1 Sampling

Lab Objective: *Sampling is an important and fundamental tool in statisical modeling. In this lab we will learn to use PyMC3 for Bayesian modeling and statical sampling.*

Sampling

When seeking to understand a group or a phenomenon, we, as data scientists, will often look to find some sort of sample. A good sample can tell us a lot, and while we will not examine what makes a good sample in this lab, we will examine how much a good sample can tell us. One goal of Bayesian statistics is to be able to quantify, with degrees of certainty, how much we can learn from a sample, given what we already know about its source. This quantification of certainty allows us to extract more information and nuance from a sample than would otherwise be possible, and in turn allow us to better predict and describe events.

Parameter Estimation

Maximum Likelihood Estimation

Maximum Likelihood Estimation is a frequentist approach to parameter estimation. We will first examine the derivation of the maximum likelihood estimate (MLE) from the frequentist point of view, then examine how it is a special case of the Bayesian method of finding the *maximum a posteriori estimate* (MAP).

Likelihood

Finding the maximum likelihood involves, unsurprisingly, maximizing the likelihood function, which is defined as follows

$$\mathcal{L}(\theta) = \mathcal{L}(\theta|\mathbf{x}) = f(\mathbf{x}|\theta),$$

where $\mathbf{x}$ is a sample, θ is the parameter we are estimating and f is the pdf. Determining the MLE $\hat{\theta}$ is as simple as finding the argmax of $\mathcal{L}$

$$\hat{\theta} = \text{argmax}_{\theta\in\Theta}\mathcal{L}(\theta|\mathbf{x}).$$

Maximum A Posteriori Estimate

If we examine closely, we can see a similarity between the likelihood function and Bayes' rule. Bayes' rule gives the following relation

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

To apply this rule to the problem of parameter estimation we get the following relation

$$f(\theta|\mathbf{x}) = \frac{f(\mathbf{x}|\theta)g(\theta)}{\int_\Theta f(\mathbf{x}|\vartheta)g(\vartheta)d\vartheta}. \tag{1.1}$$

We call $f(\theta|\mathbf{x})$ the posterior distribution and $g(\theta)$ the prior distribution. The MAP estimate is then the argmax of the posterior

$$\theta_{MAP} = \text{argmax}_{\theta\in\Theta} f(\theta|\mathbf{x}). \tag{1.2}$$

When finding the MAP estimate, the exact posterior is often left uncalculated because it is difficult to compute. Instead we approximate it by finding its value at grid points of θ. Similarly, because of the complexity of the denominator, we often don't find it until the end of the process. After we calculate $f(\mathbf{x}|\theta)g(\theta)$ for each relevant θ we can then approximate the integral in the denominator with a finite sum. First we find $f(\mathbf{x}|\theta_i)g(\theta_i)$ for a grid of θ values. Then, we can approximate the integral in denominator with the following sum $\sum_i f(\mathbf{x}|\theta_i)g(\theta_i)$.

Here we can see that the likelihood function is similar to Bayes' rule as long as we take $g(\theta)$ to be a constant, i.e. $\theta \sim \mathcal{U}(a, b)$. This means that the MAP estimate is the MLE if we assume a uniform prior distribution.

Problem 1. Write a function called `bernoulli_sampling()` that takes the following parameters: `p` a float that is the "fairness" of a coin and `n` the size of the sample to be generated. In this function simulate `n` tosses of a coin which gives heads with probability `p`. Then use that sample to calculate the posterior distribution on p given a uniform prior using Equation 1.2.

For `p`=.2 and `n`=100 plot the posterior distribution and return the MAP estimate of p, which is also the MLE in this case.

Hint: In this case f is the Binomial pmf $f(x) = p^{nx}(1-p)^{n(1-x)}$. You do not need to calculate the integral in the denominator exactly; since you are using a finite approximation of the distributions, you may use a finite approximation of the integral. You may simulate the tosses of a coin by using `np.random.binomial()` or `scipy.stats.binom()`.

Non-Uniform Priors

While we are able to get good estimates, we leave a lot of the power of Bayesian statistics on the table when we only use a uniform prior. While the uniform prior is free from any preconceptions or biases, it also imparts the least amount of information. Using a non-uniform prior allows us to actually incorporate prior knowledge or assumptions into our model. If we have good reason to believe something about a parameter we are exploring before we even draw a sample, we can learn a lot more by accounting for those beliefs.

Problem 2. Suppose you choose a coin from a bag that produces coins of many weights. However, the bag seems to be more likely to produce coins that are strongly biased in favor of heads. You're unsure of which kind of coin you've drawn so in order to find out you perform 20 flips.

Write a function called `non_uniform_prior()` that takes the following parameters: `p` a float that is the "fairness" of a coin, `n` the size of the sample to be generated, and `prior` a SciPy distribution object which will act as the prior on p.

Similar to Problem 1, simulate `n` flips and calculate and plot the posterior distribution. Return the MAP estimate.

Examine the difference in confidence we can have in estimating the bias of the coin if the coin we draw gives heads 90% of the time as opposed to 40% of the time.

Because we think that coins biased in favor of heads are likely, we can choose a prior distribution that matches that assumption. In this case we will choose Beta$(5, 1.5)$ as the prior distribution because it gives much more weight to parameters larger than .5. This is most easily achieved with `scipy.stats.beta(5,1.5)` and using the `pdf()` method to calculate $g(\theta)$

Sampling from a Markov Chain

A Markov chain is a way to model sequences of states or events. Markov chains make a few assumptions, one of those being that the probability of each state occurring is dependent only on the previous state. The relationship between the states are described by what is called a transition matrix.

Markov chains and sampling are like peanut butter and jelly: neither one really lives up to their full potential without the other. Given the transition matrix of a Markov chain, we can use sampling to better understand what that chain looks and acts like or to get a well-informed idea of what the future may hold.

Sampling from a simple (row stochastic) transition matrix like the one below is as simple as picking a starting state s_0, and then using the corresponding row to sample randomly using the probabilities in the row.

	a	b	c
a	0.7	0.1	0.2
b	0.5	0.4	0.1
c	0.1	0.8	0.1

For example, using the above transition matrix, let $s_0 = a$. Then I will randomly sample from the array $[a, b, c]$ using the respective probabilities $[0.7, 0.1, 0.2]$. If the sample gives me c, then. $s_1 = c$ and we can continue the process to find $s_2, ..., s_n$.

Problem 3. Given the transition matrix below and assuming the 0th day is sunny, sample from the markov chain to give a possible forecast of the 10 following days. Return a list of strings.

	sun	rain	wind
sun	0.6	0.1	0.3
rain	0.2	0.6	0.2
wind	0.3	0.4	0.3

Hint: `np.random.choice()` may be helpful here.

PyMC3

Python has many powerful sampling tools. Among these is PyMC3, an efficient implementation of a method known as Monte Carlo Markov Chain (MCMC) Sampling. This is a useful technique as it constructs a Markov Chain whose steady state is a probability distribution that is difficult to sample from directly. Unlike our simple Markov Chain from the last problem, certain Markov Chains are abstract. PyMC3 gives us a way to work with these more complex scenarios.

Single Variable PyMC3

Consider the following: owners of a restaurant are trying to decide if they should keep selling nachos. They gather the data for several months about how many people order nachos each day. One of the owners happened to take a class in Bayesian statistics in college, so she decides test her knowledge. She assumes the data are distributed as Poisson(λ) for some unknown value of λ where λ has a prior of Gamma(2,2). She sets up a PyMC3 Model for the situation as follows:

```
import pymc3 as pm
import numpy as np
import arviz as az #visualization package

model = pm.Model()
with model:
    lam = pm.Gamma('lambda', alpha=2, beta=2)
    y = pm.Poisson('y', mu=lam, observed=nacho_data)
    trace = pm.sample(n) #n is the desired number of samples

    az.plot_trace(trace)
    lam = trace['lambda']
    mean = lam.mean()
```

What this has done is created a model for the Poisson Distribution as well as λ itself. It then samples from the posterior to give us the expected value of λ (mean). It also gives a sample trace of length n from posterior, and the trace plot.

We will now reconsider the initial problem of the coin flip.

Problem 4. Create a function the accepts the coin flip data in array form and an integer n for desired number of samples. Given data that flips a coin 100 times, assume the data are distributed as Bernoulli(p) for some unknown value of p, where p has a prior of Beta(1,1). Set up a PyMC3 model for this situation and sample from the posterior n times. Print a trace plot. Return the mean for the posterior.

Run the function with data generated by the following code

```
from scipy.stats import bernoulli
data = bernoulli.rvs(0.2, size=30)
```

Multivariate PyMC3

Unlike the Poisson and Bernoulli distributions, many other distributions including the Normal, Beta, Gamma, and Binomial distributions have two or more parameters. These problems are where PyMC3 becomes very useful.

Problem 5. Create a function the accepts the height data in array form and an integer n for desired number of samples. Given a dataset of the measured heights of 100 men, assume the data are distributed as Normal(mu,1/tau) where mu has a prior of Normal(m,s), and tau has a prior of Gamma(alpha,beta). Set up a PyMC3 model for this situation and sample from the posterior n times. Print a trace plot for mu and tau. Return the mean for the posterior of mu.

Run the function with data generated by the following code

```
heights = np.random.normal(180,10,100)
```

2 Apache Spark

Lab Objective: *Dealing with massive amounts of data often requires parallelization and cluster computing; Apache Spark is an industry standard for doing just that. In this lab we introduce the basics of PySpark, Spark's Python API, including data structures, syntax, and use cases. Finally, we conclude with a brief introduction to the Spark Machine Learning Package.*

Apache Spark

Apache Spark is an open-source, general-purpose distributed computing system used for big data analytics. Spark is able to complete jobs substantially faster than previous big data tools (i.e. Apache Hadoop) because of its in-memory caching, and optimized query execution. Spark provides development APIs in Python, Java, Scala, and R. On top of the main computing framework, Spark provides machine learning, SQL, graph analysis, and streaming libraries.

Spark's Python API can be accessed through the PySpark package. You must install Spark, along with the supporting tools like Java, on your local machine for PySpark to work. This will include ensuring that both Java and Spark are included in the environment variable `PATH`. [1] Installation of PySpark for local execution or remote connection to an existing cluster can be done with `conda` or `pip` commands.[2]

```
# install Java
$ sudo apt-get install openjdk-8-jdk
# check the version, it may not be exactly the same
$ java -version
openjdk version "1.8.0_242"
OpenJDK Runtime Environment (build 1.8.0_242-b09)
OpenJDK 64-Bit Server VM (build 25.242-b09, mixed mode)

# Install Spark by following instructions in the footnote
# Following these steps, you must configure your PATH environment variable

# CHOOSE ONE
```

[1] See the Apache Spark configuration instructions for detailed installation instructions

[2] You may also use the script provided with the spec file that will completely install Spark and its requirements. Note however that this script is provided AS IS and is not the recommended method.

```
# PySpark installation with conda
$ conda install -c conda-forge pyspark

# PySpark installation with pip
$ pip install pyspark
```

If you use `python3` in your terminal, you will need to set the `PYSPARK_PYTHON` environment variable to `python3`. When using an IDE, you must call it from the terminal or set the variables inside the editor so that PySpark can be found.

PySpark

One major benefit of using PySpark is the ability to run it in an interactive environment. One such option is the interactive Spark shell that comes prepackaged with PySpark. To use the shell, simply run `pyspark` in the terminal. In the Spark shell you can run code one line at a time without the need to have a fully written program. This is a great way to get a feel for Spark. To get help with a function use `help(function)`; to exit the shell simply run `quit()`.

In the interactive shell, the `SparkSession` object - the main entrypoint to all Spark functionality - is available by default as `spark`. When running Spark in a standard Python script (or in IPython) you need to define this object explicitly. The code box below outlines how to do this. It is standard practice to name your `SparkSession` object `spark`.

It is important to note that when you are finished with a `SparkSession` you should end it by calling `spark.stop()`.

> **Note**
>
> While the interactive shell is very robust, it may be easier to learn Spark in an environment that you are more familiar with (like IPython). To do so, just use the code given below. Help can be accessed in the usual way for your environment. Just remember to `stop()` the `SparkSession`!

```
>>> from pyspark.sql import SparkSession

# instantiate your SparkSession object
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()

# stop your SparkSession
>>> spark.stop()
```

> **Note**
>
> The syntax

```
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()
```

is somewhat unusual. While this code can be written on a single line, it is often more readable to break it up when dealing with many chained operations; this is standard styling for Spark. Note that you *cannot* write a comment after a line continuation character '\'.

Resilient Distributed Datasets

The most fundamental data structure used in Apache Spark is the Resilient Distributed Dataset (RDD). RDDs are immutable distributed collections of objects. They are *resilient* because performing an operation on one RDD produces a *new* RDD without altering the original; if something goes wrong, you can always go back to your original RDD and restart. They are *distributed* because the data resides in logical partitions across multiple machines. While RDDs can be difficult to work with, they offer the most granular control of all the Spark data structures.

There are two main ways of creating RDDs. The first is reading a file directly into Spark and the second is parallelizing an existing collection (list, numpy array, pandas dataframe, etc.). We will use the Titanic dataset[3] in most of the examples throughout this lab. The example below shows various ways to load the Titanic dataset as an RDD.

```
# SparkSession available as spark
# load the data directly into an RDD
>>> titanic = spark.sparkContext.textFile('titanic.csv')

# the file is of the format
#Pclass,Survived,Name,Sex,Age,Sibsp,Parch,Ticket,Fare

# Survived | Class | Name | Sex | Age | Siblings/Spouses Aboard | Parents/↩
    Children Aboard | Fare

>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# note that each element is a single string - not particularly useful
# one option is to first load the data into a numpy array
>>> np_titanic = np.loadtxt('titanic.csv', delimiter=',', dtype=list)

# use sparkContext to parallelize the data into 4 partitions
>>> titanic_parallelize = spark.sparkContext.parallelize(np_titanic, 4)

>>> titanic_parallelize.take(2)
[array(['0', '3', ..., 'male', '22', '1', '0', '7.25'], dtype=object),
```

[3] https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html

```
 array(['1', '1', ..., 'female', '38', '1', '0', '71.2833'], dtype=object)]
```

ACHTUNG!

Because Apache Spark partitions and distributes data, calling for the first `n` objects using the same code (such as `take(n)`) may yield different results on different computers (or even each time you run it on one computer). This is not something you should worry about; it is the result of variation in partitioning and will not affect data analysis.

RDD Operations

Transformations

There are two types of operations you can perform on RDDs: *transformations* and *actions*. Transformations are functions that produce new RDDs from existing ones. Transformations are also lazy; they are not executed until an *action* is performed. This allows Spark to boost performance by optimizing *how* a sequence of transformations is executed at runtime.

One of the most commonly used transformations is the `map(func)`, which creates a new RDD by applying `func` to each element of the current RDD. This function, `func`, can be any callable python function, though it is often implemented as a `lambda` function. Similarly, `flatMap(func)` creates an RDD with the flattened results of `map(func)`.

```
# use map() to format the data
>>> titanic = spark.sparkContext.textFile('titanic.csv')
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# apply split(',') to each element of the RDD with map()
>>> titanic.map(lambda row: row.split(','))\
...         .take(2)
[['0', '3', 'Mr. Owen Harris Braund', 'male', '22', '1', '0', '7.25'],
 ['1', '1', ..., 'female', '38', '1', '0', '71.283']]

# compare to flatMap(), which flattens the results of each row
>>> titanic.flatMap(lambda row: row.split(','))\
...         .take(2)
['0', '3']
```

The `filter(func)` transformation returns a new RDD containing only the elements that satisfy `func`. In this case, `func` should be a callable python function that returns a Boolean. The elements of the RDD that evaluate to `True` are included in the new RDD while those that evaluate to `False` are excluded.

```
# create a new RDD containing only the female passengers
>>> titanic = titanic.map(lambda row: row.split(','))
>>> titanic_f = titanic.filter(lambda row: row[3] == 'female')
```

```
>>> titanic_f.take(3)
[['1', '1', ..., 'female', '38', '1', '0', '71.2833'],
 ['1', '3', ..., 'female', '26', '0', '0', '7.925'],
 ['1', '1', ..., 'female', '35', '1', '0', '53.1']]
```

Note

A great transformation to help validate or explore your dataset is `distinct()`. This will return a new RDD containing only the distinct elements of the original. In the case of the Titanic dataset, if you did not know how many classes there were, you could do the following:

```
>>> titanic.map(lambda row: row[1])\
...         .distinct()\
...         .collect()
['1', '3', '2']
```

Spark Command	Transformation
`map(f)`	Returns a new RDD by applying `f` to each element of this RDD
`flatmap(f)`	Same as `map(f)`, except the results are flattened
`filter(f)`	Returns a new RDD containing only the elements that satisfy `f`
`distinct()`	Returns a new RDD containing the distinct elements of the original
`reduceByKey(f)`	Takes an RDD of `(key, val)` pairs and merges the values for each `key` using an associative and commutative reduce function `f`
`sortBy(f)`	Sorts this RDD by the given function `f`
`sortByKey(f)`	Sorts an RDD assumed to consist of `(key, val)` pairs by the given function `f`
`groupBy(f)`	Returns a new RDD of groups of items based on `f`
`groupByKey()`	Takes an RDD of `(key, val)` pairs and returns a new RDD with `(key, (val1, val2, ...))` pairs

```
# the following counts the number of passengers in each class
# note that this isn't necessarily the best way to do this

# create a new RDD of (pclass, 1) elements to count occurances
>>> pclass = titanic.map(lambda row: (row[1], 1))
>>> pclass.take(5)
[('3', 1), ('1', 1), ('3', 1), ('1', 1), ('3', 1)]

# count the members of each class
>>> pclass = pclass.reduceByKey(lambda x, y: x + y)
>>> pclass.collect()
[('3', 487), ('1', 216), ('2', 184)]
```

```
# sort by number of passengers in each class, ascending order
>>> pclass.sortBy(lambda row: row[1]).collect()
[('2', 184), ('1', 216), ('3', 487)]
```

Problem 1. Write a function that accepts the name of a text file with default `filename=huck_finn.txt`.[a] Load the file as a PySpark RDD, and count the number of occurrences of each word. Sort the words by count, in descending order, and return a list of the `(word, count)` pairs for the 20 most used words. The data does not need to be cleaned.

[a] https://www.gutenberg.org/files/76/76-0.txt

Actions

Actions are operations that return non-RDD objects. Two of the most common actions, `take(n)` and `collect()`, have already been seen above. The key difference between the two is that `take(n)` returns the first `n` elements from one (or more) partition(s) while `collect()` returns the contents of the entire RDD. When working with small datasets this may not be an issue, but for larger datasets running `collect()` can be very expensive.

Another important action is `reduce(func)`. Generally, `reduce()` combines (reduces) the data in each row of the RDD using `func` to produce some useful output. Note that `func` *must* be an associative and commutative binary operation; otherwise the results will vary depending on partitioning.

```
# create an RDD with the first million integers in 4 partitions
>>> ints = spark.sparkContext.parallelize(range(1, 1000001), 4)
# [1, 2, 3, 4, 5, ..., 1000000]
# sum the first one million integers
>>> ints.reduce(lambda x, y: x + y)
500000500000

# create a new RDD containing only survival data
>>> survived = titanic.map(lambda row: int(row[0]))
>>> survived.take(5)
[0, 1, 1, 1, 0]

# find total number of survivors
>>> survived.reduce(lambda x, y: x + y)
500
```

Spark Command	Action
`take(n)`	returns the first `n` elements of an RDD
`collect()`	returns the entire contents of an RDD
`reduce(f)`	merges the values of an RDD using an associative and commutative operator `f`
`count()`	returns the number of elements in the RDD
`min(); max(); mean()`	returns the minimum, maximum, or mean of the RDD, respectively
`sum()`	adds the elements in the RDD and returns the result
`saveAsTextFile(path)`	saves the RDD as a collection of text files (one for each partition) in the directory specified
`foreach(f)`	immediately applies `f` to each element of the RDD; not to be confused with `map()`, `foreach()` is useful for saving data somewhere not natively supported by PySpark

Problem 2. Since the area of a circle of radius r is $A = \pi r^2$, one way to estimate π is to estimate the area of the unit circle. A Monte Carlo approach to this problem is to uniformly sample points in the square $[-1, 1] \times [-1, 1]$ and then count the percentage of points that land within the unit circle. The percentage of points within the circle approximates the percentage of the area occupied by the circle. Multiplying this percentage by 4 (the area of the square $[-1, 1] \times [-1, 1]$) gives an estimate for the area of the circle. [a]

Write a function that uses Monte Carlo methods to estimate the value of π. Your function should accept two keyword arguments: `n=10**5` and `parts=6`. Use `n*parts` sample points and partition your RDD with `parts` partitions. Return your estimate.

[a] See Example 7.1.1 in the Volume 2 textbook

DataFrames

While RDDs offer granular control, they can be slower than their Scala and Java counterparts when implemented in Python. The solution to this was the creation of a new data structure: Spark DataFrames. Just like RDDs, DataFrames are immutable distributed collections of objects; however, unlike RDDs, DataFrames are organized into named (and typed) columns. In this way they are conceptually similar to a relational database (or a pandas DataFrame).

The most important difference between a relational database and Spark DataFrames is in the execution of transformations and actions. When working with DataFrames, Spark's Catalyst Optimizer creates and optimizes a logical execution plan before sending any instructions to the drivers. After the logical plan has been formed, an optimal physical plan is created and executed. This provides significant performance boosts, especially when working with massive amounts of data. Since the Catalyst Optimizer functions the same across all language APIs, DataFrames bring performance parity to all of Spark's APIs.

