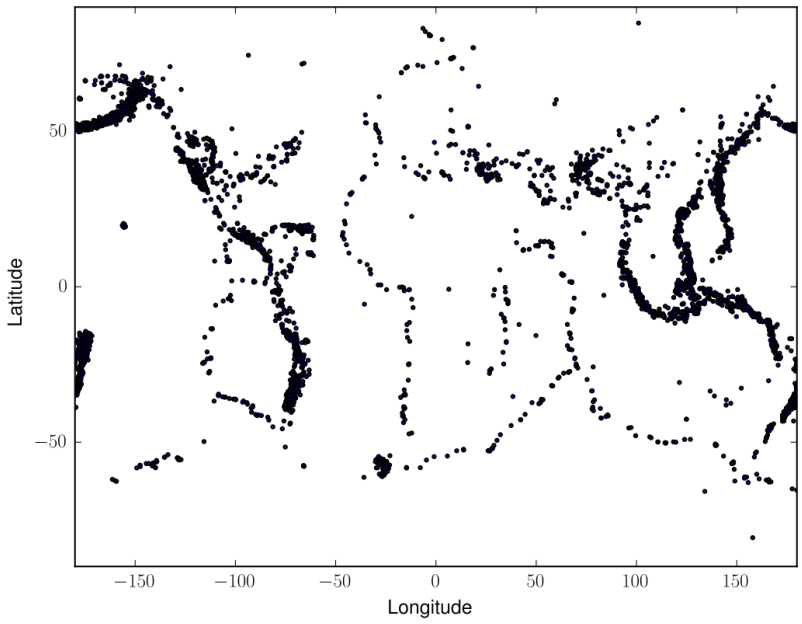


Figure 1.4: Earthquake epicenters over a 6 month period.

The file `earthquake_coordinates.npy` contains earthquake data throughout the world from January 2010 through June 2010. Each row represents a different earthquake; the columns are scaled longitude and latitude measurements. We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in $\mathbb{R}^2$ with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. Instead, latitude and longitude should be viewed in *spherical coordinates* in $\mathbb{R}^3$, which could then be clustered.

A simple way to accomplish this transformation is to first transform the latitude and longitude values to spherical coordinates, and then to Euclidean coordinates. Recall that a spherical coordinate in $\mathbb{R}^3$ is a triple (r, 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, once the longitude is converted to radians it is an appropriate θ value; the latitude needs to be offset by $90°$ degrees, then converted to radians to obtain φ. For simplicity, we can take $r = 1$, since the earth is roughly a sphere. We can then transform to Euclidean coordinates using the following relationships.

$$\theta = \frac{\pi}{180}\,(\text{longitude}) \qquad \varphi = \frac{\pi}{180}\,(90 - \text{latitude})$$

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} & x &= r\sin\varphi\cos\theta \\ \varphi &= \arccos\frac{z}{r} & y &= r\sin\varphi\sin\theta \\ \theta &= \arctan\frac{y}{x} & z &= r\cos\varphi \end{aligned}$$

There is one last issue to solve before clustering. Each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. Therefore, the cluster centers should also have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth, and the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors.

Problem 3. Add a keyword argument `normalize=False` to your `KMeans` constructor. Modify `fit()` so that if `normalize` is `True`, the cluster centers are normalized at each iteration.

Cluster the earthquake data in three dimensions by converting the data from raw data to spherical coordinates to euclidean coordinates on the sphere.

1. Convert longitude and latitude to radians, then to spherical coordinates.
   (Hint: `np.deg2rad()` may be helpful.)
2. Convert the spherical coordinates to euclidean coordinates in $\mathbb{R}^3$.
3. Use your `KMeans` class with normalization to cluster the euclidean coordinates.
4. Translate the cluster center coordinates back to spherical coordinates, then to degrees. Transform the cluster means back to latitude and longitude coordinates.
   (Hint: use `numpy.arctan2()` for arctan, so that that correct quadrant is chosen).
5. Plot the data, coloring by cluster. Also mark the cluster centers.

With 15 clusters, your plot should resemble the Figure 1.5.