Spark SQL and DataFrames

Creating a DataFrame from an existing text, csv, or JSON file is generally easier than creating an RDD. The DataFrame API also has arguments to deal with file headers or to automatically infer the

schema.

```
# load the titanic dataset using default settings
>>> titanic = spark.read.csv('titanic.csv')
>>> titanic.show(2)
+---+---+--------------------+------+---+---+---+-------+
|_c0|_c1|                 _c2|   _c3|_c4|_c5|_c6|    _c7|
+---+---+--------------------+------+---+---+---+-------+
|  0|  3|Mr. Owen Harris B...|  male| 22|  1|  0|   7.25|
|  1|  1|Mrs. John Bradley...|female| 38|  1|  0|71.2833|
+---+---+--------------------+------+---+---+---+-------+
only showing top 2 rows

# spark.read.csv('titanic.csv', inferSchema=True) will try to infer
# data types for each column

# load the titanic dataset specifying the schema
>>> schema = ('survived INT, pclass INT, name STRING, sex STRING, '
...           'age FLOAT, sibsp INT, parch INT, fare FLOAT'
...           )
>>> titanic = spark.read.csv('titanic.csv', schema=schema)
>>> titanic.show(2)
+--------+------+--------------------+------+---+-----+-----+-------+
|survived|pclass|                name|   sex|age|sibsp|parch|   fare|
+--------+------+--------------------+------+---+-----+-----+-------+
|       0|     3|Mr. Owen Harris B...|  male| 22|    1|    0|   7.25|
|       1|     1|Mrs. John Bradley...|female| 38|    1|    0|71.2833|
+--------+------+--------------------+------+---+-----+-----+-------+
only showing top 2 rows

# for files with headers, the following is convenient
spark.read.csv('my_file.csv', header=True, inferSchema=True)
```

> **Note**
>
> To convert a DataFrame to an RDD use `my_df.rdd`; to convert an RDD to a DataFrame use `spark.createDataFrame(my_rdd)`. You can also use `spark.createDataFrame()` on numpy arrays and pandas DataFrames.

DataFrames can be easily updated, queried, and analyzed using SQL operations. Spark allows you to run queries directly on DataFrames similar to how you perform transformations on RDDs. Additionally, the `pyspark.sql.functions` module contains many additional functions to further analysis. Below are many examples of basic DataFrame operations; further examples involving the `pyspark.sql.functions` module can be found in the additional materials section. Full documentation can be found at https://spark.apache.org/docs/latest/api/python/pyspark.sql.html.

```
# select data from the survived column
```

```
>>> titanic.select(titanic.survived).show(3)  # or titanic.select("survived")
+--------+
|survived|
+--------+
|       0|
|       1|
|       1|
+--------+
only showing top 3 rows

# find all distinct ages of passengers (great for data exploration)
>>> titanic.select("age")\
...         .distinct()\
...         .show(3)
+----+
| age|
+----+
|18.0|
|64.0|
|0.42|
+----+
only showing top 3 rows

# filter the DataFrame for passengers between 20-30 years old (inclusive)
>>> titanic.filter(titanic.age.between(20, 30)).show(3)
+--------+------+--------------------+------+----+-----+-----+------+
|survived|pclass|                name|   sex| age|sibsp|parch|  fare|
+--------+------+--------------------+------+----+-----+-----+------+
|       0|     3|Mr. Owen Harris B...|  male|22.0|    1|    0|  7.25|
|       1|     3|Miss. Laina Heikk...|female|26.0|    0|    0| 7.925|
|       0|     3|     Mr. James Moran|  male|27.0|    0|    0|8.4583|
+--------+------+--------------------+------+----+-----+-----+------+
only showing top 3 rows

# find total fare by pclass (or use .avg('fare') for an average)
>>> titanic.groupBy('pclass')\
...         .sum('fare')\
...         .show()
+------+---------+
|pclass|sum(fare)|
+------+---------+
|     1| 18177.41|
|     3|  6675.65|
|     2|  3801.84|
+------+---------+

# group and count by age and survival; order age/survival descending
>>> titanic.groupBy("age", "survived").count()\
...         .sort("age", "survived", ascending=False)\
```

```
...         .show(2)
+---+--------+-----+
|age|survived|count|
+---+--------+-----+
| 80|       1|    1|
| 74|       0|    1|
+---+--------+-----+
only showing top 2 rows

# join two DataFrames on a specified column (or list of columns)
>>> titanic_cabins.show(3)
+--------------------+-------+
|                name|  cabin|
+--------------------+-------+
|Miss. Elisabeth W...|     B5|
|Master. Hudsen Tr...|C22 C26|
|Miss. Helen Lorai...|C22 C26|
+--------------------+-------+
only showing top 3 rows

>>> titanic.join(titanic_cabins, on='name').show(3)
+--------------------+--------+------+------+----+-----+-----+------+-------+
|                name|survived|pclass|   sex| age|sibsp|parch|  fare|  cabin|
+--------------------+--------+------+------+----+-----+-----+------+-------+
|Miss. Elisabeth W...|       0|     3|  male|22.0|    1|    0|  7.25|     B5|
|Master. Hudsen Tr...|       1|     3|female|26.0|    0|    0| 7.925|C22 C26|
|Miss. Helen Lorai...|       0|     3|  male|27.0|    0|    0|8.4583|C22 C26|
+--------------------+--------+------+------+----+-----+-----+------+-------+
only showing top 3 rows
```

Note

If you prefer to use traditional SQL syntax you can use `spark.sql("SQL QUERY")`. Note that this requires you to first create a temporary view of the DataFrame.

```
# create the temporary view so we can access the table through SQL
>>> titanic.createOrReplaceTempView("titanic")

# query using SQL syntax
>>> spark.sql("SELECT age, COUNT(*) AS count\
...             FROM titanic\
...             GROUP BY age\
...             ORDER BY age DESC").show(3)
+---+-----+
|age|count|
+---+-----+
```

```
| 80|    1|
| 74|    1|
| 71|    2|
+---+-----+
only showing top 3 rows
```

Spark SQL Command	SQLite Command
`select(*cols)`	`SELECT`
`groupBy(*cols)`	`GROUP BY`
`sort(*cols, **kwargs)`	`ORDER BY`
`filter(condition)`	`WHERE`
`when(condition, value)`	`WHEN`
`between(lowerBound, upperBound)`	`BETWEEN`
`drop(*cols)`	`DROP`
`join(other, on=None, how=None)`	`JOIN` (join type specified by `how`)
`count()`	`COUNT()`
`sum(*cols)`	`SUM()`
`avg(*cols)` or `mean(*cols)`	`AVG()`
`collect()`	`fetchall()`

Problem 3. Write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame and find (1) the number of women on-board, (2) the number of men on-board, (3) the survival rate of women, and (4) the survival rate of men. Return these four values in the order given.

Problem 4. In this problem, you will be using the `london_income_by_borough.csv` and the `london_crime_by_lsoa.csv` files to visualize the relationship between income and the frequency of crime.[a] The former contains estimated mean and median income data for each London borough, averaged over 2008-2016; the first line of the file is a header with columns `borough`, `mean-08-16`, and `median-08-16`. The latter contains over 13 million lines of crime data, organized by borough and LSOA (Lower Super Output Area) code, for London between 2008 and 2016; the first line of the file is a header, containing the following seven columns:

`lsoa_code`: LSOA code (think area code) where the crime was committed
`borough`: London borough were the crime was committed
`major_category`: major (read: general) category of the crime
`minor_category`: minor (read: specific) category of the crime
`value`: number of occurrences of this crime in the given `lsoa_code`, `month`, and `year`
`year`: year the crime was committed
`month`: month the crime was committed

Write a function that accepts three keyword arguments: `crimefile='london_crime_by_lsoa.csv'`, `incomefile='london_income_by_borough.csv'`, and `major_cat='Robbery'`. Load the two files as PySpark DataFrames. Use them to create a new DataFrame containing, for each borough, the total number of crimes for the given major_category (`major_cat`) and the median income. Order the DataFrame by the average crime rate, descending. The final DataFrame should have three columns: `borough`, `average_crime_rate`, `median-08-16` (column names may be different).

Convert the DataFrame to a numpy array using `np.array(df.collect())`, and create a scatter plot of the number of murders by the median income for each borough. Return the numpy array.

[a]data.london.gov.uk

Machine Learning with Apache Spark

Apache Spark includes a vast and expanding ecosystem to perform machine learning. PySpark's primary machine learning API, `pyspark.ml`, is DataFrame-based.

Here we give a start to finish example using Spark ML to tackle the classic Titanic classification problem.

```
# prepare data
# convert the 'sex' column to binary categorical variable
>>> from pyspark.ml.feature import StringIndexer, OneHotEncoder
>>> sex_binary = StringIndexer(inputCol='sex', outputCol='sex_binary')

# one-hot-encode pclass (Spark automatically drops a column)
>>> onehot = OneHotEncoder(inputCols=['pclass'],
...                                 outputCols=['pclass_onehot'])

# create single features column
from pyspark.ml.feature import VectorAssembler
features = ['sex_binary', 'pclass_onehot', 'age', 'sibsp', 'parch', 'fare']
features_col = VectorAssembler(inputCols=features, outputCol='features')

# now we create a transformation pipeline to apply the operations above
# this is very similar to the pipeline ecosystem in sklearn
>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=[sex_binary, onehot, features_col])
>>> titanic = pipeline.fit(titanic).transform(titanic)

# drop unnecessary columns for cleaner display (note the new columns)
>>> titanic = titanic.drop('pclass', 'name', 'sex')
>>> titanic.show(2)
+--------+----+-----+-----+----+----------+-------------+----------+
|survived| age|sibsp|parch|fare|sex_binary|pclass_onehot|  features|
+--------+----+-----+-----+----+----------+-------------+----------+
|       0|22.0|    1|    0|7.25|       0.0|    (3,[],[])|(8,[4,5...|
|       1|38.0|    1|    0|71.3|       1.0|   (3,[1],...|[0.0,1....|
```

```
+--------+----+-----+-----+----+----------+-------------+----------+

# split into train/test sets (75/25)
>>> train, test = titanic.randomSplit([0.75, 0.25], seed=11)

# initialize logistic regression
>>> from pyspark.ml.classification import LogisticRegression
>>> lr = LogisticRegression(labelCol='survived', featuresCol='features')

# run a train-validation-split to fit best elastic net param
# ParamGridBuilder constructs a grid of parameters to search over.
>>> from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator as MCE
>>> paramGrid = ParamGridBuilder()\
...                 .addGrid(lr.elasticNetParam, [0, 0.5, 1]).build()
# TrainValidationSplit will try all combinations and determine best model using
# the evaluator (see also CrossValidator)
>>> tvs = TrainValidationSplit(estimator=lr,
...                            estimatorParamMaps=paramGrid,
...                            evaluator=MCE(labelCol='survived'),
...                            trainRatio=0.75,
...                            seed=11)

# we train the classifier by fitting our tvs object to the training data
>>> clf = tvs.fit(train)

# use the best fit model to evaluate the test data
>>> results = clf.bestModel.evaluate(test)
>>> results.predictions.select(['survived', 'prediction']).show(5)
+--------+----------+
|survived|prediction|
+--------+----------+
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       0.0|
+--------+----------+

# performance information is stored in various attributes of "results"
>>> results.accuracy
0.7527272727272727

>>> results.weightedRecall
0.7527272727272727

>>> results.weightedPrecision
0.751035147726004
```

```
# many classifiers do not have this object-oriented interface (yet)
# it isn't much more effort to generate the same statistics for a ↩
    DecisionTreeClassifier, for example
>>> dt_clf = dt_tvs.fit(train) # same process, except for a different paramGrid

# generate predictions - this returns a new DataFrame
>>> preds = clf.bestModel.transform(test)
>>> preds.select('survived', 'probability', 'prediction').show(5)
+--------+-----------+----------+
|survived|probability|prediction|
+--------+-----------+----------+
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [0.0,1.0]|       1.0|
+--------+-----------+----------+

# initialize evaluator object
>>> dt_eval = MCE(labelCol='survived')
>>> dt_eval.evaluate(preds, {dt_eval.metricName: 'accuracy'})
0.8433179723502304
```

Below is a broad overview of the `pyspark.ml` ecosystem. It should help give you a starting point when looking for a specific functionality.

PySpark ML Module	Module Purpose
`pyspark.ml.feature`	provides functions to transform data into feature vectors
`pyspark.ml.tuning`	grid search, cross validation, and train/validation split functions
`pyspark.ml.evaluation`	tools to compute prediction metrics (accuracy, f1, etc.)
`pyspark.ml.classification`	classification models (logistic regression, SVM, etc.)
`pyspark.ml.clustering`	clustering models (k-means, Gaussian mixture, etc.)
`pyspark.ml.regression`	regression models (linear regression, decision tree regressor, etc.)

Problem 5. Write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame, and use the `pyspark.ml` package to train a classifier that outperforms the logistic regression each of the three metrics from the example above (`accuracy`, `weightedRecall`, `weightedPrecision`).

Some of Spark's available classifiers are listed below. For complete documentation, visit https://spark.apache.org/docs/latest/api/python/pyspark.ml.html.

```
# from pyspark.ml.classification import LinearSVC
#                              DecisionTreeClassifier
#                              GBTClassifier
#                              MultilayerPerceptronClassifier
#                              NaiveBayes
#                              RandomForestClassifier
```

Use `randomSplit([0.75, 0.25], seed=11)` to split your data into train and test sets before fitting the model. Return the `accuracy`, `weightedRecall`, and `weightedPrecision` for your model, in the given order.

Hint: to calculate the accuracy of a classifer in PySpark, use `accuracy = MCE(labelCol ='survived',metricName='accuracy').evaluate(predictions)`.

Additional Material

Further DataFrame Operations

There are a few other functions built directly on top of DataFrames to further analysis. Additionally, the `pyspark.sql.functions` module expands the available functions significantly.[4]

```
# some immediately accessible functions
# covariance between pclass and fare
>>> titanic.cov('pclass', 'fare')
-22.86289824115662

# summary of statistics for selected columns
>>> titanic.select("pclass", "age", "fare")\
...         .summary().show()
+-------+------------------+------------------+------------------+
|summary|            pclass|               age|              fare|
+-------+------------------+------------------+------------------+
|  count|               887|               887|               887|
|   mean| 2.305524239007892|29.471443066501564|32.305420253026846|
| stddev|0.8366620036697728|14.121908405492908| 49.78204096767521|
|    min|                 1|              0.42|               0.0|
|    25%|                 2|              20.0|             7.925|
|    50%|                 3|              28.0|           14.4542|
|    75%|                 3|              38.0|            31.275|
|    max|                 3|              80.0|          512.3292|
+-------+------------------+------------------+------------------+

# additional functions from the functions module
>>> from pyspark.sql import functions as sqlf

# finding the mean of a column without grouping requires sqlf.avg()
# alias(new_name) allows us to rename the column on the fly
>>> titanic.select(sqlf.avg("age").alias("Average Age")).show()
+------------------+
|       Average Age|
+------------------+
|29.471443066516347|
+------------------+

# use .agg([dict]) on GroupedData to specify [multiple] aggregate
# functions, including those from pyspark.sql.functions
>>> titanic.groupBy('pclass')\
...         .agg({'fare': 'var_samp', 'age': 'stddev'})\
...         .show(3)
+------+------------------+------------------+
|pclass|    var_samp(fare)|       stddev(age)|
+------+------------------+------------------+
```

[4] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

```
|     1| 6143.483042924841|14.183632587264817|
|     3|139.64879027298073|12.095083834183779|
|     2|180.02658999396826|13.756191206499766|
+------+------------------+------------------+

# perform multiple aggregate actions on the same column
>>> titanic.groupBy('pclass')\
...         .agg(sqlf.sum('fare'), sqlf.stddev('fare'))\
...         .show()
+------+------------------+------------------+
|pclass|         sum(fare)| stddev_samp(fare)|
+------+------------------+------------------+
|     1|18177.412506103516| 78.38037409278448|
|     3| 6675.653553009033| 11.81730892686574|
|     2|3801.8417053222656|13.417398778972332|
+------+------------------+------------------+
```

pyspark.sql.functions	**Operation**
`ceil(col)`	computes the ceiling of each element in `col`
`floor(col)`	computes the floor of each element in `col`
`min(col)`, `max(col)`	returns the minimum/maximum value of `col`
`mean(col)`	returns the average of the values of `col`
`stddev(col)`	returns the unbiased sample standard deviation of `col`
`var_samp(col)`	returns the unbiased variance of the values in `col`
`rand(seed=None)`	generates a random column with i.i.d. samples from $[0, 1]$
`randn(seed=None)`	generates a random column with i.i.d. samples from the standard normal distribution
`exp(col)`	computes the exponential of `col`
`log(arg1, arg2=None)`	returns `arg1`-based logarithm of `arg2`; if there is only one argument, then it returns the natural logarithm
`cos(col)`, `sin(col)`, etc.	computes the given trigonometric or inverse trigonometric (`asin(col)`, etc.) function of `col`

3 Web Scraping

Lab Objective: *Web Scraping is the process of gathering data from websites on the internet. Since almost everything rendered by an internet browser as a web page uses HTML, the first step in web scraping is being able to extract information from HTML. In this lab, we introduce the requests library for scraping web pages, and BeautifulSoup, Python's canonical tool for efficiently and cleanly navigating and parsing HTML.*

HTTP and Requests

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP, which we used to build a server in the Web Technologies lab, but uses TCP protocols to manage connections and provide network capabilities. The HTTP protocol is centered around a request and response paradigm, in which a client makes a request to a server and the server replies with a response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. A GET request asks for information from the server, a POST request modifies the state of the server, and a PUT request adds new pieces of data to the server.

The standard way to get the source code of a website using Python is via the `requests` library.[1] Calling `requests.get()` sends an HTTP GET request to a specified website. The website returns a response code, which indicates whether or not the request was received, understood, and accepted. If the response code is good, typically `200`[2], then the response will also include the website source code as an HTML file.

```
>>> import requests

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

[1] Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See `http://docs.python-requests.org/`.

[2] See `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes` for explanation of specific response codes.

```
# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!DOCTYPE html>
<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/↩
    content/  dc: http://purl.org/dc/terms/  foaf: http://xmlns.com/foaf/0.1/  ↩
    og: http://ogp.me/ns#  rdfs: http://www.w3.org/2000/01/rdf-schema#  schema:↩
     http://schema.org/  sioc: http://rdfs.org/sioc/ns#  sioct: http://rdfs.org↩
    /sioc/types#  skos: http://www.w3.org/2004/02/skos/core#  xsd: http://www.↩
    w3.org/2001/XMLSchema# " class=" ">

  <head>
    <meta charset="utf-8" />
# ...
```

Note that some websites aren't built to handle large amounts of traffic or many repeated requests. Most are built to identify web scrapers or crawlers that initiate many consecutive GET requests without pauses, and retaliate or block them. When web scraping, always make sure to store the data that you receive in a file and include error checks to prevent retrieving the same data unnecessarily. We won't spend much time on that in this lab, but it's especially important in larger applications.

Problem 1. Use the `requests` library to get the HTML source for the website `http://www.example.com`. Save the source as a file called example.html. If the file already exists, make sure not to scrape the website or overwrite the file. You will use this file later in the lab.

Achtung!

Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.[a]

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`.[b]

We will cover this more in the next lab.

[a] See `www.robotstxt.org/orig.html` and `en.wikipedia.org/wiki/Robots_exclusion_standard`.

[b] Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see `https://docs.python.org/3/library/urllib.robotparser.html`.

HTML

Hyper Text Markup Language, or *HTML*, is the standard *markup language*—a language designed for the processing, definition, and presentation of text—for creating webpages. It structures a document

using pairs of *tags* that surround and define content. Opening tags have a tag name surrounded by angle brackets (`<tag-name>`). The companion closing tag looks the same, but with a forward slash before the tag name (`</tag-name>`). A list of all current HTML tags can be found at `http://htmldog.com/reference/htmltags`.

Most tags can be combined with *attributes* to include more data about the content, help identify individual tags, and make navigating the document much simpler. In the following example, the `<a>` tag has `id` and `href` attributes.

```
<html>                                  <!-- Opening tags -->
   <body>
       <p>
           Click <a id='info' href='http://www.example.com'>here</a>
           for more information.
       </p>                             <!-- Closing tags -->
   </body>
</html>
```

In HTML, `href` stands for *hypertext reference*, a link to another website. Thus the above example would be rendered by a browser as a single line of text, with `here` being a clickable link to `http://www.example.com`:

Click here for more information.

Unlike Python, HTML does not enforce indentation (or any whitespace rules), though indentation generally makes HTML more readable. The previous example can be written in a single line.

```
<html><body><p>Click <a id='info' href='http://www.example.com/info'>here</a>
  for more information.</p></body></html>
```

Special tags, which don't contain any text or other tags, are written without a closing tag and in a single pair of brackets. A forward slash is included between the name and the closing bracket. Examples of these include `<hr/>`, which describes a horizontal line, and `<img/>`, the tag for representing an image.

> **Note**
>
> You can open .html files using a text editor or any web browser. In a browser, you can inspect the source code associated with specific elements. Right click the element and select `Inspect`. If you are using Safari, you may first need to enable "Show Develop menu" in "Preferences" under the "Advanced" tab.

BeautifulSoup

BeautifulSoup (`bs4`) is a package[3] that makes it simple to navigate and extract data from HTML documents. See `http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html` for the full documentation.

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

The `bs4.BeautifulSoup` class accepts two parameters to its constructor: a string of HTML code and an HTML parser to use under the hood. The HTML parser is technically a keyword argument, but the constructor prints a warning if one is not specified. The standard choice for the parser is `"html.parser"`, which means the object uses the standard library's `html.parser` module as the engine behind the scenes.

> **Note**
>
> Depending on project demands, a parser other than `"html.parser"` may be useful. A couple of other options are `"lxml"`, an extremely fast parser written in C, and `"html5lib"`, a slower parser that treats HTML in much the same way a web browser does, allowing for irregularities. Both must be installed independently; see `https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser` for more information.

A `BeautifulSoup` object represents an HTML document as a tree. In the tree, each tag is a *node* with nested tags and strings as its *children*. The `prettify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format that reflects the tree structure.

```
>>> from bs4 import BeautifulSoup

>>> small_example_html = """
<html><body><p>
    Click <a id='info' href='http://www.example.com'>here</a>
    for more information.
</p></body></html>
"""

>>> small_soup = BeautifulSoup(small_example_html, 'html.parser')
>>> print(small_soup.prettify())
<html>
  <body>
    <p>
      Click
      <a href="http://www.example.com" id="info">
         here
      </a>
      for more information.
    </p>
  </body>
</html>
```

Each tag in a `BeautifulSoup` object's HTML code is stored as a `bs4.element.Tag` object, with actual text stored as a `bs4.element.NavigableString` object. Tags are accessible directly through the `BeautifulSoup` object.