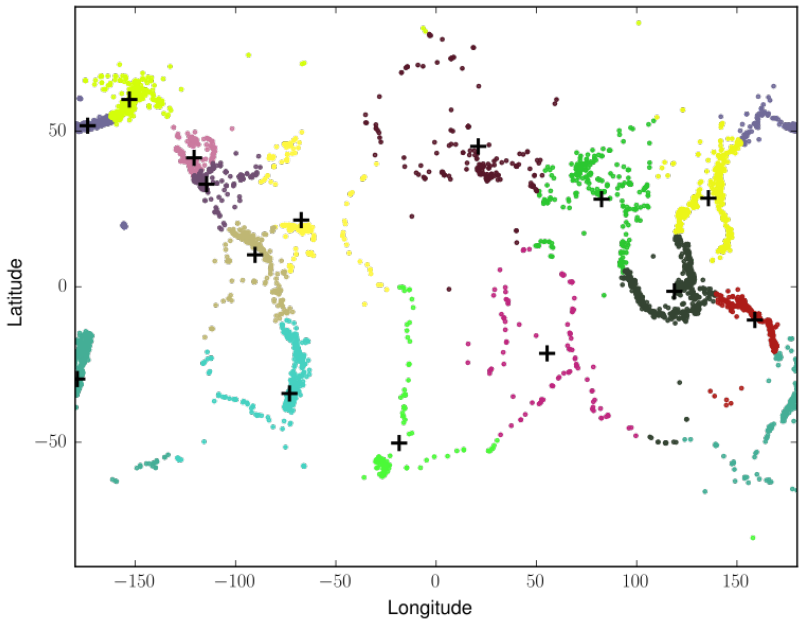


Figure 1.5: Earthquake epicenter clusters with $k = 15$.

Color Quantization

The k-means algorithm uses the euclidean metric, so it is natural to cluster geographic data. However, clustering can be done in any abstract vector space. The following application is one example.

Images are usually represented on computers as 3-dimensional arrays. Each 2-dimensional layer represents the red, green, and blue color values, so each pixel on the image is really a vector in $\mathbb{R}^3$. Clustering the pixels in RGB space leads a one kind of image segmentation that facilitate memory reduction.

Reading: `https://en.wikipedia.org/wiki/Color_quantization`

Problem 4. Write a function that accepts an image array (of shape $(m, n, 3)$), an integer number of clusters k, and an integer number of samples S. Reshape the image so that each row represents a single pixel. Choose S pixels to train a k-means model on with k clusters. Make a copy of the original picture where each pixel has the same color as its cluster center. Return the new image. For this problem, you may use `sklearn.cluster.KMeans` instead of your `KMeans` class from Problem 1.

Test your function on some of the provided NASA images.

Additional Material

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*, $L_{sym} = I - D^{-1/2} W D^{-1/2}$
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 $u_1, \cdots, u_k$ of the Laplacian matrix.
- Set $U = [u_1, \cdots, u_k]$, 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 $x_1, \cdots, x_n$.

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 5. 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 6. 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
```

Fire Station Placement II

In problem 2 we looked at choosing the best location for a fire station. However, because we looked at the city of Sacramento where the geography doesn't role in choosing a location, we didn't need to double check that there is a place for the station. The `sanfrancisco.npy` data is organized the same way as `sacramento.py`, as this also comes from the SKLearn California Housing Module. Doing the same method as before will give us groups of houses, however, the group centers may be in the middle of the bay. When implementing this problem, perform a check on the centers to make sure they are not in water. The file `bayboundary.npy` gives a rough outline of where the bay is. The `bayboundary.npy` has only 2 columns, longitude and latitude. Using the boundaries set, make sure that the chosen centers are on land and not on water.

Problem 7. Import and parse the data from the `bayboundary.npy` and the `sanfrancisco.npy` files. Using either the algorithm that you wrote in problem 1 or the k-means algorithm in the SK Learn library, find the optimal locations for the 16 fire stations.

After the algorithm has finished running, check to see if the new coordinates are on land. Return the graph of the clusters, the centers (the fire station locations) as different colors.