```
# Get the <p> tag (and everything inside of it).
>>> small_soup.p
```

```
<p>
    Click <a href="http://www.example.com" id="info">here</a>
    for more information.
</p>

# Get the <a> sub-tag of the <p> tag.
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='\n')
<a href="http://www.example.com" id="info">here</a>
<class 'bs4.element.Tag'>

# Get just the name, attributes, and text of the <a> tag.
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a
{'id': 'info', 'href': 'http://www.example.com'}
here
```

Attribute	Description
`name`	The name of the tag
`attrs`	A dictionary of the attributes
`string`	The single string contained in the tag
`strings`	Generator for strings of children tags
`stripped_strings`	Generator for strings of children tags, stripping whitespace
`text`	Concatenation of strings from all children tags

Table 3.1: Data attributes of the `bs4.element.Tag` class.

Problem 2. The `BeautifulSoup` class has a `find_all()` method that, when called with `True` as the only argument, returns a list of all tags in the HTML source code.

Write a function that accepts a string of HTML code as an argument. Use BeautifulSoup to return a list of the **names** of the tags in the code.

Navigating the Tree Structure

Not all tags are easily accessible from a `BeautifulSoup` object. Consider the following example.

```
>>> pig_html = """
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title"><b>The Three Little Pigs</b></p>
<p class="story">Once upon a time, there were three little pigs named
<a href="http://example.com/larry" class="pig" id="link1">Larry,</a>
<a href="http://example.com/mo" class="pig" id="link2">Mo</a>, and
<a href="http://example.com/curly" class="pig" id="link3">Curly.</a>
<p>The three pigs had an odd fascination with experimental construction.</p>
```

```
<p>...</p>
</body></html>
"""

>>> pig_soup = BeautifulSoup(pig_html, "html.parser")
>>> pig_soup.p
<p class="title"><b>The Three Little Pigs</b></p>

>>> pig_soup.a
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>
```

Since the HTML in this example has several `<p>` and `<a>` tags, only the **first** tag of each name is accessible directly from `pig_soup`. The other tags can be accessed by manually navigating through the HTML tree.

Every HTML tag (except for the topmost tag, which is usually `<html>`) has a *parent* tag. Each tag also has zero or more *sibling* and *children* tags or text. Following a true tree structure, every `bs4.element.Tag` in a soup has multiple attributes for accessing or iterating through parent, sibling, or child tags.

Attribute	Description
`parent`	The parent tag
`parents`	Generator for the parent tags up to the top level
`next_sibling`	The tag immediately after to the current tag
`next_siblings`	Generator for sibling tags after the current tag
`previous_sibling`	The tag immediately before the current tag
`previous_siblings`	Generator for sibling tags before the current tag
`contents`	A list of the immediate children tags
`children`	Generator for immediate children tags
`descendants`	Generator for all children tags (recursively)

Table 3.2: Navigation attributes of the `bs4.element.Tag` class.

```
# Start at the first <a> tag in the soup.
>>> a_tag = pig_soup.a
>>> a_tag
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

# Get the names of all of <a>'s parent tags, traveling up to the top.
# The name '[document]' means it is the top of the HTML code.
>>> [par.name for par in a_tag.parents]     # <a>'s parent is <p>, whose
['p', 'body', 'html', '[document]']         # parent is <body>, and so on.

# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'                                        # The first sibling is just text.
>>> a_tag.next_sibling.next_sibling         # The second sibling is a tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

Note carefully that newline characters are considered to be children of a parent tag. Therefore iterating through children or siblings often requires checking which entries are tags and which are just text. In the next example, we use a tag's `attrs` attribute to access specific attributes within the tag (see Table 3.1).

```
# Get to the <p> tag that has class="story" using these commands.
>>> p_tag = pig_soup.body.p.next_sibling.next_sibling
>>> p_tag.attrs["class"]                # Make sure it's the right tag.
['story']

# Iterate through the child tags of <p> and print hrefs whenever they exist.
>>> for child in p_tag.children:
...     # Skip the children that are not bs4.element.Tag objects
...     # These don't have the attribute "attrs"
...     if hasattr(child, "attrs") and "href" in child.attrs:
...         print(child.attrs["href"])
http://example.com/larry
http://example.com/mo
http://example.com/curly
```

Note that the `"class"` attribute of the `<p>` tag is a list. This is because the `"class"` attribute can take on several values at once; for example, the tag `<p class="story book">` is of class `'story'` and of class `'book'`.

The behavior of the `string` attribute of a `bs4.element.Tag` object depends on the structure of the corresponding HTML tag.

1. If the tag has a string of text and no other child elements, then `string` is just that text.

2. If the tag has exactly one child tag and the child tag has only a string of text, then the tag has the same `string` as its child tag.

3. If the tag has more than one child, then `string` is `None`. In this case, use `strings` to iterate through the child strings. Alternatively, the `get_text()` method returns all text belonging to a tag and to all of its descendants. In other words, it returns anything inside a tag that isn't another tag.

```
>>> pig_soup.head
<head><title>Three Little Pigs</title></head>

# Case 1: the <title> tag's only child is a string.
>>> pig_soup.head.title.string
'Three Little Pigs'

# Case 2: The <head> tag's only child is the <title> tag.
>>> pig_soup.head.string
'Three Little Pigs'

# Case 3: the <body> tag has several children.
>>> pig_soup.body.string is None
```

```
True
>>> print(pig_soup.body.get_text().strip())
The Three Little Pigs
Once upon a time, there were three little pigs named
Larry,
Mo, and
Curly.
The three pigs had an odd fascination with experimental construction.
...
```

Problem 3. Write a function that reads a file of the same format as the output from Problem 1 and loads it into BeautifulSoup. Find the first `<a>` tag, and return its text along with a boolean value indicating whether or not it has a hyperlink (`href` attribute).

Searching for Tags

Navigating the HTML tree manually can be helpful for gathering data out of lists or tables, but these kinds of structures are usually buried deep in the tree. The `find()` and `find_all()` methods of the `BeautifulSoup` class identify tags that have distinctive characteristics, making it much easier to jump straight to a desired location in the HTML code. The `find()` method only returns the **first** tag that matches a given criteria, while `find_all()` returns a list of all matching tags. Tags can be matched by name, attributes, and/or text.

```
# Find the first <b> tag in the soup.
>>> pig_soup.find(name='b')
<b>The Three Little Pigs</b>

# Find all tags with a class attribute of 'pig'.
# Since 'class' is a Python keyword, use 'class_' as the argument.
>>> pig_soup.find_all(class_="pig")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the first tag that matches several attributes.
>>> pig_soup.find(attrs={"class": "pig", "href": "http://example.com/mo"})
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

# Find the first tag whose text is 'Mo'.
>>> pig_soup.find(string='Mo')
'Mo'                                    # The result is the actual string,
>>> pig_soup.find(string='Mo').parent   # so go up one level to get the tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

Problem 4. The file `san_diego_weather.html` contains the HTML source for an old page from Weather Underground.[a] Write a function that reads the file and loads it into BeautifulSoup.

Return a list of the following tags:

1. The tag containing the date "Thursday, January 1, 2015".
2. The tags which contain the **links** "Previous Day" and "Next Day."
3. The tag which contains the number associated with the Actual Max Temperature.

[a] See `http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1`

Advanced Search Techniques: Regular Expressions

Consider the problem of finding the tag that is a link to the URL `http://example.com/curly`.

```
>>> pig_soup.find(href="http://example.com/curly")
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>
```

This approach works, but it requires entering in the entire URL. To perform generalized searches, the `find()` and `find_all()` method also accept compiled regular expressions from the `re` module. This way, the methods locate tags whose name, attributes, and/or string matches a pattern.

```
>>> import re

# Find the first tag with an href attribute containing 'curly'.
>>> pig_soup.find(href=re.compile(r"curly"))
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a

# Find the first tag with a string that starts with 'Cu'.
>>> pig_soup.find(string=re.compile(r"^Cu")).parent
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find all tags with text containing 'Three'.
>>> [tag.parent for tag in pig_soup.find_all(string=re.compile(r"Three"))]
[<title>Three Little Pigs</title>, <b>The Three Little Pigs</b>]
```

Finally, to find a tag that has a particular attribute, regardless of the actual value of the attribute, use `True` in place of search values.

```
# Find all tags with an 'id' attribute.
>>> pig_soup.find_all(id=True)
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

```
# Find the names all tags WITHOUT an 'id' attribute.
>>> [tag.name for tag in pig_soup.find_all(id=False)]
['html', 'head', 'title', 'body', 'p', 'b', 'p', 'p', 'p']
```

Advanced Search Techniques: CSS Selectors

`BeautifulSoup` also supports the use of CSS selectors. CSS (Cascading Style Sheet) describes the style and layout of a webpage, and CSS selectors provide a useful way to navigate HTML code. Use the method `soup.select()` to find all elements matching an argument. The general format for an argument is `tag-name[attribute-name = 'attribute value']`. The table below lists symbols you can use to more precisely locate various elements.

Symbol	Meaning
=	Matches an attribute value exactly
*=	Partially matches an attribute value
^=	Matches the beginning of an attribute value
$=	Matches the end of an attribute value
+	Next sibling of matching element
>	Search an element's children

Table 3.3: CSS symbols for use with Selenium

You can do many other useful things with CSS selectors. A helpful guide can be found at `https://www.w3schools.com/cssref/css_selectors.asp`. The code below gives an example using arguments described above.

```
# Find all <a> tags with id="link1"
>>> pig_soup.select("[id='link1']")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>]

# Find all tags with an href attribute containing 'curly'.
>>> pig_soup.select("[href*='curly']")
[<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <a> tags with an href attribute
>>> pig_soup.select("a[href]")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

 # Find all <b> tags within a <p> tag with class='title'
 >>> pig_soup.select("p[class='title'] b")
 [<b>The Three Little Pigs</b>]

 # Use a comma to find elements matching one of two arguments
 >>> pig_soup.select("a[href$='mo'],[id='link3']")
[<a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

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

[a] See `https://www.federalreserve.gov/releases/lbr/`.

Problem 6. The file `large_banks_data.html` is one of the pages from the index in Problem 5.[a] Write a function that reads the file and loads the source into BeautifulSoup. Create a single figure with two subplots:

1. A sorted bar chart of the seven banks with the most domestic branches.
2. A sorted bar chart of the seven banks with the most foreign branches.

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

[a] See `http://www.federalreserve.gov/releases/lbr/20030930/default.htm`.

4 Web Crawling

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

Scraping Etiquette

There are two main ways that web scraping can be problematic for a website owner.

1. The scraper doesn't respect the website's terms and conditions or gathers private or proprietary data.

2. The scraper imposes too much extra server load by making requests too often or in quick succession.

These are extremely important considerations in any web scraping program. Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits, and how often requests can be made according to the *robots exclusion standard*.[1]

> **Achtung!**
>
> Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`. Make sure to scrape websites legally.

Recall that consecutive requests without pauses can strain a website's server and provoke retaliation. Most servers are designed to identify such scrapers, block their access, and sometimes even

[1] See `www.robotstxt.org/orig.html` and `en.wikipedia.org/wiki/Robots_exclusion_standard`.

blacklist the user. This is especially common in smaller websites that aren't built to handle enormous amounts of traffic. To briefly pause the program between requests, use `time.sleep()`.

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

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

Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. Below is an example of using this library to check where robots are allowed on arxiv.org. A website's `robots.txt` file will often include different instructions for specific crawlers. These crawlers are identified by a `User-agent` string. For example, Google's webcrawler, `User-agent` Googlebot, may be directed to index only the pages the website wants to have listed on a Google search. We will use the default `User-agent`, `"*"`.

```
>>> from urllib import robotparser
>>> rp = robotparser.RobotFileParser()
# Set the URL for the robots.txt file. Note that the URL contains `robots.txt'
>>> rp.set_url("https://arxiv.org/robots.txt")
>>> rp.read()
# Request the crawl-delay time for the default User-agent
>>> rp.crawl_delay("*")
15
# Check if User-agent "*" can access the page
>>> rp.can_fetch("*", "https://arxiv.org/archive/math/")
True
>>> rp.can_fetch("*", "https://arxiv.org/IgnoreMe/")
False
```

Problem 1. Write a program that accepts a web address defaulting to the site `http://example.webscraping.com` and a list of pages defaulting to `["/", "/trap", "/places/default/search"]`. For each page, check if the `robots.txt` file permits access. Return a list of boolean values corresponding to each page. Also return the crawl delay time.

Crawling Through Multiple Pages

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

Consider `books.toscrape.com`, a site to practice web scraping that mimics a bookstore. The page `http://books.toscrape.com/catalogue/category/books/mystery_3/index.html` lists mystery books with overall ratings and review. More mystery books can be accessed by clicking on the

`next` link. The following example demonstrates how to navigate between webpages to collect all of the mystery book titles.

```
def scrape_books(start_page = "index.html"):
    """ Crawl through http://books.toscrape.com and extract mystery titles"""

    # Initialize variables, including a regex for finding the 'next' link.
    base_url="http://books.toscrape.com/catalogue/category/books/mystery_3/"
    titles = []
    page = base_url + start_page                # Complete page URL.
    next_page_finder = re.compile(r"next")      # We need this button.

    current = None

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

        # Navigate to the correct tag and extract title
        for book in current:
            titles.append(book.h3.a["title"])

        # Find the URL for the page with the next data.
        if "page-4" not in page:
            new_page = soup.find(string=next_page_finder).parent["href"]
            page = base_url + new_page      # New complete page URL.
            current = None
    return titles
```

In this example, the `for` loop cycles through the pages of books, and the `while` loop ensures that each website page loads properly: if the downloaded `page_source` doesn't have a tag whose class is `product_pod`, the request is sent again. After recording all of the titles, the function locates the link to the next page. This link is stored in the HTML as a relative website path (`page-2.html`); the complete URL to the next day's page is the concatenation of the base URL `http://books.toscrape.com/catalogue/category/books/mystery_3/` with this relative link.

Problem 2. Modify `scrape_books()` so that it gathers the price for each fiction book and returns the mean price, in £, of a fiction book.

Asynchronously Loaded Content and User Interaction

Web crawling with the methods presented in the previous section fails under a few circumstances. First, many webpages use *JavaScript*, the standard client-side scripting language for the web, to

load portions of their content *asynchronously*. This means that at least some of the content isn't initially accessible through the page's source code (for example, if you have to scroll down to load more results). Second, some pages require user interaction, such as clicking buttons which aren't links (`<a>` tags which contain a URL that can be loaded) or entering text into form fields (like search bars).

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

> **Note**
>
> Selenium requires an executable *driver* file for each kind of browser. The following examples use Google Chrome, but Selenium supports Firefox, Internet Explorer, Safari, Opera, and PhantomJS (a special browser without a user interface). See `https://seleniumhq.github.io/selenium/docs/api/py` or `http://selenium-python.readthedocs.io/installation.html` for installation instructions and driver download instructions.
>
> If your program still can't find the driver after you've downloaded it, add the argument `executable_path = "path/to/driver/file"` when you call `webdriver`. If this doesn't work, you may need to add the location to your system PATH. On a Mac, open the file `/etc/path` and add the new location. On Linux, add `export PATH="path/to/driver/file:$PATH"` to the file `/.bashrc` . For Windows, follow a tutorial such as this one: `https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/`.

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

```
>>> from selenium import webdriver

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

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

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

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

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

Table 4.1: Methods of the `selenium.webdriver.Chrome` class.

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

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

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

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

Problem 3. The website IMDB contains a variety of information on movies. Specifically, information on the top 10 box office movies of the week can be found at `https://www.imdb.com/chart/boxoffice`. Using BeaufiulSoup, Selenium, or both, return a list, with each title on a new row, of the top 10 movies of the week and order the list according to the total grossing of the movies, from most money to the least. Break ties using the weekend gross, from most money to the least.

Using CSS Selectors

In addition to the methods listed in Table 4.1, you can also use CSS or XPath selectors to interact more precisely with the page source. Refer to Table 3 from the WebScraping lab for a review of CSS syntax. The following code searches Google for "Python Selenium Docs" and then clicks on the second result.

```
#As before, go to Google and type in the search bar,
# but this time we use CSS selectors

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

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://google.com")
...     try:
...         search_bar = browser.find_element_by_css_selector(
                        "input[name='q']")
...         search_bar.clear()
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN)
...         try:
...             # Wait a second, then get the second search result
...             time.sleep(1)
...             # "+ div" returns the element's  next sibling with a "div" tag
...             second_result = browser.find_element_by_css_selector(
                                "div[class='g'] + div")
...             try:
...                 # Get the link, which is a child of second_result
...                 link = second_result.find_element_by_css_selector(
                            "div[class='r']")
...                 link.click()
...                 time.sleep(1)

...             #Remember to handle exceptions
...             except NoSuchElementException:
...                 print("Could not find link")
...         except NoSuchElementException:
...             print("Could not find second result")
```

```
...     except NoSuchElementException:
...         print("Could not find the search bar")
... finally:
...     browser.close()
```

In the above example, we could have used `find_element_by_class_name()`, but when you need more precision than that, CSS selectors can be very useful. Remember that to view specific HTML associated with an object in Chrome or Firefox, you can right click on the object and click "Inspect." For Safari, you need to first enable "Show Develop menu" in "Preferences" under "Advanced." Keep in mind that you can also search through the source code (ctrl+f or cmd+f) to make sure you're using a unique identifier.

> **Note**
>
> Using Selenium to access a page's source code is typically much safer, though slower, than using `requests.get()`, since Selenium waits for each web page to load before proceeding. For instance, some websites are somewhat defensive about scrapers, but Selenium can sometimes make it possible to gather info without offending the administrators.

Problem 4. *Project Euler* (`https://projecteuler.net`) is a collection of mathematical computing problems. Each problem is listed with an ID, a description/title, and the number of users that have solved the problem.

Using Selenium, BeautifulSoup, or both, record the number of people who have solved each of the 700+ problems in the archive at https://projecteuler.net/archives. Plot the number of people who have solved each problem against the problem IDs, using a log scale for the y-axis. Display the scatter plot, then state the IDs of which problems have been solved most and least number of times.

Problem 5. The website `http://example.webscraping.com` contains a list of countries of the world. Using Selenium, go to the search page, enter the letters `"ca"`, and hit `enter`. Remember to use the crawl delay time you found in Problem 1 so you don't send your requests too fast. Gather the `href` links associated with the `<a>` tags of all 10 displayed results. Print each link on a different line.

5 Parallel Programming with MPI

Lab Objective: *In the world of parallel computing, MPI is the most widespread and standardized message passing library. As such, it is used in the majority of parallel computing programs. In this lab, we explore and practice the basic principles and commands of MPI to further recognize when and how parallelization can occur.*

MPI: the Message Passing Interface

At its most basic, the Message Passing Interface (MPI) provides functions for sending and receiving messages between different processes. MPI was developed to provide a standard framework for parallel computing in any language. It specifies a library of functions — the syntax and semantics of message passing routines — that can be called from programming languages such as Fortran and C.

MPI can be thought of as "the assembly language of parallel computing," because of this generality.[1] MPI is important because it was the first portable and universally available standard for programming parallel systems and continues to be the de facto standard today.

For more information on how MPI works and how to get it installed on your machine, see the additional material for this lab.

> **Note**
>
> Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are "parallel" programs and are typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed for any general architecture.

Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial because it requires managing multiple processors and their interactions. Python, however, is an excellent language for simplifying algorithm design because it allows for problem solving without too much detail. Unfortunately, Python is not designed for high performance computing and is a notably slower

[1] *Parallel Programming with MPI*, by Peter S. Pacheco, pg. 7.

scripted language. It is best practice to prototype in Python and then to write production code in fast compiled languages such as C or Fortran.

In this lab, we will explore the Python library `mpi4py` which retains most of the functionality of C implementations of MPI and is a good learning tool. If you do not have the MPI library and mpi4py installed on your machine, please refer to the Additional Material at the end of this lab. There are three main differences to keep in mind between mpi4py and MPI in C:

- Python is array-based while C is not.
- mpi4py is object oriented but MPI in C is not.
- mpi4py supports two methods of communication to implement each of the basic MPI commands. They are the upper and lower case commands (e.g. `Bcast(...)` and `bcast(...)`). The uppercase implementations use traditional MPI datatypes while the lower case use Python's pickling method. Pickling offers extra convenience to using mpi4py, but the traditional method is faster. In these labs, we will only use the uppercase functions.

Using MPI

We will start with a Hello World program.

```
#hello.py
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

print("Hello world! I'm process number {}.".format(RANK))
```

hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpiexec -n 5 python hello.py
```

The program should output something like this:

```
Hello world! I'm process number 3.
Hello world! I'm process number 2.
Hello world! I'm process number 0.
Hello world! I'm process number 4.
Hello world! I'm process number 1.
```

Notice that when you try this on your own, the lines will not necessarily print in order. This is because there will be five separate processes running autonomously, and we cannot know beforehand which one will execute its `print()` statement first.

Achtung!

It is usually bad practice to perform I/O (e.g., call `print()`) from any process besides the root

process (rank 0), though it can be a useful tool for debugging.

How does this program work? First, the `mpiexec` program is launched. This is the program which starts MPI, a wrapper around whatever program you to pass into it. The `-n 5` option specifies the desired number of processes. In our case, 5 processes are run, with each one being an instance of the program "python". To each of the 5 instances of python, we pass the argument `hello.py` which is the name of our program's text file, located in the current directory. Each of the five instances of python then opens the `hello.py` file and runs the same program. The difference in each process's execution environment is that the processes are given different ranks in the communicator. Because of this, each process prints a different number when it executes.

MPI and Python combine to make succinct source code. In the above program, the line `from mpi4py import MPI` loads the MPI module from the mpi4py package. The line `COMM = MPI.COMM_WORLD` accesses a static communicator object, which represents a group of processes which can communicate with each other via MPI commands. The next line, `RANK = COMM.Get_rank()`, accesses the processes *rank* number. A rank is the process's unique ID within a communicator, and they are essential to learning about other processes. When the program `mpiexec` is first executed, it creates a global communicator and stores it in the variable `MPI.COMM_WORLD`. One of the main purposes of this communicator is to give each of the five processes a unique identifier, or rank. When each process calls `COMM.Get_rank()`, the communicator returns the rank of that process. `RANK` points to a local variable, which is unique for every calling process because each process has its own separate copy of local variables. This gives us a way to distinguish different processes while writing all of the source code for the five processes in a single file.

Here is the syntax for `Get_size()` and `Get_rank()`, where `Comm` is a communicator object:

Comm.Get_size() Returns the number of processes in the communicator. It will return the same number to every process. Parameters:

Return value - the number of processes in the communicator

Return type - integer

Example:

```
#Get_size_example.py
from mpi4py import MPI
SIZE = MPI.COMM_WORLD.Get_size()
print("The number of processes is {}.".format(SIZE))
```

Get_size_example.py

Comm.Get_rank() Determines the rank of the calling process in the communicator. Parameters:

Return value - rank of the calling process in the communicator

Return type - integer

Example:

```
#Get_rank_example.py
from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()
print("My rank is {}.".format(RANK))
```

Get_rank_example.py

The Communicator

A communicator is a logical unit that defines which processes are allowed to send and receive messages. In most of our programs we will only deal with the `MPI.COMM_WORLD` communicator, which contains all of the running processes. In more advanced MPI programs, you can create custom communicators to group only a small subset of the processes together. This allows processes to be part of multiple communicators at any given time. By organizing processes this way, MPI can physically rearrange which processes are assigned to which CPUs and optimize your program for speed. Note that within two different communicators, the same process will most likely have a different rank.

Note that one of the main differences between `mpi4py` and MPI in C or Fortran, besides being array-based, is that `mpi4py` is largely object oriented. Because of this, there are some minor changes between the `mpi4py` implementation of MPI and the official MPI specification.

For instance, the MPI Communicator in `mpi4py` is a Python class and MPI functions like `Get_size()` or `Get_rank()` are instance methods of the communicator class. Throughout these MPI labs, you will see functions like `Get_rank()` presented as `Comm.Get_rank()` where it is implied that `Comm` is a communicator object.

Separate Codes in One File

When an MPI program is run, each process receives the same code. However, each process is assigned a different rank, allowing us to specify separate behaviors for each process. In the following code, the three processes perform different operations on the same pair of numbers.

```
#separateCode.py
from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()

a = 2
b = 3
if RANK == 0:
  print a + b
elif RANK == 1:
  print a*b
elif RANK == 2:
  print max(a, b)
```

separateCode.py

Problem 1. Write a program in which processes with an even rank print "Hello" and process with an odd rank print "Goodbye." Print the process number along with the "Hello" or "Goodbye" (for example, "Goodbye from process 3").

Message Passing between Processes

Let us begin by demonstrating a program designed for two processes. One will draw a random number and then send it to the other. We will do this using the routines `Comm.Send()` and `Comm.Recv()`.

```
#passValue.py
import numpy as np
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if RANK == 1:  # This process chooses and sends a random value
    num_buffer = np.random.rand(1)
    print("Process 1: Sending: {} to process 0.".format(num_buffer))
    COMM.Send(num_buffer, dest=0)
    print("Process 1: Message sent.")
if RANK == 0:  # This process recieves a value from process 1
    num_buffer = np.zeros(1)
    print("Process 0: Waiting for the message... current num_buffer={}.".format↩
        (num_buffer))
    COMM.Recv(num_buffer, source=1)
    print("Process 0: Message recieved! num_buffer={}.".format(num_buffer))
```

passValue.py

To illustrate simple message passing, we have one process choose a random number and then pass it to the other. Inside the receiving process, we have it print out the value of the variable `num_buffer` before it calls `Recv()` to prove that it really is receiving the variable through the message passing interface.

Here is the syntax for `Send()` and `Recv()`, where `Comm` is a communicator object:

Comm.Send(buf, dest=0, tag=0) Performs a basic send from one process to another. Parameters:

buf (array-like) : data to send

dest (integer) : rank of destination

tag (integer) : message tag

The `buf` object is not as simple as it appears. It must contain a pointer to a Numpy array. It cannot, for example, simply pass a string. The string would have to be packaged inside an array first.

Comm.Recv(buf, source=0, tag=0, Status status=None) Basic point-to-point receive of data. Parameters:

buf (array-like) : initial address of receive buffer (choose receipt location)

source (integer) : rank of source

tag (integer) : message tag

status (Status) : status of object

Example:

```python
#Send_example.py
from mpi4py import MPI
import numpy as np

RANK = MPI.COMM_WORLD.Get_rank()

a = np.zeros(1, dtype=int)  # This must be an array.
if RANK == 0:
    a[0] = 10110100
    MPI.COMM_WORLD.Send(a, dest=1)
elif RANK == 1:
    MPI.COMM_WORLD.Recv(a, source=0)
    print(a[0])
```

Send_example.py

Problem 2. Write a script that runs on two processes and passes an n by 1 vector of random values from one process to the other. Write it so that the user passes the value of n in as a command-line argument. The following code demonstrates how to access command-line arguments.

```python
from sys import argv

# Pass in the first command line argument as n.
n = int(argv[1])
```

Note

`Send()` and `Recv()` are referred to as *blocking* functions. That is, if a process calls `Recv()`, it will sit idle until it has received a message from a corresponding `Send()` before it will proceed. (However, in Python the process that calls `Comm.Send` will *not* necessarily block until the message is received, though in C, `MPI_Send` does block) There are corresponding *non-blocking* functions `Isend()` and `Irecv()` (The *I* stands for immediate). In essence, `Irecv()` will return immediately. If a process calls `Irecv()` and doesn't find a message ready to be picked up, it will indicate to the system that it is expecting a message, proceed beyond the `Irecv()` to do other useful work, and then check back later to see if the message has arrived. This can be used to dramatically improve performance.

Problem 3. Write a script in which the process with rank i sends a random value to the process with rank $i + 1$ in the global communicator. The process with the highest rank will

send its random value to the root process. Notice that we are communicating in a ring. For communication, only use `Send()` and `Recv()`. The program should work for any number of processes. Does the order in which `Send()` and `Recv()` are called matter?

Note

When calling `Comm.Recv`, you can allow the calling process to accept a message from any process that happened to be sending to the receiving process. This is done by setting source to a predefined MPI constant, `source=ANY_SOURCE` (note that you would first need to import this with from `mpi4py.MPI import ANY_SOURCE` or use the syntax `source=MPI.ANY_SOURCE`).

Application: Monte Carlo Integration

Monte Carlo integration uses random sampling to approximate volumes (whereas most numerical integration methods employ some sort of regular grid). It is a useful technique, especially when working with higher-dimensional integrals. It is also well-suited to parallelization because it involves a large number of independent operations. In fact, Monte Carlo algorithms can be made "embarassingly parallel" — the processes don't need to communicate with one another during execution, simply reporting results to the root process upon completion.

In a simple example, the following code calculates the value of π by sampling random points inside the square $[-1, 1] \times [-1, 1]$. Since the volume of the unit circle is π and the volume of the square is 4, the probability of a given point landing inside the unit circle is $\pi/4$, so the proportion of samples that fall within the unit circle should also be $\pi/4$. The program samples $N = 2000$ points, determines which samples are within the unit circle (say M are), and estimates $\pi \approx 4M/N$.

```
# pi.py
import numpy as np
from scipy import linalg as la


# Get 2000 random points in the 2-D domain [-1,1]x[-1,1].
points = np.random.uniform(-1, 1, (2,2000))

# Determine how many points are within the unit circle.
lengths = la.norm(points, axis=0)
num_within = np.count_nonzero(lengths < 1)

# Estimate the circle's area.
print(4 * (num_within / 2000))
```

pi.py

```
$ python pi.py
3.166
```

Problem 4. The n-dimensional *open unit ball* is the set $U_n = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 < 1\}$. Write a script that accepts integers n and N on the command line. Estimate the volume of U_n by drawing N points over the n-dimensional domain $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ on each available process except the root process (for a total of $(r-1)N$ draws, where r is the number of processes). Have the root process print the volume estimate.
(Hint: the volume of $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ is 2^n.)

When $n = 2$, this is the same experiment outlined above so your function should return an approximation of π. The volume of the U_3 is $\frac{4}{3}\pi \approx 4.18879$, and the volume of U_4 is $\frac{\pi^2}{2} \approx 4.9348$. Try increasing the number of sample points N or processes r to see if your estimates improve.

NOTE

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages requires a level of synchronization and causes some processes to pause as they wait to receive or send messages, negating the advantages of parallelism. It is also important to divide work evenly between simultaneous processes, as a program can only be as fast as its slowest process. This is called load balancing, and can be difficult in more complex algorithms.

Additional Material

Installation of MPI

MPI is a library of functions that interface with your computer's hardware to provide optimal parallel computing performance. In order to use mpi4py, we need to have an MPI Library on installed on the computer as well as the mpi4py package. When you invoke mpi4py in your python code, mpi4py takes what you have written in python and applies it using an MPI Library, so only installing mpi4py is not enough to use MPI.

Installing MPI Library

1. For Linux/Mac: We recommend using OpenMPI for your MPI Library installation, though it is not the only library available.
   - Download the binary files from `https://www-lb.open-mpi.org/software/ompi/v4.0/`.
   - Extract the files from their compressed form and navigate into the new folder titled "openmpi-X.X.X".
   - Configure the files so that they will install correctly on your machine.
   - Compile OpenMPI and install it.

   The following is a bash script written for Linux that will install OpenMPI version 4.0.2. It will take about 15 minutes to complete.

```
#!/bin/bash
# download openMPI
wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi↩
    -4.0.2.tar.gz
```

```
# extract the files
tar -zxf openmpi-4.0.2.tar.gz
cd openmpi-4.0.2
# configure the files
./configure --prefix=/usr/local/openmpi
# compile openMPI
make all
# install openMPI
sudo make install
```

Finally, you must add OpenMPI to your PATH variable. This is so your computer knows where to look when it wants to execute a certain MPI command. Here is a link that describes how to edit the PATH variable `https://gist.github.com/nex3/c395b2f8fd4b02068be37c961301caa7`.

On linux you will open a file called .bashrc, on Mac the file is called .bash_profile, both are in the home directory. Add the following line, save the file, and restart your terminal.

```
export PATH=/usr/local/openmpi/bin:$PATH
```

2. For Windows: There is only one free MPI library available for Windows at `https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx`. Download the appropriate .exe or .msi file to install on your machine.

Installing mpi4py

1. For All Systems: The easiest installation is using `conda install mpi4py`. You may also run `pip install mpi4py`

6 Metropolis Algorithm

Lab Objective: *Understand the basic principles of the Metropolis algorithm and apply these ideas to the Ising Model.*

The Metropolis Algorithm

Sampling from a given probability distribution is an important task in many different applications found throughout the sciences. When these distributions are complicated, as is often the case when modeling real-world problems, direct sampling methods can become difficult, as they might involve computing high-dimensional integrals. The Metropolis algorithm is an effective method to sample from many distributions, requiring only that we be able to evaluate the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

The Metropolis algorithm is an MCMC sampling method which generates a sequence of random variables, similar to Gibbs sampling. These random variables form a Markov Chain whose invariant distribution is equal to the distribution from which we wish to sample. Suppose that $h : \mathbb{R}^n \to \mathbb{R}$ is the probability density function of distribution, and suppose that $f(\boldsymbol{\theta}) = c \cdot h(\boldsymbol{\theta})$ for some nonzero constant c (in practice, we assume that f is an easy function to evaluate, while h is difficult). Let $Q : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a symmetric *proposal function* (so that $Q(\cdot, \mathbf{y})$ is a probability density function for all $\mathbf{y} \in \mathbb{R}^n$, and $Q(\mathbf{x}, \mathbf{y}) = Q(\mathbf{y}, \mathbf{x})$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$) and let $A : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be an *acceptance function* defined by

$$A(\mathbf{x}, \mathbf{y}) = \min\left(1, \frac{f(\mathbf{x})}{f(\mathbf{y})}\right).$$

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 6.1. The Metropolis algorithm can be interpreted as follows: given our current state $\mathbf{y}$, we propose a new state according to the distribution $Q(\cdot, \mathbf{y})$. We then accept or reject it according to A. We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h. Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

$$\log A(\mathbf{x}, \mathbf{y}) = \min(0, \log f(\mathbf{x}) - \log f(\mathbf{y})).$$

Algorithm 6.1 Metropolis Algorithm

procedure METROPOLIS ALGORITHM
  Choose initial point $\mathbf{y}_0$.
  for $t = 1, 2, \ldots$ **do**
    Draw $\mathbf{x} \sim Q(\cdot, \mathbf{y}_{t-1})$
    Draw $a \sim \text{unif}(0, 1)$
    if $a \leq A(\mathbf{x}, \mathbf{y}_{t-1})$ **then**
      $\mathbf{y}_t = \mathbf{x}$
    else
      $\mathbf{y}_t = \mathbf{y}_{t-1}$
  Return $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots$

Let's apply the Metropolis algorithm to a simple example of Bayesian analysis. Consider the problem of computing the posterior distribution over the mean μ and variance σ^2 of a normal distribution for which we have n data points $y_1, \ldots, y_n$. For concreteness, we use the data in `examscores.csv` and we assume the prior distributions

$$\mu \sim \mathcal{N}(m = 80,\ s^2 = 16)$$
$$\sigma^2 \sim IG(\alpha = 3, \beta = 50).$$

In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 \,|\, y_1, \ldots, y_N) = \frac{p(\mu)p(\sigma^2)\prod_{i=1}^{n}\mathcal{N}(y_i \,|\, \mu, \sigma^2)}{\int_{-\infty}^{\infty}\int_{0}^{\infty} p(\mu)p(\sigma^2)\prod_{i=1}^{n}\mathcal{N}(y_i \,|\, \mu, \sigma^2)\, d\sigma^2 d\mu}.$$

However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to μ and σ^2, the numerator can serve as the function f in the Metropolis algorithm, and the denominator can serve as the constant c.

We choose our proposal function to be based on a bivariate Normal distribution:

$$Q(x, y) = \mathcal{N}(x \,|\, y, sI),$$

where I is the 2×2 identity matrix and s is some positive scalar.

```
>>> def proposal(y, s):
...     """The proposal function Q(x,y) = N(x|y,sI)."""
...     return stats.multivariate_normal.rvs(mean=y, cov=s*np.eye(len(y)))
...
>>> def propLogDensity(x):
...     """Calculate the log of the proportional density."""
...     logprob = muprior.logpdf(x[0]) + sig2prior.logpdf(x[1])
...     logprob += stats.norm.logpdf(scores, loc=x[0], scale=sqrt(x[1])).sum()
...     return logprob    # ^this is where the scores are used.
...
>>> def acceptance(x, y):
...     return min(0, propLogDensity(x) - propLogDensity(y))
```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log densities of the samples and the proportion of proposed samples that were accepted.

We can evaluate the quality of our results by plotting the log probabilities, the μ samples, the σ^2 samples, and kernel density estimators for the marginal posterior distributions of μ and σ^2. The kernel density estimator is the posterior distribution for a parameter. It measures the frequency of each draw. In this example, the kernel density estimator for μ should be approximately normal, and the kernel density estimator for σ^2 should be approximately an inverse gamma.

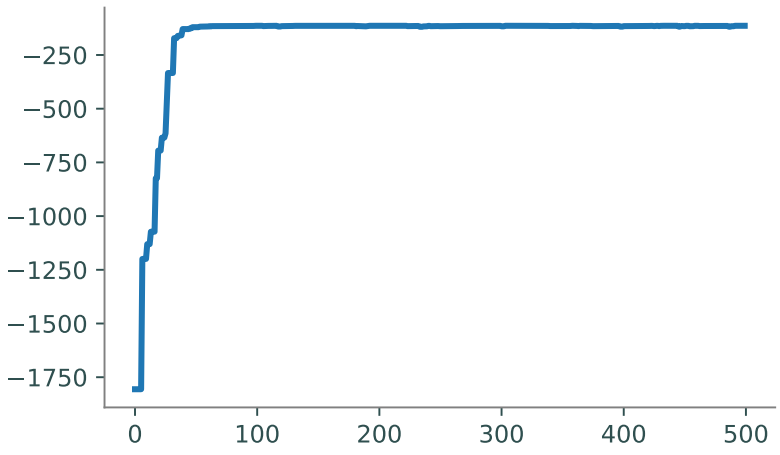


Figure 6.1: Log densities of the first 500 Metropolis samples.

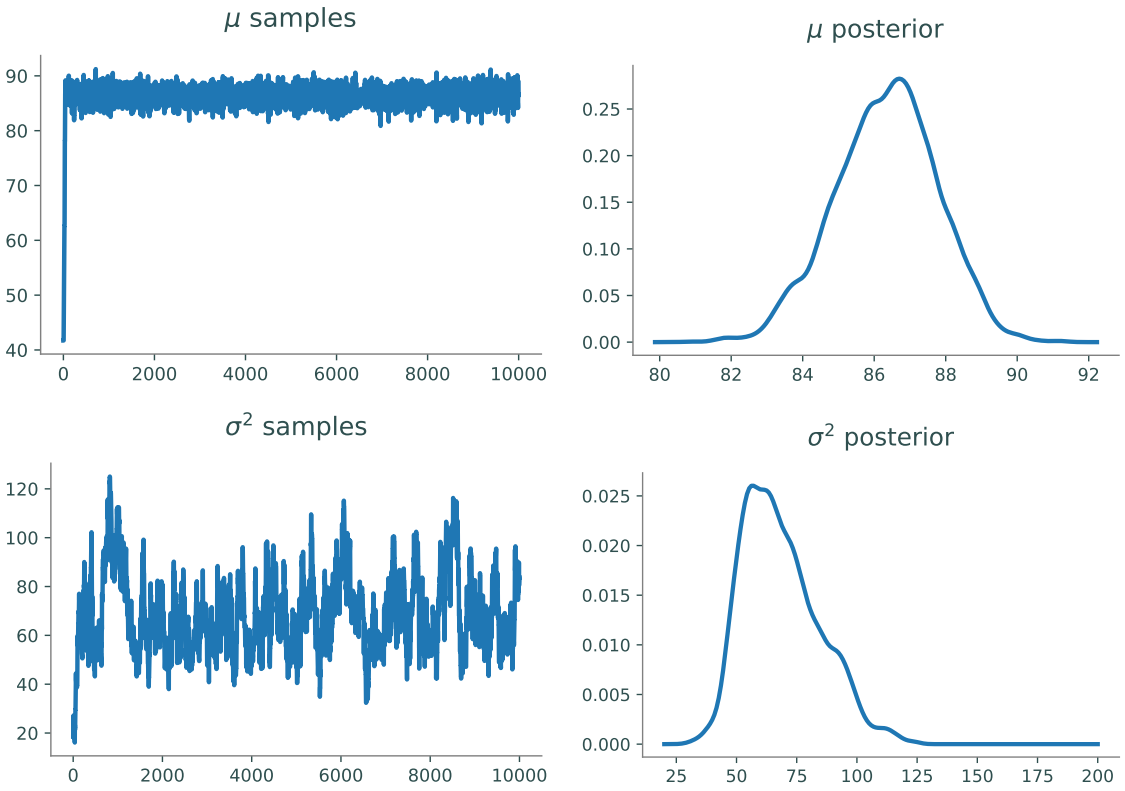


Figure 6.2: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

Problem 1. Write a function that uses the Metropolis Hastings algorithm to draw from the posterior distribution over the mean μ and variance σ^2. Use the given functions and algorithm 6.1 to complete the problem.

Your function should return an array of draws, an array of the log probabilities, and an acceptance rate. Use the following code to check your work. Using the seaborn.kdeplot function, plot the first 500 log probabilities, the μ samples and posterior distribution, and the σ^2 samples and posterior distribution. The results should be *similar* to Figures 6.1 and 6.2.

When comparing `a` to the acceptance, remember to use `log(a)` as we are in log space.

```
# Load in the data and initialize hyperparameters.
>>> scores = np.load("examscores.npy")

# Prior sigma^2 ~ IG(alpha, beta)
>>> alpha = 3
>>> beta = 50

#Prior mu ~ N(m, s)
>>> m = 80
```

```
>>> s = 4

# Initialize the prior distributions.
>>> muprior = stats.norm(loc=m, scale=sqrt(s**2))
>>> sig2prior = stats.invgamma(alpha, scale=beta)
```

Hint: The seaborn package is very useful in plotting kernel densities, with the distplot method. See the documentation.

The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice Λ of sites. We say $i \sim j$ if i and j are adjacent sites. Each site i in our lattice is assigned an associated *spin* $\sigma_i \in \{\pm 1\}$. A *state* in our Ising model is a particular spin configuration $\sigma = (\sigma_k)_{k \in \Lambda}$. If $L = |\Lambda|$, then there are 2^L possible states in our model. If L is large, the state space becomes huge, which is why MCMC sampling methods (in particular the Metropolis algorithm) are so useful in calculating model estimations.

With any spin configuration σ, there is an associated energy

$$H(\sigma) = -J \sum_{i \sim j} \sigma_i \sigma_j$$

where $J > 0$ for ferromagnetic materials, and $J < 0$ for antiferromagnetic materials. Throughout this lab, we will assume $J = 1$, leaving the energy equation to be $H(\sigma) = -\sum_{i \sim j} \sigma_i \sigma_j$ where the interaction from each pair is added only once.

We will consider a lattice that is a 100×100 square grid. The adjacent sites for a given site are those directly above, below, to the left, and to the right of the site, so to speak. For sites on the edge of the grid, we assume it wraps around. In other words, a site at the farthest left side of the grid is adjacent to the corresponding site on the farthest right side. Thus, a single spin configuration can be represented as a 100×100 array, with entries of ± 1.

The following code will construct a random spin configuration of size n:

```
def random_lattice(n):
    """Constructs a random spin configuration for an nxn lattice."""
    random_spin = np.zeros((n,n))
    for k in range(n):
        random_spin[k,:] = 2*np.random.binomial(1,.5, n) -1
    return random_spin
```

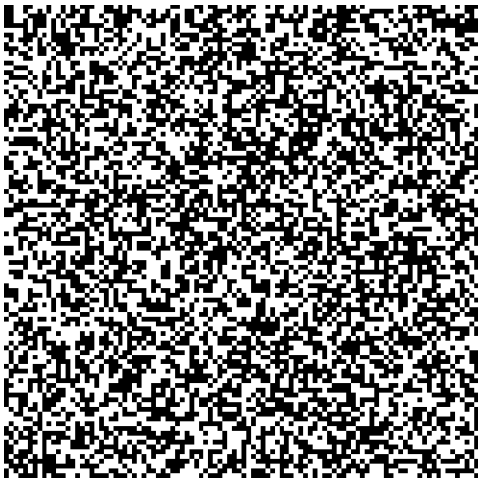


Figure 6.3: Spin configuration from random initialization.

Problem 2. Write a function that accepts a spin configuration σ for a lattice as a NumPy array. Compute the energy $H(\sigma)$ of the spin configuration. Be careful to not double count site pair interactions!
(Hint: `np.roll()` may be helpful.)

Different spin configurations occur with different probabilities, depending on the energy of the spin configuration and $\beta > 0$, a quantity inversely proportional to the temperature. More specifically, for a given β, we have

$$\mathbb{P}_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta}$$

where $Z_\beta = \sum_\sigma e^{-\beta H(\sigma)}$. Because there are $2^{100 \cdot 100} = 2^{10000}$ possible spin configurations for our particular lattice, computing this sum is infeasible. However, the numerator is quite simple, provided we can efficiently compute the energy $H(\sigma)$ of a spin configuration. Thus the ratio of the probability densities of two spin configurations is simple:

$$\frac{\mathbb{P}_\beta(\sigma^*)}{\mathbb{P}_\beta(\sigma)} = \frac{e^{-\beta H(\sigma^*)}}{e^{-\beta H(\sigma)}} = e^{\beta(H(\sigma) - H(\sigma^*))}$$

The simplicity of this ratio should lead us to think that a Metropolis algorithm might be an appropriate way by which to sample from the spin configuration probability distribution, in which case the acceptance probability would be

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases} \tag{6.1}$$

By choosing our transition matrix Q cleverly, we can also make it easy to compute the energy for any proposed spin configuration. We restrict our possible proposals to only those spin configurations

in which we have flipped the spin at exactly one lattice site, i.e. we choose a lattice site i and flip its spin. Thus, there are only L possible proposal spin configurations σ^* given σ, each being proposed with probability $\frac{1}{L}$, and such that $\sigma_j^* = \sigma_j$ for all $j \neq i$, and $\sigma_i^* = -\sigma_i$. Note that we would never actually write out this matrix (it would be $2^{10000} \times 2^{10000}$). Computing the proposed site's energy is simple: if the spin flip site is i, then we have

$$H(\sigma^*) = H(\sigma) + 2 \sum_{j:j\sim i} \sigma_i \sigma_j. \tag{6.2}$$

Problem 3. Write a function that accepts an integer n and chooses a pair of indices (i, j) where $0 \leq i, j \leq n-1$. Each possible pair should have an equal probability $\frac{1}{n^2}$ of being chosen.

Problem 4. Write a function that accepts a spin configuration σ, its energy $H(\sigma)$, and integer indices i and j. Use (6.2) to compute the energy of the new spin configuration σ^*, which is σ but with the spin flipped at the (i, j)th entry of the corresponding lattice. Do not explicitly construct the new lattice for σ^*.

Problem 5. Write a function that accepts a float β and spin configuration energies $H(\sigma)$ and $H(\sigma^*)$. Using (6.1), calculate whether or not the new spin configuration σ^* should be accepted (return `True` or `False`). Consider doing the calculations in log space. (Hint: np.random.binomial() might be useful)

To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator Z_β, which is generally the reason we have to use a Metropolis algorithm to begin with. We can get away with examining only $-\beta H(\sigma)$. We should see this value increase as the algorithm proceeds, and it should converge once we are sampling from the correct distribution. Note that we don't expect these values to converge to a specific value, but rather to a restricted range of values.

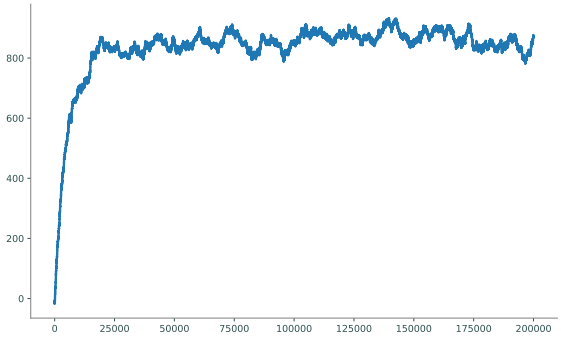
Problem 6. Write a function that accepts a float $\beta > 0$ and integers n, `n_samples`, and `burn_in`. Initialize an $n \times n$ lattice for a spin configuration σ using Problem 2. Use the Metropolis algorithm to (potentially) update the lattice `burn_in` times.

1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration σ^*.
2. Use Problem 4 to calculate the energy $H(\sigma^*)$ of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set $\sigma = \sigma^*$ by flipping the spin at the indicated site.
4. Track $-\beta H(\sigma)$ at each iteration (independent of acceptance).

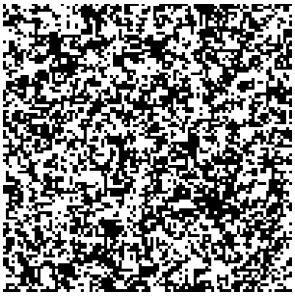
After the burn-in period, continue the iteration `n_samples` times, also recording every 100th sample (to prevent memory failure). The acceptance rate is counted after the burn-in period. Return the samples, the sequence of weighted energies $-\beta H(\sigma)$, and the acceptance rate.

Test your sampler on a 100×100 grid with 200000 total iterations, with `n_samples` large enough so that you will keep 50 samples, for $\beta = 0.2, 0.4, 1$. Plot the proportional log probabilities, as well as a late sample from each test. How does the ferromagnetic material behave differently with differing temperatures? Recall that β is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figure 6.4.

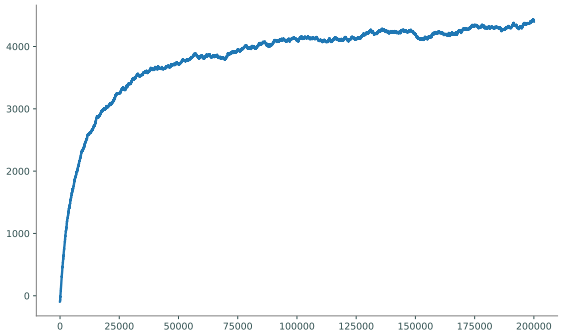
To show the spin configuration, use `plt.imshow(L,cmap='gray')`.



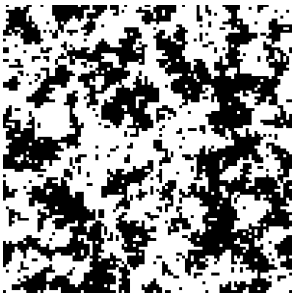
(a) Proportional log probs when $\beta = 0.2$.



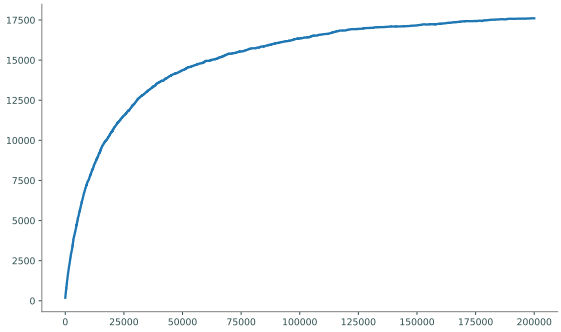
(b) Spin configuration sample when $\beta = 0.2$.



(c) Proportional log probs when $\beta = 0.4$.



(d) Spin configuration sample when $\beta = 0.4$.



(e) Proportional log probs when $\beta = 1$.

(f) Spin configuration sample when $\beta = 1$.

Figure 6.4

7 Gibbs Sampling and LDA

Lab Objective: *Understand the basic principles of implementing a Gibbs sampler. Apply this to Latent Dirichlet Allocation.*

Gibbs Sampling

Gibbs sampling is an MCMC sampling method in which we construct a Markov chain which is used to sample from a desired joint (conditional) distribution

$$\mathbb{P}(x_1, \cdots, x_n|\mathbf{y}).$$

Often it is difficult to sample from this high-dimensional joint distribution, while it may be easy to sample from the one-dimensional conditional distributions

$$\mathbb{P}(x_i|\mathbf{x}_{-i}, \mathbf{y})$$

where $\mathbf{x}_{-i} = x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n$.

Algorithm 7.1 Basic Gibbs Sampling Process.

```
procedure GIBBS SAMPLER
    Randomly initialize x_1, x_2, ..., x_n.
    for k = 1, 2, 3, ... do
        for i = 1, 2, ..., n do
            Draw x ~ P(x_i | x_{-i}, y)
            Fix x_i = x
        x^(k) = (x_1, x_2, ..., x_n)
```

A Gibbs sampler proceeds according to Algorithm 7.1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of $\mathbf{x}^{(k)}$ after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible $\mathbf{x}$. The unique invariant distribution for the chain is the desired joint distribution

$$\mathbb{P}(x_1, \cdots, x_n|\mathbf{y}).$$

Thus, after a burn-in period, our samples $\mathbf{x}^{(k)}$ are effectively samples from the desired distribution.

Consider the dataset of N scores from a calculus exam in the file `examscores.npy`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean μ and variance σ^2. Because we are unsure of the true value of μ and σ^2, we take a Bayesian approach and place priors on each parameter to quantify this uncertainty:

$$\begin{aligned} \mu &\sim N(\nu, \tau^2) && \text{(a normal distribution)} \\ \sigma^2 &\sim IG(\alpha, \beta) && \text{(an inverse gamma distribution)} \end{aligned}$$

Letting $\mathbf{y} = (y_1, \dots, y_N)$ be the set of exam scores, we would like to update our beliefs of μ and σ^2 by sampling from the posterior distribution

$$\mathbb{P}(\mu, \sigma^2 | \mathbf{y}, \nu, \tau^2, \alpha, \beta).$$

Sampling directly can be difficult. However, we *can* easily sample from the following conditional distributions:

$$\begin{aligned} \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) \end{aligned}$$

The reason for this is that these conditional distributions are *conjugate* to the prior distributions, and hence are part of the same distributional families as the priors. In particular, we have

$$\begin{aligned} \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) &= N(\mu^*, (\sigma^*)^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) &= IG(\alpha^*, \beta^*), \end{aligned}$$

where

$$\begin{aligned} (\sigma^*)^2 &= \left(\frac{1}{\tau^2} + \frac{N}{\sigma^2} \right)^{-1} \\ \mu^* &= (\sigma^*)^2 \left(\frac{\nu}{\tau^2} + \frac{1}{\sigma^2} \sum_{i=1}^{N} y_i \right) \\ \alpha^* &= \alpha + \frac{N}{2} \\ \beta^* &= \beta + \frac{1}{2} \sum_{i=1}^{N} (y_i - \mu)^2 \end{aligned}$$

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling μ and sampling σ^2. We can sample from a normal distribution and an inverse gamma distribution as follows:

```
>>> from math import sqrt
>>> from scipy.stats import norm
>>> from scipy.stats import invgamma
>>> mu = 0. # the mean
>>> sigma2 = 9. # the variance
>>> normal_sample = norm.rvs(mu, scale=sqrt(sigma))
>>> alpha = 2.
>>> beta = 15.
>>> invgamma_sample = invgamma.rvs(alpha, scale=beta)
```

Note that when sampling from the normal distribution, we need to set the `scale` parameter to the standard deviation, *not* the variance.

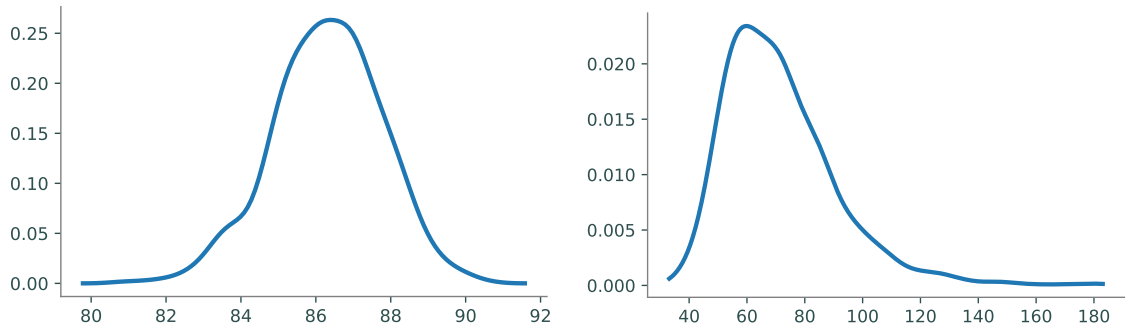
Problem 1. Write a function that accepts data $\mathbf{y}$, prior parameters ν, τ^2, α, and β, and an integer n. Use Gibbs sampling to generate n samples of μ and σ^2 for the exam scores problem.

Test your sampler with priors $\nu = 80$, $\tau^2 = 16$, $\alpha = 3$, and $\beta = 50$, collecting 1000 samples. Plot your samples of μ and your samples of σ^2. They should each to converge quickly.

We'd like to look at the posterior marginal distributions for μ and σ^2. To plot these from the samples, use a kernel density estimator from `scipy.stats`. If our samples of μ are called `mu_samples`, then we can do this with the following code.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> from scipy.stats import gaussian_kde

>>> mu_kernel = gaussian_kde(mu_samples)
>>> x = np.linspace(min(mu_samples) - 1, max(mu_samples) + 1, 200)
>>> plt.plot(x, mu_kernel(x))
>>> plt.show()
```



(a) Posterior distribution of μ.

(b) Posterior distribution of σ^2.

Figure 7.1: Posterior marginal probability densities for μ and σ^2.

Keep in mind that the plots above are of the posterior distributions of the *parameters*, not of the scores. If we would like to compute the posterior distribution of a new exam score $\tilde{y}$ given our data $\mathbf{y}$ and prior parameters, we compute what is known as the *posterior predictive distribution*:

$$\mathbb{P}(\tilde{y}|\mathbf{y}, \lambda) = \int_{\Theta} \mathbb{P}(\tilde{y}|\Theta)\mathbb{P}(\Theta|\mathbf{y}, \lambda)d\Theta$$

where Θ denotes our parameters (in our case μ and σ^2) and λ denotes our prior parameters (in our case ν, τ^2, α, and β).

Rather than actually computing this integral for each possible $\tilde{y}$, we can do this by sampling scores from our parameter samples. In other words, sample

$$\tilde{y}_{(t)} \sim N(\mu_{(t)}, \sigma^2_{(t)})$$

for each sample pair $\mu_{(t)}, \sigma^2_{(t)}$. Now we have essentially drawn samples from our posterior predictive distribution, and we can use a kernel density estimator to plot this distribution from the samples.

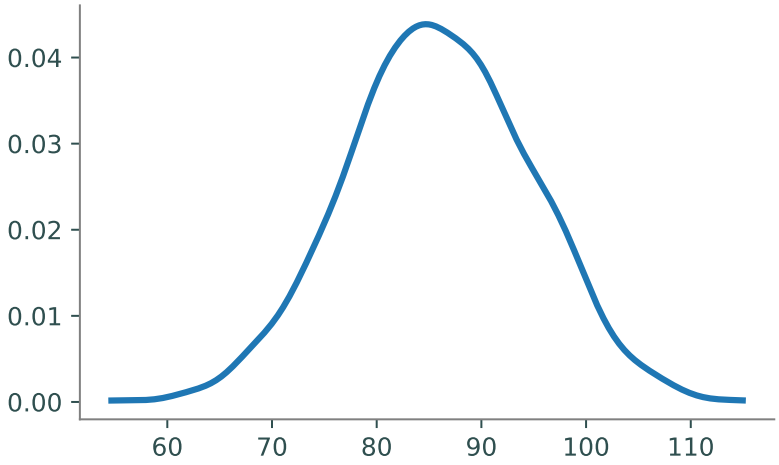


Figure 7.2: Predictive posterior distribution of exam scores.

Problem 2. Plot the kernel density estimators for the posterior distributions of μ and σ^2. You should get plots similar to those in Figure 7.1.

Next, use your samples of μ and σ^2 to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. Compare your plot to Figure 7.2.

Latent Dirichlet Allocation

Gibbs sampling can be applied to an interesting problem in natural language processing (NLP): determining which topics are prevalent in a document. *Latent Dirichlet Allocation* (LDA) is a generative model for a collection of text documents. It supposes that there is some fixed vocabulary (composed of V distinct terms) and K different topics, each represented as a probability distribution ϕ_k over the vocabulary, each with a Dirichlet prior β. This means $\phi_{k,v}$ is the probability that topic k is represented by vocabulary term v.

With the vocabulary and topics chosen, the LDA model assumes that we have a set of M documents (each "document" may be a paragraph or other section of the text, rather than a "full" document). The m-th document consists of N_m words, and a probability distribution θ_m over the topics is drawn from a Dirichlet distribution with parameter α. Thus $\theta_{m,k}$ is the probability that document m is assigned the label k. If $\phi_{k,v}$ and $\theta_{m,k}$ are viewed as matrices, their rows sum to one.

We will now iterate through each document in the same manner. Assume we are working on document m, which you will recall contains N_m words. For word n, we first draw a topic assignment $z_{m,n}$ from the categorical distribution θ_m, and then we draw a word $w_{m,n}$ from the categorical distribution $\phi_{z_{m,n}}$. Throughout this implementation, we assume α and β are scalars. In summary, we have

1. Draw $\phi_k \sim \text{Dir}(\beta)$ for $1 \leq k \leq K$.

2. For $1 \leq m \leq M$:

   (a) Draw $\theta_m \sim \text{Dir}(\alpha)$.

   (b) Draw $z_{m,n} \sim \text{Cat}(\theta_m)$ for $1 \leq n \leq N_m$.

   (c) Draw $w_{m,n} \sim \text{Cat}(\phi_{z_{m,n}})$ for $1 \leq n \leq N_m$.

We end up with n words which represent document m. Note that these words are *not* necessarily distinct from one another; indeed, we are most interested in the words that have been repeated the most.

This is typically depicted with graphical plate notation as in Figure 7.3.

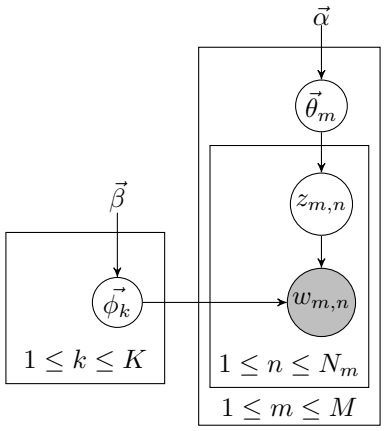


Figure 7.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables $w_{m,n}$ are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each ϕ_k and each θ_m. This will allow us to understand what each topic is, as well as understand how each document is distributed over the K topics. In other words, we want to predict the topic of each document, and also which words best represent this topic. We can estimate these well if we know $z_{m,n}$ for each m, n, collectively referred to as $\mathbf{z}$. Thus, we need to sample $\mathbf{z}$ from the posterior distribution $\mathbb{P}(\mathbf{z}|\mathbf{w}, \alpha, \beta)$, where $\mathbf{w}$ is the collection words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$, the conditional posterior distributions

$$\mathbb{P}(z_{m,n} = k|\mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta)$$

have nice, closed form solutions, making them easy to sample from.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k|\mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta) \propto \frac{(n_{(k,m,\cdot)}^{-(m,n)} + \alpha)(n_{(k,\cdot,w_{m,n})}^{-(m,n)} + \beta)}{n_{(k,\cdot,\cdot)}^{-(m,n)} + V\beta}$$

where

$$
\begin{aligned}
n_{(k,m,\cdot)} &= \text{ the number of words in document } m \text{ assigned to topic } k \\
n_{(k,\cdot,v)} &= \text{ the number of times term } v = w_{m,n} \text{ is assigned to topic } k \\
n_{(k,\cdot,\cdot)} &= \text{ the number of times topic } k \text{ is assigned in the corpus} \\
n^{-(m,n)}_{(k,m,\cdot)} &= n_{(k,m,\cdot)} - \mathbf{1}_{z_{m,n}=k} \\
n^{-(m,n)}_{(k,\cdot,v)} &= n_{(k,\cdot,v)} - \mathbf{1}_{z_{m,n}=k} \\
n^{-(m,n)}_{(k,\cdot,\cdot)} &= n_{(k,\cdot,\cdot)} - \mathbf{1}_{z_{m,n}=k}
\end{aligned}
$$

Thus, if we simply keep track of these count matrices, then we can easily create a Gibbs sampler over the topic assignments. This is actually a particular class of samplers known as *collapsed Gibbs samplers*, because we have collapsed the sampler by integrating out θ and ϕ.

We have provided for you the structure of a Python object `LDACGS` with several methods, listed at the end of the lab. The object is already defined to have attributes `n_topics`, `documents`, `vocab`, `alpha`, and `beta`, where `vocab` is a list of strings (terms), and documents is a list of dictionaries (a dictionary for each document). Each entry in dictionary m is of the form $n : w$, where w is the index in `vocab` of the n^{th} word in document m.

Throughout this lab we will guide you through writing several more methods in order to implement the Gibbs sampler. The first step is to initialize our assignments, and create the count matrices $n_{(k,m,\cdot)}, n_{(k,\cdot,v)}$ and vector $n_{(k,\cdot,\cdot)}$.

Problem 3. Complete the method `initialize()`. By randomly assigning initial topics, fill in the count matrices and topic assignment dictionary. In this method, you will initialize the count matrices (among other things). Note that the notation provided in the code is slightly different than that used above. Be sure to understand how the formulae above connect with the code.

To be explicit, you will need to initialize nmz, nzw, and nz to be zero arrays of the correct size. Then, in the second for loop, you will assign z to be a random integer in the correct range of topics. In the increment step, you need to figure out the correct indices to increment by one for each of the three arrays. Finally, assign *topics* as given.

The next method we need to write fully outlines a sweep of the Gibbs sampler.

Problem 4. Complete the method `_sweep()`, which needs to iterate through each word of each document. It should call on the method `_conditional()` to get the conditional distribution at each iteration.

Note that the first part of this method will undo what `initialize()` did. Then we will use the conditional distribution (instead of the uniform distribution we used previously) to pick a more accurate topic assignment. Finally, the latter part repeats what we did in `initialize()`, but does so using this more accurate topic assignment.

We are now prepared to write the full Gibbs sampler.

Problem 5. Complete the method `sample()`. The argument *filename* is the name and location of a .txt file, where each line is considered a document. The corpus is built by method `buildCorpus`, and stopwords are removed (if argument *stopwords* is provided). Burn in the Gibbs sampler, computing and saving the log-likelihood with the method `_loglikelihood`. After the burn in, iterate further, accumulating your count matrices, by adding `nzw` and `nmz` to `total_nzw` and `total_nmz` respectively, where you only add every $sample\_rate^{th}$ iteration. Also save each log-likelihood.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on one of Ronald Reagan's State of the Union addresses, found in `reagan.txt`.

Problem 6. Create an `LDACGS` object with 20 topics, letting α and β be the default values. Run the Gibbs sampler, with a burn in of 100 iterations, accumulating 10 samples, only keeping the results of every 10th sweep. Use `stopwords.txt` as the stopwords file.

Plot the log-likelihoods. How long did it take to burn in?

We can estimate the values of each ϕ_k and each θ_m as follows:

$$\widehat{\theta}_{m,k} = \frac{n_{(k,m,\cdot)} + \alpha}{K \cdot \alpha + \sum_{k=1}^{K} n_{(k,m,\cdot)}}$$
$$\widehat{\phi}_{k,v} = \frac{n_{(k,\cdot,v)} + \beta}{V \cdot \beta + \sum_{v=1}^{V} n_{(k,\cdot,v)}}$$

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions ϕ_k by looking at the n terms with the highest probability, where n is small (say 10 or 20). We have provided a method `topterms` which does this for you.

Problem 7. Using the methods described above, examine the topics for Reagan's addresses. As best as you can, come up with labels for each topic. If $ntopics = 20$ and $n = 10$, we will get the top 10 words that represent each of the 20 topics; for each topic, decide what these ten words jointly represent.

We can use $\widehat{\theta}$ to find the paragraphs in Reagan's addresses that focus the most on each topic. The documents with the highest values of $\widehat{\theta}_k$ are those most heavily focused on topic k. For example, if you chose the topic label for topic p to be *the Cold War*, you can find the five highest values in $\widehat{\theta}_p$, which will tell you which five paragraphs are most centered on the Cold War.

Let's take a moment to see what our Gibbs sampler has accomplished. By simply feeding in a group of documents, and with no human input, we have found the most common topics discussed, which are represented by the words most frequently used in relation to that particular topic. The only work that the user has done is to assign topic labels, saying what the words in each group have in common. As you may have noticed, however, these topics may or may not be *relevant* topics. You might have noticed that some of the most common topics were simply English particles (words such as *a*, *the*, *an*) and conjunctions (*and*, *so*, *but*). Industrial grade packages can effectively remove such topics so that they are not included in the results.

Additional Material

LDACGS Source Code

```
class LDACGS:
    """Do LDA with Gibbs Sampling."""

    def __init__(self, n_topics, alpha=0.1, beta=0.1):
        """Initialize system parameters."""
        self.n_topics = n_topics
        self.alpha = alpha
        self.beta = beta

    def buildCorpus(self, filename, stopwords_file=None):
        """Read the given filename and build the vocabulary."""
        with open(filename, 'r') as infile:
            doclines = [line.rstrip().lower().split(' ') for line in infile]
        n_docs = len(doclines)
        self.vocab = list({v for doc in doclines for v in doc})
        if stopwords_file:
            with open(stopwords_file, 'r') as stopfile:
                stops = stopfile.read().split()
            self.vocab = [x for x in self.vocab if x not in stops]
            self.vocab.sort()
        self.documents = []
        for i in range(n_docs):
            self.documents.append({})
            for j in range(len(doclines[i])):
                if doclines[i][j] in self.vocab:
                    self.documents[i][j] = self.vocab.index(doclines[i][j])

    def initialize(self):
        """Initialize the three count matrices."""
        self.n_words = len(self.vocab)
        self.n_docs = len(self.documents)

        # Initialize the three count matrices.
        # The (i,j) entry of self.nmz is the number of words in document i ↩
            assigned to topic j.
        self.nmz = np.zeros((self.n_docs, self.n_topics))
        # The (i,j) entry of self.nzw is the number of times term j is assigned↩
             to topic i.
        self.nzw = np.zeros((self.n_topics, self.n_words))
        # The (i)-th entry is the number of times topic i is assigned in the ↩
            corpus.
        self.nz = np.zeros(self.n_topics)

        # Initialize the topic assignment dictionary.
        self.topics = {} # key-value pairs of form (m,i):z
```

```
        for m in range(self.n_docs):
            for i in self.documents[m]:
                # Get random topic assignment, i.e. z = ...
                # Increment count matrices
                # Store topic assignment, i.e. self.topics[(m,i)]=z
                raise NotImplementedError("Problem 3 Incomplete")

    def sample(self,filename, burnin=100, sample_rate=10, n_samples=10, ↩
        stopwords=None):
        self.buildCorpus(filename, stopwords)
        self.initialize()
        self.total_nzw = np.zeros((self.n_topics, self.n_words))
        self.total_nmz = np.zeros((self.n_docs, self.n_topics))
        self.logprobs = np.zeros(burnin + sample_rate*n_samples)
        for i in range(burnin):
            # Sweep and store log likelihood.
            raise NotImplementedError("Problem 5 Incomplete")
        for i in range(n_samples*sample_rate):
            # Sweep and store log likelihood
            raise NotImplementedError("Problem 5 Incomplete")
            if not i % sample_rate:
                # accumulate counts
                raise NotImplementedError("Problem 5 Incomplete")

    def phi(self):
        phi = self.total_nzw + self.beta
        self._phi = phi / np.sum(phi, axis=1)[:,np.newaxis]

    def theta(self):
        theta = self.total_nmz + self.alpha
        self._theta = theta / np.sum(theta, axis=1)[:,np.newaxis]

    def topterms(self,n_terms=10):
        self.phi()
        self.theta()
        vec = np.atleast_2d(np.arange(0,self.n_words))
        topics = []
        for k in range(self.n_topics):
            probs = np.atleast_2d(self._phi[k,:])
            mat = np.append(probs,vec,0)
            sind = np.array([mat[:,i] for i in np.argsort(mat[0])]).T
            topics.append([self.vocab[int(sind[1,self.n_words - 1 - i])] for i ↩
                in range(n_terms)])
        return topics

    def toplines(self,n_lines=5):
        lines = np.zeros((self.n_topics,n_lines))
        for i in range(self.n_topics):
```

```
            args = np.argsort(self._theta[:,i]).tolist()
            args.reverse()
            lines[i,:] = np.array(args)[0:n_lines] + 1
        return lines

    def _removeStopwords(self, stopwords):
        return [x for x in self.vocab if x not in stopwords]

    def _conditional(self, m, w):
        dist = (self.nmz[m,:] + self.alpha) * (self.nzw[:,w] + self.beta) / (↩
            self.nz + self.beta*self.n_words)
        return dist / np.sum(dist)

    def _sweep(self):
        for m in range(self.n_docs):
            for i in self.documents[m]:
                # Retrieve vocab index for i-th word in document m.
                # Retrieve topic assignment for i-th word in document m.
                # Decrement count matrices.
                # Get conditional distribution.
                # Sample new topic assignment.
                # Increment count matrices.
                # Store new topic assignment.
                raise NotImplementedError("Problem 4 Incomplete")

    def _loglikelihood(self):
        lik = 0

        for z in range(self.n_topics):
            lik += np.sum(gammaln(self.nzw[z,:] + self.beta)) - gammaln(np.sum(↩
                self.nzw[z,:] + self.beta))
            lik -= self.n_words * gammaln(self.beta) - gammaln(self.n_words*↩
                self.beta)

        for m in range(self.n_docs):
            lik += np.sum(gammaln(self.nmz[m,:] + self.alpha)) - gammaln(np.sum↩
                (self.nmz[m,:] + self.alpha))
            lik -= self.n_topics * gammaln(self.alpha) - gammaln(self.n_topics*↩
                self.alpha)

        return lik
```

8 Speech Recognition using CDHMMs

Lab Objective: *Understand how speech recognition via CDHMMs works, and implement a simplified speech recognition system.*

8.0.1 Continuous Density Hidden Markov Models

Some of the most powerful applications of Hidden Markov Models, speech and voice recognition, result from allowing the observation space to be continuous instead of discrete. These are called Continuous Density Hidden Markov Models (CDHMMs), and they have two standard formulations: Gaussian HMMs and Gaussian Mixture Model HMMs (GMMHMMs). In fact, the former is a special case of the latter, so we will just discuss GMMHMMs in this lab.

In order to understand GMMHMMs, we need to be familiar with a particular continuous, multivariate distribution called a *mixture of Gaussians*. A mixture of Gaussians is a distribution composed of several Gaussian (or Normal) distributions with corresponding weights. Such a distribution is parameterized by the number of mixture components K, the dimension N of the normal distributions involved, a collection of component weights $\{c_1, \dots, c_K\}$ that are nonnegative and sum to 1, and a collection of mean and covariance parameters $\{(\mu_1, \Sigma_1), \dots, (\mu_K, \Sigma_K)\}$ for each Gaussian component. To sample from a mixture of Gaussians, one first chooses the mixture component i according to the probability weights $\{c_1, \dots, c_K\}$, and then one samples from the normal distribution $\mathcal{N}(\mu_k, \Sigma_k)$. The probability density function for a mixture of Gaussians is given by

$$p(\mathbf{z}|\theta) = \sum_{k=1}^{K} c_k N(\mathbf{z};\, \mu_k, \Sigma_k),$$

where $N(\cdot;\, \mu_k, \Sigma_k)$ denotes the probability density function for the normal distribution $\mathcal{N}(\mu_k, \Sigma_k)$. See Figure 8.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

In a GMMHMM, we seek to model a hidden state sequence $\{X_1, \dots, X_T\}$ and a corresponding observation sequence $\{Z_1, \dots, Z_T\}$, just as with discrete HMMs. The major difference, of course, is that each observation $\mathbf{z}_t$ is a real-valued vector of length M distributed according to a mixture of Gaussians with K components. The parameters for such a model include the initial state distribution π and the state transition matrix A (just as with discrete HMMs). Additionally, for each state

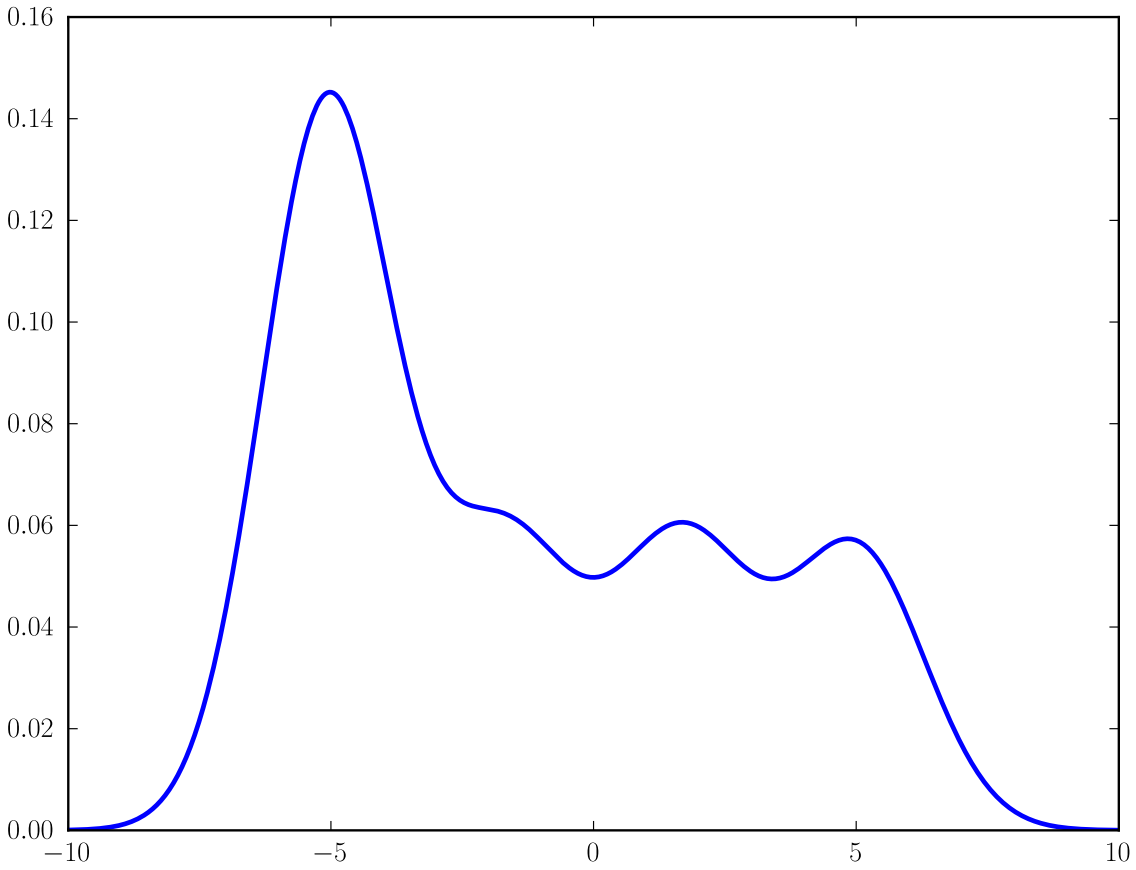


Figure 8.1: The probability density function of a mixture of Gaussians with four components.

$i = 1, \ldots, N$, we have component weights $\{c_{k,1}, \ldots, c_{i,K}\}$, component means $\{\mu_{k,1}, \ldots, \mu_{k,K}\}$, and component covariance matrices $\{\Sigma_{k,1}, \ldots, \Sigma_{i,K}\}$.

Let's define a full GMMHMM with $N = 2$ states, $M = 3$, and $K = 3$ components.

```
>>> import numpy as np
>>> A = np.array([[.65, .35], [.15, .85]])   # state transition matrix
>>> pi = np.array([.8, .2])   # initial state distribution
>>> weights = np.array([[.7, .2, .1], [.1, .5, .4]])
>>> means1 = np.array([[0., 17., -4.], [5., -12., -8.], [-16., 22., 2.]])
>>> means2 = np.array([[-5., 3., 23.], [-12., -2., 14.], [15., -32., 0.]])
>>> means = np.array([means1, means2])
>>> covars1 = np.array([5*np.eye(3), 7*np.eye(3), np.eye(3)])
>>> covars2 = np.array([10*np.eye(3), 3*np.eye(3), 4*np.eye(3)])
>>> covars = np.array([covars1, covars2])
>>> gmmhmm = [A, weights, means, covars, pi]
```

Once we have a GMMHMM, we can randomly choose the first state based on the initial state distribution π. As explained above, to sample from a GMMHMM, we draw a sample from one of the Gaussians $\mathcal{N}(\mu_i, \Sigma_i)$, with component i chosen according to the probably weights $\{c_1, \ldots, c_M\}$. We

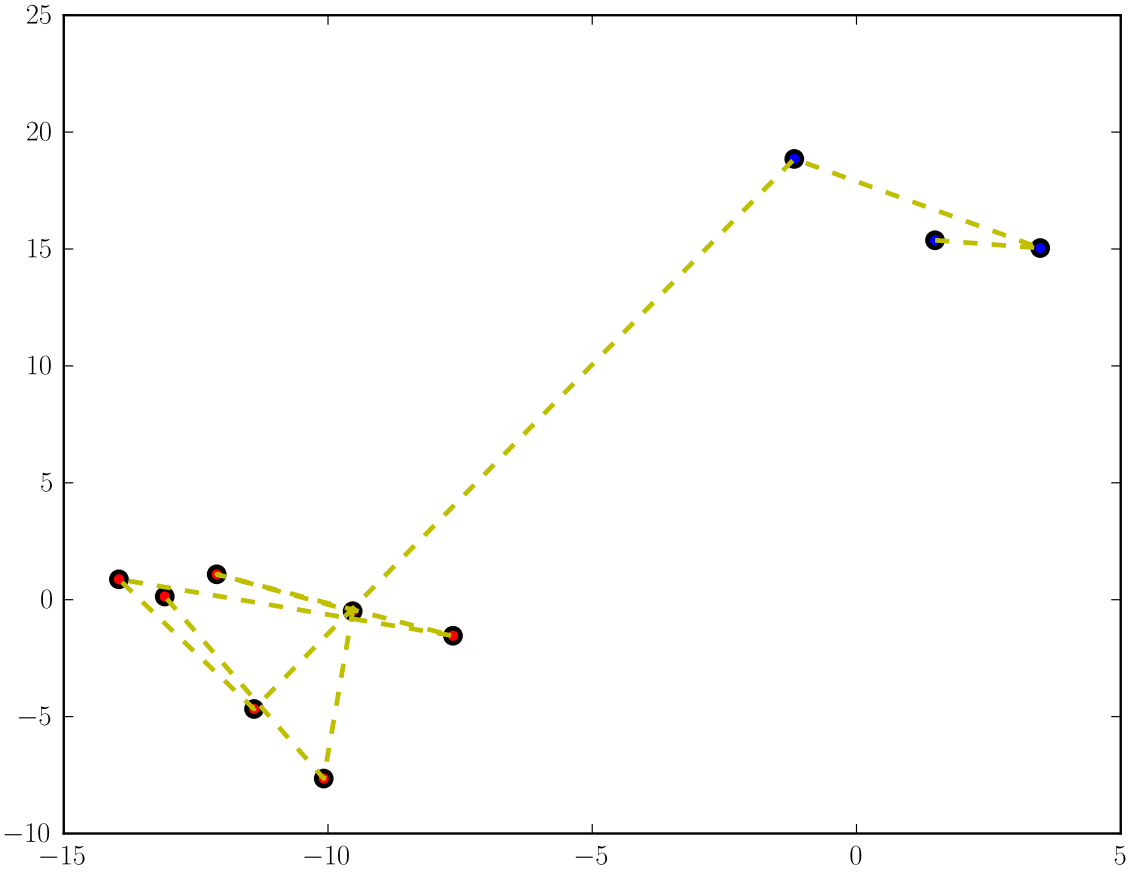


Figure 8.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

can draw a random sample from the GMMHMM corresponding to the second state. Once we have the sample, we use the transition matrix A to determine the second state.

```
# choose initial state
>>> state = np.argmax(np.random.multinomial(1, pi))

# randomly sample
>>> sample_component = np.argmax(np.random.multinomial(1, weights[state,:]))
>>> sample = np.random.multivariate_normal(means[state, sample_component, :], ↩
    covars[state, sample_component, :, :])
```

Figure 8.2 shows an observation sequence generated from a GMMHMM with one mixture component and two states.

Problem 1. Write a function which accepts a GMMHMM in the format above as well as an integer n_sim, and which simulates the GMMHMM process, generating n_sim different observations. Do so by implementing the following function declaration.

```
def sample_gmmhmm(gmmhmm, n_sim):
    """
    Simulate sampling from a GMMHMM.

    Returns
    -------
    states : ndarray of shape (n_sim,)
        The sequence of states
    obs : ndarray of shape (n_sim, K)
        The generated observations (column vectors of length K)
    """
    pass
```

Test your function by running it on the gmmhmm given in the example, with $n\_sim = 900$. Use `sklearn.decomposition.PCA` with 2 components to plot the observations in two-dimensional space. Color the observations by state. How many clusters do you see?

Note, M is the same M that defines the gmmhmm and is identified through the gmmhmm, so you don't need to set it.

The classic problems for which we normally use discrete observation HMMs can also be solved by using CDHMMs, though with continuous observations it is much more difficult to keep things numerically stable. We will not have you implement any of the three problems for CDHMMs yourself; instead, you will use a stable module we will provide for you. Note, however, that the techniques for solving these problems are still based on the forward-backward algorithm; the implementation may be trickier, but the mathematical ideas are virtually the same as those for discrete HMMs.

Speech Recognition and Hidden Markov Models

Hidden Markov Models are the basis of modern speech recognition systems. However, a fair amount of signal processing must precede the HMM stage, and there are other components of speech recognition, such as language models, that we will not address in this lab.

The basic signal processing and HMM stages of the speech recognition system that we develop in this lab can be summarized as follows: The audio to be processed is divided into small frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these inervals. We can then take this framed signal and, through a series of transformations, represent it by mel-frequency cepstral coefficients (MFCCs), keeping only the first M (say $M = 10$). Viewing these MFCCs as continuous observations in $\mathbb{R}^M$, we can train a GMMHMM on sequences of MFCCs for a given word, spoken multiple times. Doing this for several words, we have a collection of GMMHMMs, one for each word. Given a new speech signal, after framing and decomposing it into its MFCC array, we can score the signal against each GMMHMM, returning the word whose GMMHMM scored the highest.

Industrial-grade speech recognition systems do not train a GMMHMM for each word in a vocabulary (that would be ludicrous for a large vocabulary), but rather on *phonemes*, or distinct

sounds. The English language has 44 phonemes, yielding 44 different GMMHMMs. As you could imagine, this greatly facilitates the problem of speech recognition. Each and every word can be represented by some combination of these 44 distinct sounds. By correctly classifying a signal by its phonemes, we can determine what word was spoken. Doing so is beyond the scope of this lab, so we will simply train GMMHMMs on five words/phrases: biology, mathematics, political science, psychology, and statistics.

Problem 2. Samples.zip contains 30 recordings for each of the words/phrases mathematics, biology, political science, psychology, and statistics. Remove the files that end in 00 (eg. Biology00.wav). These audio samples are 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. Load the recordings into Python using *scipy.io.wavfile.read*.

Extract the MFCCs from each sample using code from the file MFCC.py. Store the MFCCs for each word in a separate list. You should have five lists, each containing 30 MFCC arrays, corresponding to each of the five words under consideration.

For a specific word, given enough distinct samples of that word (decomposed into MFCCs), we can train a GMMHMM. Recall, however, that the training procedure does not always produce a very effective model, as it can get stuck in a poor local minimum. To combat this, we will train 10 GMMHMMs for each word (using a different random initialization of the parameters each time) and keep the model with the highest log-likelihood.

For training, we will use the file we have provided called `gmmhmm.py`, as this is a stable implementation of GMMHMM algorithms. To facilitate random restarts, we need a function to provide initializations for the initial state distribution and the transition matrix.

Let `samples` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Using a function `initialize()` that returns a random initial state distribution and row-stochastic transition matrix, we can train a GMMHMM with 5 states and 3 mixture components and view its log-likelihood as follows:

```
>>> import gmmhmm as hmm
>>> startprob, transmat = initialize(5)
>>> model = hmm.GMMHMM(n_components=5, n_mix=3, transmat=transmat, startprob=↩
    startprob, cvtype='diag')
>>> # these values for covars_prior and var should work well for this problem
>>> model.covars_prior = 0.01
>>> model.fit(samples, init_params='mc', var=0.1)
>>> print(model.logprob)
```

Problem 3. Partition each list of MFCCs into a training set of 20 samples, and a test set of the remaining 10 samples. Using the training sets, train a GMMHMM on each of the words from the previous problem with at least 10 random restarts (reinitializing and creating a new model). Use $n\_components = 5$ and $initialize(5)$. Keep the best model for each word (the one with the highest log-likelihood). This process may take up to a couple of hours. Since you

will not want to run this more than once, you will want to save the best model for each word to disk using the pickle module so that you can use it later.

Given a trained model, we would like to compute the log-likelihood of a new sample. Letting `obs` be an array of MFCCs for a speech sample we do this as follows:

```
>>> score = model.score(obs)
```

We classify a new speech sample by scoring it against each of the 5 trained GMMHMMs, and returning the word corresponding to the GMMHMM with the highest score.

Problem 4. Classify the 10 test samples for each word.

How does your system perform? Which words are the hardest to correctly classify? Make a dictionary containing the accuracy of the classification of your five testing sets. Specifically, the words/phrases will be the keys, and the values will be the percent accuracy. For example, to find the accuracy for the biology model score (`model.score(sample)`) each model on all 10 samples in the biology test set. The predicted class for each sample is the class of the model with the highest score. The accuracy of the biology model is the number of words in the biology test set that the biology model gave the highest score for over ten, since there were 10 words in the test set.

9 Kalman Filter

Lab Objective: *Understand how to implement the standard Kalman Filter. Apply to the problem of projectile tracking.*

Measured observations are often prone to significant noise, due to restrictions on measurement accuracy. For example, most commercial GPS devices can provide a good estimate of geolocation, but only within a dozen meters or so. A Kalman filter is an algorithm that takes a sequence of noisy observations made over time and attempts to get rid of the noise, producing more accurate estimates than the original observations. To do this, the algorithm needs information about the system being observed.

Consider the problem of tracking a projectile as it travels through the air. Short-range projectiles approximately trace out parabolas, but a sensor that is recording measurements of the projectile's position over time will likely show a path that is much less smooth. Because we know something about the laws of physics, we can filter out the noise in the measurements using basic Newtonian mechanics, recovering a more accurate estimate of the projectile's trajectory. In this lab, we will simulate measurements of a projectile and implement a Kalman filter to estimate the complete trajectory of the projectile.

Linear Dynamical Systems

The standard Kalman filter assumes that: (1) we have a linear dynamical system, (2) the state of the system evolves over time with some noise, and (3) we receive noisy measurements about the state of the system at each iteration. More formally, letting $\mathbf{x}_k$ denote the state of the system at time k, we have

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + G_k\mathbf{u}_k + \mathbf{w}_k \tag{9.1}$$

where F_k is a state-transition model, G_k is a control-input model, $\mathbf{u}_k$ is a control vector, and $\mathbf{w}_k$ is the noise present in state k. This noise is assumed to be drawn from a multivariate Gaussian distribution with zero mean and covariance matrix Q_k. The control-input model and control vector allow the assumption that the state can be additionally influenced by some other factor than the linear state-transition model.

We further assume that the states are "hidden," and we only get the noisy observations

$$\mathbf{z}_k = H_k\mathbf{x}_k + \mathbf{v}_k \tag{9.2}$$

where H_k is the observation model mapping the state space to the observation space, and $\mathbf{v}_k$ is the observation noise present at iteration k. As with the aforementioned error, we assume that this noise is drawn from a multivariate Gaussian distribution with zero mean and covariance matrix R_k.

The dynamics stated above are all taken to be linear. Thus, for our purposes, the operators F_k, G_k, and H_k are all matrices, and $\mathbf{x}_k$, $\mathbf{u}_k$, $\mathbf{z}_k$, and $\mathbf{v}_k$ are all vectors.

We will assume that the transition and observation models, the control vector, and the noise covariances are constant, i.e. for each k, we will replace $F_k, H_k, \mathbf{u}_k, Q_k$, and R_k with $F, H, \mathbf{u}, Q$, and R. We will also assume that $G = I$ is the identity matrix, so it can safely be ignored.

Problem 1. Begin implementing a `KalmanFilter` class by writing an initialization method that stores the transition and observation models, noise covariances, and control vector. We provide an interface below:

```
class KalmanFilter(object):
    def __init__(self,F,Q,H,R,u):
        """
        Initialize the dynamical system models.

        Parameters
        ----------
        F : ndarray of shape (n,n)
            The state transition model.
        Q : ndarray of shape (n,n)
            The covariance matrix for the state noise.
        H : ndarray of shape (m,n)
            The observation model.
        R : ndarray of shape (m,m)
            The covariance matric for observation noise.
        u : ndarray of shape (n,)
            The control vector.
        """
        pass
```

We now derive the linear dynamical system parameters for a projectile traveling through $\mathbb{R}^2$ undergoing a constant downward gravitational force of 9.8 m/s^2. The relevant information needed to describe how the projectile moves through space is its position and velocity. Thus, our state vector has the form

$$\mathbf{x} = \begin{pmatrix} s_x \\ s_y \\ V_x \\ V_y \end{pmatrix},$$

where s_x and s_y give the x and y coordinates of the position (in meters), and V_x and V_y give the horizontal and vertical components of the velocity (in meters per second), respectively.

How does the system evolve from one time step to the next? Assuming each time step is 0.1

seconds, it is easy enough to calculate the new position:

$$s'_x = s_x + 0.1V_x$$
$$s'_y = s_y + 0.1V_y.$$

Further, since the only force acting on the projectile is gravity (we are ignoring things like wind resistance), the horizontal velocity remains constant:

$$V'_x = V_x.$$

The vertical velocity, however, does change due to the effects of gravity. From basic Newtonian mechanics, we have

$$V'_y = V_y - 0.1 \cdot 9.8.$$

In summary, over one time step, the state evolves from $\mathbf{x}$ to $\mathbf{x}'$, where

$$\mathbf{x}' = \begin{pmatrix} s_x + 0.1V_x \\ s_y + 0.1V_y \\ V_x \\ V_y - 0.98 \end{pmatrix}.$$

From this equation, you can extract the state transition model F and the control vector u.

We now turn our attention to the observation model. Imagine that a radar sensor captures (noisy) measurements of the projectile's position as it travels through the air. At each time step, the radar transmits the observation $z = (z_x, z_y)$ given by

$$z_x = s_x + v_x$$
$$z_y = s_y + v_y,$$

where (v_x, v_y) is a noise vector assumed to be drawn from a multivariate Gaussian with mean zero and some known covariance. These equations indicate the appropriate choice of observation model.

Problem 2. Work out the transition and observation models F and H, along with the control vector $\mathbf{u}$, corresponding to the projectile. Assume that the noise covariances are given by

$$Q = 0.1 \cdot I_4$$
$$R = 5000 \cdot I_2.$$

Instantiate a `KalmanFilter` object with these values.

We now wish to simulate a sequence of states and observations from the dynamical system. In addition to the system parameters, we need an initial state $\mathbf{x}_0$ to get started. Computing the subsequent states and observations is simply a matter of following equations 9.1 and 9.2.

Problem 3. Add a method to your `KalmanFilter` class to generate a state and observation sequence by evolving the system from a given initial state (the function `numpy.random.multivariate_normal` will be useful). To do this, implement the following:

```
def evolve(self,x0,N):
    """
    Compute the first N states and observations generated by the Kalman ↩
```

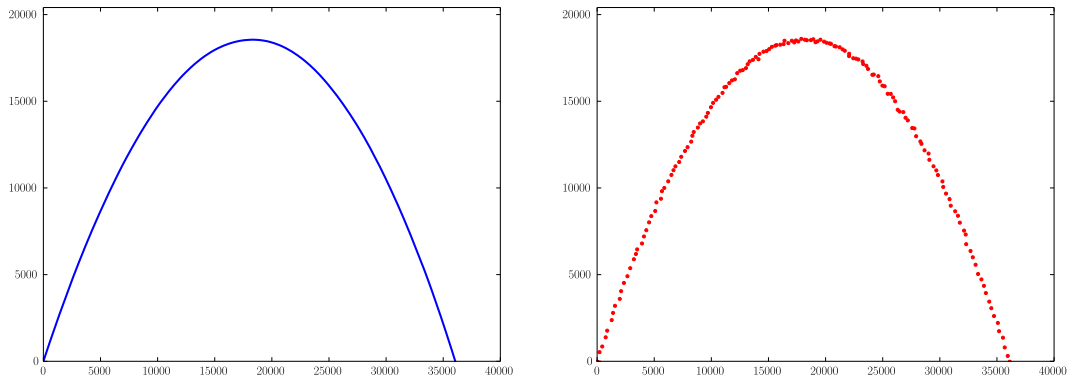


Figure 9.1: State sequence (left) and sampling of observation sequence (right).

```
    system.

Parameters
----------
x0 : ndarray of shape (n,)
    The initial state.
N : integer
    The number of time steps to evolve.

Returns
-------
states : ndarray of shape (n,N)
    States 0 through N-1, given by each column.
obs : ndarray of shape (m,N)
    Observations 0 through N-1, given by each column.
"""
pass
```

Simulate the true and observed trajectory of a projectile with initial state

$$\mathbf{x}_0 = \begin{pmatrix} 0 \\ 0 \\ 300 \\ 600 \end{pmatrix}.$$

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the y coordinate to return to 0). Your results should qualitatively match those given in Figure 9.1.

State Estimation with the Kalman Filter

The Kalman filter is a recursive estimator that smooths out the noise in real time, estimating each current state based on the past state estimate and the current measurement. This process is done by

repeatedly invoking two steps: Predict and Update. The predict step is used to estimate the current state based on the previous state. The update step then combines this prediction with the current observation, yielding a more robust estimate of the current state.

To describe these steps in detail, we need additional notation. Let

- $\widehat{\mathbf{x}}_{n|m}$ be the state estimate at time n given only measurements up through time m; and
- $P_{n|m}$ be an error covariance matrix, measuring the estimated accuracy of the state at time n given only measurements up through time m.

The elements $\widehat{\mathbf{x}}_{k|k}$ and $P_{k|k}$ represent the state of the filter at time k, giving the state estimate and the accuracy of the estimate.

We evolve the filter recursively, as follows:

$$\begin{aligned}
&\textbf{Predict} && \widehat{\mathbf{x}}_{k|k-1} = F\widehat{\mathbf{x}}_{k-1|k-1} + \mathbf{u} \\
& && P_{k|k-1} = FP_{k-1|k-1}F^T + Q \\
&\textbf{Update} && \tilde{\mathbf{y}}_k = \mathbf{z}_k - H\widehat{\mathbf{x}}_{k|k-1} \\
& && S_k = HP_{k|k-1}H^T + R \\
& && K_k = P_{k|k-1}H^T S_k^{-1} \\
& && \widehat{\mathbf{x}}_{k|k} = \widehat{\mathbf{x}}_{k|k-1} + K_k\tilde{\mathbf{y}}_k \\
& && P_{k|k} = (I - K_kH)P_{k|k-1}
\end{aligned}$$

The more observations we have, the greater the accuracy of these estimates becomes (i.e the norm of the accuracy matrix converges to 0).

Problem 4. Add code to your `KalmanFilter` class to estimate a state sequence corresponding to a given observation sequence and initial state estimate. Implement the following class method:

```
def estimate(self,x,P,z):
    """
    Compute the state estimates using the Kalman filter.
    If x and P correspond to time step k, then z is a sequence of
    observations starting at time step k+1.

    Parameters
    ----------
    x : ndarray of shape (n,)
        The initial state estimate.
    P : ndarray of shape (n,n)
        The initial error covariance matrix.
    z : ndarray of shape(m,N)
        Sequence of N observations (each column is an observation).

    Returns
    -------
    out : ndarray of shape (n,N)
        Sequence of state estimates (each column is an estimate).
```

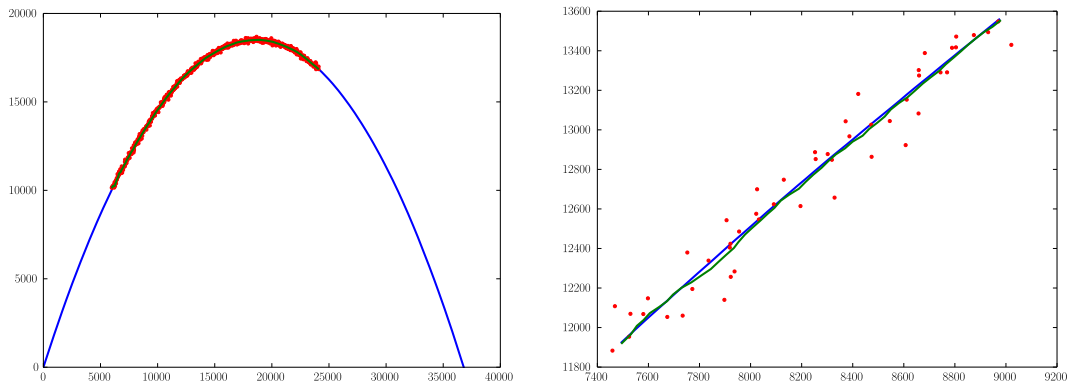


Figure 9.2: State estimates together with observations and true state sequence (detailed view on the right).

```
    """
    pass
```

Returning to the projectile example, we now assume that our radar sensor has taken observations from time steps 200 through 800 (take the corresponding slice of the observations produced in Problem 3). Using these observations, we seek to estimate the corresponding true states of the projectile. We must first come up with a state estimate $\widehat{\mathbf{x}}_{200}$ for time step 200, and then feed this into the Kalman filter to obtain estimates $\widehat{\mathbf{x}}_{201}, \ldots, \widehat{\mathbf{x}}_{800}$.

Problem 5. Calculate an initial state estimate $\widehat{\mathbf{x}}_{200}$ as follows: For the horizontal and vertical positions, simply use the observed position at time 200. For the velocity, compute the average velocity between the observations $\mathbf{z}_k$ and $\mathbf{z}_{k+1}$ for $k = 200, \ldots, 208$, then average these 9 values and take this as the initial velocity estimate. (Hint: the NumPy function `diff` is useful here.)

Using the initial state estimate, $P_{200} = 10^6 \cdot Q$, and your Kalman filter, compute the next 600 state estimates, i.e. compute $\widehat{\mathbf{x}}_{201}, \ldots, \widehat{\mathbf{x}}_{800}$. Plot these state estimates as a smooth green curve together with the radar observations (as red dots) and the entire true state sequence (as a blue curve). Zoom in to see how well it follows the true path. Your plots should be similar to Figure 9.2.

In the absence of observations, we can still estimate some information about the state of the system at some future time. We can do this by recognizing that the expected state noise $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$ at any time k. Thus, given a current state estimate $\widehat{\mathbf{x}}_{n|m}$ using only measurements up through time m, the expected state at time $n+1$ is

$$\widehat{\mathbf{x}}_{n+1|m} = F\widehat{\mathbf{x}}_{n|m} + \mathbf{u}$$

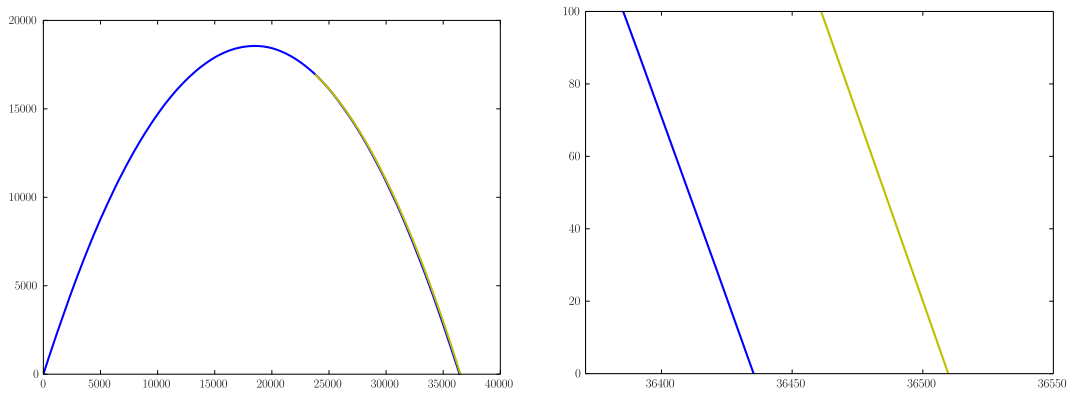


Figure 9.3: Predicted vs. actual point of impact (detailed view on right).

Problem 6. Add a function to your class that predicts the next k states given a current state estimate but in the absence of observations. Do so by implementing the following function:

```
def predict(self,x,k):
    """
    Predict the next k states in the absence of observations.

    Parameters
    ----------
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of states to predict.

    Returns
    -------
    out : ndarray of shape (n,k)
        The next k predicted states.
    """
    pass
```

We can use this prediction routine to estimate where the projectile will hit the surface.

Problem 7. Using the final state estimate $\widehat{\mathbf{x}}_{800}$ that you obtained in Problem 5, predict the future states of the projectile until it hits the ground. Predicting approximately the next 450 states should be sufficient.

Plot the actual state sequence together with the predicted state sequence (as a yellow curve), and observe how near the prediction is to the actual point of impact. Your results should be similar to those shown in Figure 9.3.

In the absence of observations, we can also reverse the system and iterate backward in time to infer information about states of the system prior to measured observations. The system is reversed by

$$\mathbf{x}_k = F^{-1}(\mathbf{x}_{k+1} - \mathbf{u} - \boldsymbol{\varepsilon}_{k+1}).$$

Considering again that $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$ at any time k, we can ignore this term, simplifying the recursive estimation backward in time.

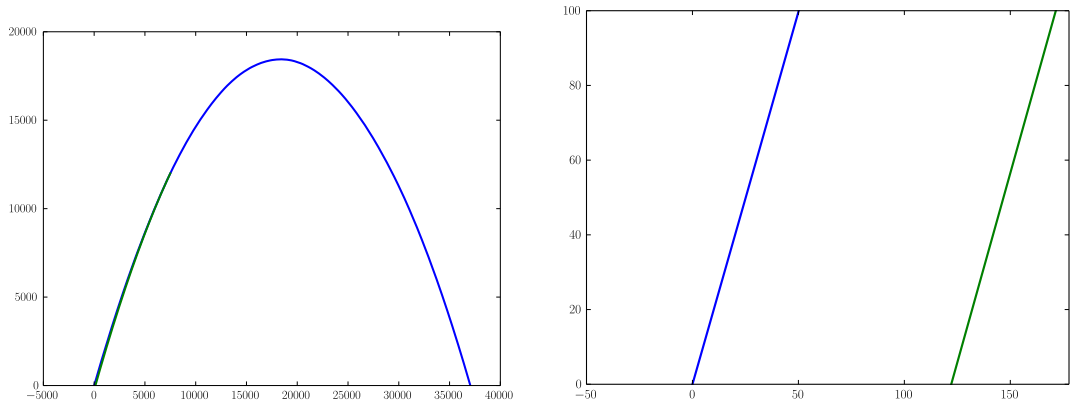


Figure 9.4: Predicted vs. actual point of origin (detailed view on right).

Problem 8. Add a function to you class that rewinds the system from a given state estimate, returning predictions for the previous states. Do so by implementing the following function:

```
def rewind(self,x,k):
    """
    Predict the k states preceding the current state estimate x.

    Parameters
    ----------
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of preceding states to predict.

    Returns
    -------
    out : ndarray of shape (n,k)
        The k preceding predicted states.
    """
    pass
```

Returning to the projectile example, we can now predict the point of origin.

Problem 9. Using your state estimate $\widehat{\mathbf{x}}_{250}$, predict the point of origin of the projectile along with all states leading up to time step 250. Note that you may have to take a few extra time steps to predict the point of origin. (The point of origin is the first point along the trajectory where the y coordinate is 0.) Plot these predicted states (in cyan) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 9.4.

Repeat the prediction starting with $\widehat{\mathbf{x}}_{600}$. Compare to the previous results. Which is better? Why?

10 ARMA Models

Lab Objective: *$ARMA(p, q)$ models combine autoregressive and moving-average models in order to forecast future observations using time-series. In this lab, we will build an $ARMA(p, q)$ model to analyze and predict future weather data and then compare this model to statsmodels built-in ARMA package as well as the VARMAX package. Then we will forecast macroeconomic data as well as the future height of the Rio Negro.*

Time Series

A time series is any discrete-time stochastic process. In other words, it is a sequence of random variables, $\{Z_t\}_{t=1}^T$, that are determined by their time t. We let the realization of the time series $\{Z_t\}_{t=1}^T$ be denoted by $\{z_t\}_{t=1}^T$. Examples of time series include heart rate readings over time, pollution readings over time, stock prices at the closing of each day, and air temperature. Often when analyzing time series, we want to forecast future data, such as what will the stock price of a company will be in a week and what will the temperature be in 10 days.

ARMA(p, q) Models

One way to forecast a time series is using an ARMA model. The *Wold Theorem* says that any covariance-stationary time series can be well approximated with an ARMA model. An ARMA(p, q) model combines an autoregressive model of order p and a moving average model of order q on a time series $\{Z_t\}_{t=1}^T$. The model itself is a discrete-time stochastic process $(Z_t)_{t\in\mathbb{Z}}$ satisfying the equation

$$Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i Z_{t-i}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} \tag{10.1}$$

where each $\boldsymbol{\varepsilon_t}$ is an identically-distributed Gaussian variable with mean 0 and constant covariance Σ, $\mathbf{c} \in \mathbb{R}^n$, and Φ_i and Θ_j are in $M_n(\mathbb{R})$.

AR(p) Models

An AR(p) model works similar to a weighted random walk. Recall that in a random walk, the current position depends on the immediate past position. In the autogregressive model, the current data point in the time series depends on the past p data points. However, the importance of each of the past p data points is not uniform. With an error term to represent white noise and a constant term to adjust the model along the y-axis, we can model the stochastic process with the following equation:

$$Z_t = \mathbf{c} + \sum_{i=1}^{p} \Phi_i Z_{t-i} + \boldsymbol{\varepsilon_t} \tag{10.2}$$

If there is a high correlation between the current and previous values of the time series, then the AR(p) model is a good representation of the data, and thus the ARMA(p, q) model will most likely be a good representation. The coefficients $\{\Phi_i\}_{i=1}^{p}$ are larger when the correlation is stronger.

In this lab, we will be using weather data from Provo, Utah[1]. To check that the data can be represented well, we need to look at the correlation between the current and previous values.

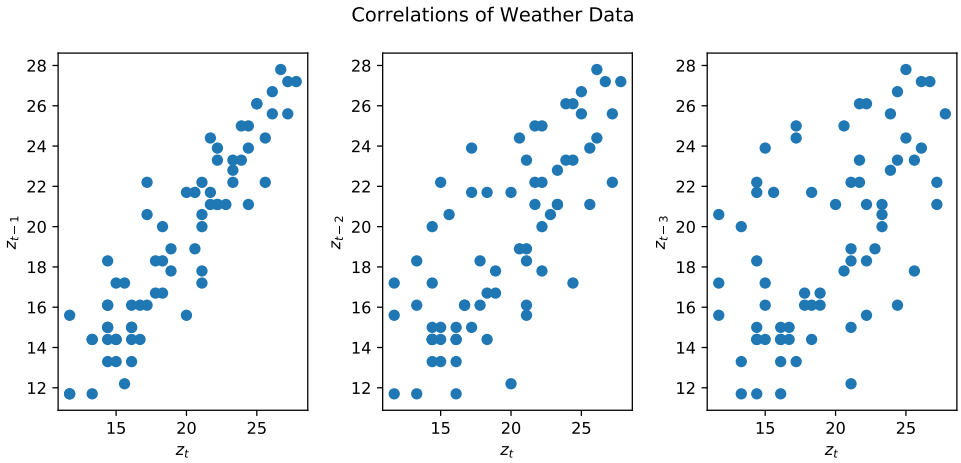


Figure 10.1: These graphs show that the weather data is correlated to its previous values. The correlation is weaker in each graph successively, showing that the further in the past the data is, the less correlated the data becomes.

MA(q) Models

A moving average model of order q is used to factor in the varying error of the time series. This model uses the error of the current data point and the previous data points to predict the next datapoint. Similar to an AR(p) model, this model uses a linear combination (which includes a constant term to adjust along the y-axis..

$$Z_t = \mathbf{c} + \boldsymbol{\varepsilon_t} + \sum_{i=1}^{q} \Theta_i \boldsymbol{\varepsilon_{t-i}} \tag{10.3}$$

This part of the model simulates shock effects in the time series. Examples of shock effects include volatility in the stock market or sudden cold fronts in the temperature.

[1]This data was taken from https://forecast.weather.gov/data/obhistory/metric/KPVU.html

Combining both the AR(p) and MA(q) models, we get an ARMA(p, q) model which forecasts based on previous observations and error trends in the data.

ARIMA(p, d, q) Models

Not all ARMA models are covariance stationary. However, many time series can be made covariance stationary by differencing. Let ∇Z_t represent the time series $Y_t = Z_t - Z_{t-1}$ obtained by taking a difference of the terms. If the trend is linear a first difference is usually stationary. If the trend is quadratic a second difference may be necessary $\nabla^2 Z_t = \nabla(\nabla Z_t)$. An ARIMA($p, d, q$) model is a discrete-time stochastic process $(Z_t)_{t\in\mathbb{Z}}$ satisfying the equation

$$\nabla^d Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i \mathbf{y_{t-i}}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} \tag{10.4}$$

Finding Parameters

One of the most difficult parts of using an ARMA(p, q) model is identifying the proper parameters of the model. For simplicity, at the beginning of this lab we discuss univariate ARMA models with parameters $\{\phi_i\}_{i=1}^p$, $\{\theta_i\}_{i=1}^q$, μ, and σ, where μ and σ are the mean and variance of the error. Note that $\{\phi_i\}_{i=1}^p$ and $\{\theta_i\}_{i=1}^q$ determine the order of the ARMA model.

A naive way to use an ARMA model is to choose p and q based on intuition. Figure 10.1 showed that there is a strong correlation between z_t and z_{t-1} and between z_t and z_{t-2}. The correlation is weaker between z_t and z_{t-3}. Intuition then suggests to choose $p = 2$. By looking at the correlations between the current noise with previous noise, similar to Figure 10.1, it can also be seen that there is a weak correlation between z_t and ε_t and between z_t and ε_{t-1}. Between z_t and ε_{t-2} there is no correlation. For more on how these error correlations were found, see Additional Materials. Intuition from these correlations suggests to choose $q = 1$. Thus, a naive choice for our model is an ARMA$(2, 1)$ model.

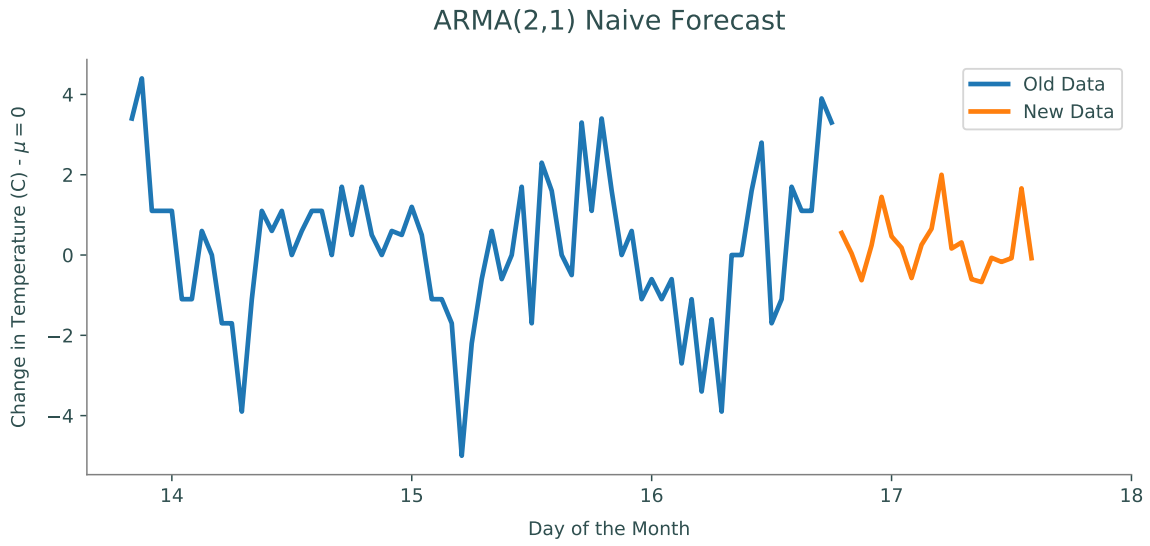


Figure 10.2: Naive forecast on `weather.npy`

Problem 1. Write a function `arma_forecast_naive()` that builds an ARMA(p,q) model where the values of $\phi_i = .5$ and $\theta_i = .1$ for all i. Let $\varepsilon_i \sim \mathcal{N}(0, 1)$ for all i.

Use your function to predict the next n values of the time series. The time series that should be used is the first difference of the time series found in the file `weather.npy`, which we denote $\{z_t\}_{t=1}^{T}$. This is done because we want the time series to be covariance stationary. The function should accept a parameter p, q, and n (the number of observations to predict). Plot the observed differences $\{z_t\}_{t=1}^{T}$ followed by your predicted observations of z_t.

The file `weather.npy` contains data on the temperature in Provo, Utah from 7:56 PM May 13, 2019 to 6:56 PM May 16, 2019, taken every hour.

Use this file to test your code. For $p = 2$, $q = 1$, and $T = 20$, your plot should look similar to Figure 10.2, however, due to the variance of the error ε_t, the plot will not look exactly like Figure 10.2. The predictions may be higher or lower on the x-axis.

Let $\Theta = \{\phi_i, \theta_j, \mu, \sigma_a^2\}$ be the set of parameters for an ARMA(p, q) model. Suppose we have a set of observations $\{z_t\}_{t=1}^{n}$. Our goal is to find the p, q, and Θ that maximize the likelihood of the ARMA model given the data. Using the chain rule, we can factorize the likelihood of the model given this data as

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} p(z_t|z_{t-1}, \dots, z_1, \Theta) \tag{10.5}$$

State Space Representation

In a general ARMA(p, q) model, the likelihood is a function of the unobserved error terms ε_t and is not trivial to compute. Simple approximations can be made, but these may be inaccurate under certain circumstances. Explicit derivations of the likelihood are possible, but tedious. However, when

the ARMA model is placed in state-space, the Kalman filter affords a straightforward, recursive way to compute the likelihood.

We demonstrate one possible state-space representation of an ARMA(p, q) model. Let $r = \max(p, q + 1)$. Define

$$\hat{\mathbf{x}}_{t|t-1} = \begin{bmatrix} x_{t-1} & x_{t-2} & \cdots & x_{t-r} \end{bmatrix}^T \tag{10.6}$$

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{10.7}$$

$$H = \begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_{r-1} \end{bmatrix} \tag{10.8}$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{10.9}$$

$$w_t \sim \text{MVN}(0, Q), \tag{10.10}$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Note that Equation 10.2 gives

$$F\hat{\mathbf{x}}_{t-1|t-2} + w_t = \begin{bmatrix} \sum_{i=1}^{r} \phi_i x_{t-i} \\ x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-(r-1)} \end{bmatrix} + \begin{bmatrix} \varepsilon_t \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{10.11}$$

$$= \begin{bmatrix} x_t & x_{t-1} & \cdots & x_{t-(r-1)} \end{bmatrix}^T \tag{10.12}$$

$$= \hat{\mathbf{x}}_{t|t-1} \tag{10.13}$$

We note that $z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu$.[2]

Then the linear stochastic dynamical system

$$\hat{\mathbf{x}}_{t+1|t} = F\hat{\mathbf{x}}_{t|t-1} + w_t \tag{10.14}$$

$$z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{10.15}$$

describes the same process as the original ARMA model.

> **Note**
>
> Equation 10.15 involves a deterministic component, namely μ. The Kalman filter theory developed in the previous lab, however, assumed $\mathbb{E}[\varepsilon_t] = 0$ for the observations $z_{t|t-1}$,. This means you should subtract off the mean μ of the error from the time series observations $z_{t|t-1}$ when using them in the predict and update steps.

[2] For a proof of this fact, see Additional Materials.

Likelihood via Kalman Filter

We assumed in Equation 10.10 that the error terms of the model are Gaussian. This means that each conditional distribution in 10.5 is also Gaussian, and is completely characterized by its mean and variance. These two quantities are easily found via the Kalman filter:

$$\text{mean} \quad H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{10.16}$$

$$\text{variance} \quad HP_{t|t-1}H^T \tag{10.17}$$

where $\hat{\mathbf{x}}_{t|t-1}$ and $P_{t|t-1}$ are found during the Predict step. Given that each conditional distribution is Gaussian, the likelihood can then be found as follows:

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} N(z_t \mid H\hat{\mathbf{x}}_{t|t-1} + \mu,\ HP_{t|t-1}H^T) \tag{10.18}$$

Problem 2. Write a function `arma_likelihood()` that returns the log-likelihood of an ARMAmodel, given a time series $\{z_t\}_{t=1}^T$. This function should accept a `file` with the observations and each of the parameters in Θ. In this case, using `weather.npy`, the time series should be the change in temperature. This means to take the first difference of the time series found in the file `weather.npy` as done in the first problem. Return the log-likelihood of the ARMA(p, q) model as a `float`.

Use the `state_space_rep()` function provided to create F, Q, and H. A `kalman()` filter has been provided to calculate the means and covariances of each observation.

(Hint: Calling the function `kalman()` on a time series will return an array whose values are $x_{k|k-1}$ and an array whose values are $P_{k|k-1}$ for each $k \leq n$. Remember that the time series should have μ subtracted when using `kalman()`.)

When done correctly, your function should match the following output:

```
>>> arma_likelihood(file='weather.npy', phis=np.array([0.9]), thetas=np.↩
    array([0]), mu=17., std=0.4)
-1375.1805469978776
```

Model Identification

Now that we can compute the likelihood of a given ARMA model, we want to find the best choice of parameters given our time series. In this lab, we define the model with the "best" choice of parameters as the model which minimizes the AIC. The benefit of minimizing the AIC is that it rewards goodness of fit while penalizing overfitting. The AIC is expressed by

$$2k\left(1 + \frac{k+1}{n-k}\right) - 2\ell(\Theta) \tag{10.19}$$

where n is the sample size, $k = p + q + 2$ is the number of parameters in the model, and $\ell(\Theta)$ is the maximum likelihood for the model class.

To compute the maximum likelihood for a model class, we need to optimize 10.18 over the space of parameters Θ. We can do so by using an optimization routine such as `scipy.optimize.minimize` on the function `arma_likelihood()` from Problem 2. Use the following code to run this routine.

```
>>> from scipy.optimize import minimize

>>> # assume p, q, and time_series are defined
>>> def f(x): # x contains the phis, thetas, mu, and std
>>>     return -1*arma_likelihood(filename, phis=x[:p], thetas=x[p:p+q], mu=x↩
    [-2],std=x[-1])
>>> # create initial point
>>> x0 = np.zeros(p+q+2)
>>> x0[-2] = time_series.mean()
>>> x0[-1] = time_series.std()
>>> sol = minimize(f,x0, method = "SLSQP")
>>> sol = sol['x']
```

This routine will return a vector `sol` where the first p values are $\{\phi_i\}_{i=1}^{p}$, the next q values are $\{\theta_i\}_{i=1}^{q}$, and the last two values are μ and σ, respectively. Note the wrapper $f(x)$ returns the negative log-likelihood. This is because `scipy.optimize.minimize` finds the *minimizer* of $f(x)$ and we are solving for the *maximum* likelihood.

To minimize the AIC, we perform *model identification*. This is choosing the order of our model, p and q, from some admissible set. The order of the model which minimizes the AIC is then the optimal model.

Problem 3. Write a function `model_identification()` that accepts a `file` containing the time series data and two integers, p and q. Return each parameter in Θ that minimizes the AIC of an ARMA(i, j) model, given that $1 \leq i \leq p$ and $1 \leq j \leq q$.

Your code should produce the following output (it may take awhile to run):

```
>>> model_identification(filename='weather.npy',p=4,q=4)
(array([ 0.7213538]), array([-0.26246426]), 0.359785001944352, ↩
    1.5568374351425505)
```

Forecasting with Kalman Filter

We now have identified the optimal ARMA(p, q) model. We can use this model to predict future states. The Kalman filter provides a straightforward way to predict future states by giving the mean and variance of the conditional distribution of future observations. Observations can be found as follows

$$z_{t+k}|z_1, \cdots, z_t \sim N(z_{t+k};\ H\hat{x}_{t+k|t} + \mu,\ HP_{t+k|t}H^T) \tag{10.20}$$

To evolve the Kalman filter, recall the predict and update rules of a Kalman filter.

Predict

$$\hat{\mathbf{x}}_{k|k-1} = F\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{u}$$
$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q$$

Update

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - H\hat{\mathbf{x}}_{k|k-1}$$
$$S_k = HP_{k|k-1}H^T + R$$
$$K_k = P_{k|k-1}H^T S_k^{-1}$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k\tilde{\mathbf{y}}_k$$
$$P_{k|k} = (I - K_kH)P_{k|k-1}$$

Achtung!

Recall that the values returned by `kalman()` are conditional on the previous observation. To compute the mean and variance of future observations, the values $x_{n|n}$ and $P_{n|n}$ MUST be computed using the update step. Once computed, only the predict step is needed to find the future means and covariances.

Problem 4. Write a function `arma_forecast()` that accepts a `file` containing a time series, the parameters for an ARMA model, and the number n of observations to forecast. Calculate the mean and covariance of the future n observations using a Kalman filter. Plot the original observations as well as the `mean` for each future observation. Plot a 95% confidence interval (2 standard deviations away from the mean) around the means of future observations. Return the means and standard deviations calculated.

(Hint: The standard deviation is the square root of the covariance calculated.)

The following code should create a plot similar to Figure 10.3.

```
>>> # Get optimal model as found in the previous problem
>>> phis, thetas, mu, std =np.array([ 0.72135856]), array([-0.26246788]), ↩
    0.35980339870105321, 1.5568331253098422)

>>> # Forecast optimal mode
>>> arma_forecast(filename='weather.npy', phis=phis, thetas=thetas, mu=mu,↩
     std=std)
```

How does this plot compare to the naive ARMA model made in Problem 1?

Statsmodel ARMA

The module `statsmodels` contains a package that includes an ARMA model class. This is accessed through ARIMA model, which stands for Autoregressive Integrated Moving Average. This class also uses a Kalman Filter to calculate the MLE. When creating an ARIMA object, initialize the variables `endog` (the data) and `order` (the order of the model). The order is of the form (p, d, q) where d is

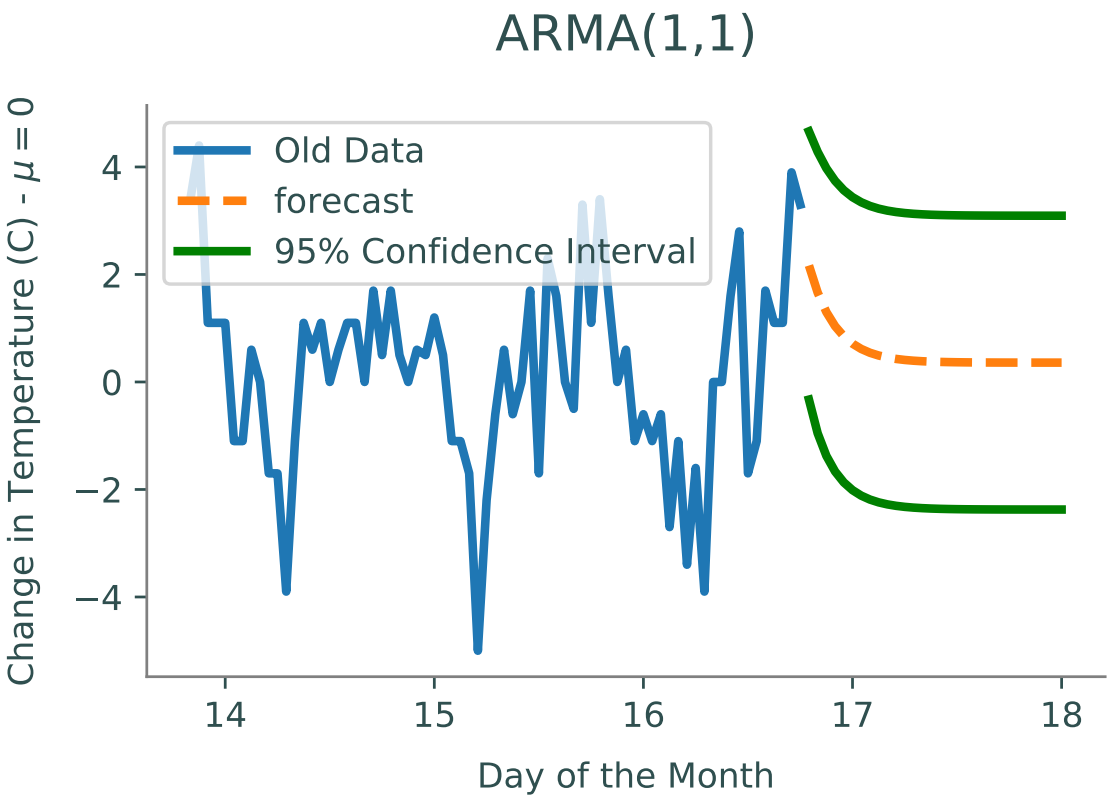


Figure 10.3: ARMA(1,1) forecast on `weather.npy`

the differences. To create an ARMA model, set $d = 0$. The object can then be fitted based on the MLE using a Kalman Filter.

```
from statsmodels.tsa.arima.model import ARIMA
# Intialize the object with weather data and order (1,1)
model = ARIMA(z,order=(p,0,q),trend='c').fit(method='innovations_mle')

# Access p and q
>>> model.specification.k_ar
p
>>> model.specification.k_ma
q
```

As in other problems, the data passed in should be the time series stationary. The AIC of an ARMA model object is saved as the attribute `aic`. Since the AIC is much faster to compute using `statsmodels`, model identification is much faster. Once a model is chosen, the method `predict` will forecast n observations, where n is the number of known observations. It will return the mean of each future observation.

```
# Predict from the beginning of the model to 30 observations in the future
model.predict(start=0,end=len(data)+30)
```

Problem 5. Write a function `sm_arma()` that accepts a `file` containing a time series, maximum integer values for p and q, and the number n of values to predict. Use `statsmodels` to perform model identification as in Problem 3, where the order of ARMA(i, j) satisfies $1 \leq i \leq p$ and $1 \leq j \leq q$. Ensure the model is fit using the MLE.

Use the optimal model to predict n future observations of the time series. Plot the original observations along with the mean of each future observations given by `statsmodels`. Return the AIC of the optimal model.

For $p = 3, q = 3$, and $n = 30$, your graph should look similar to Figure 10.4. How does this graph compare to Problem 1? Problem 4?

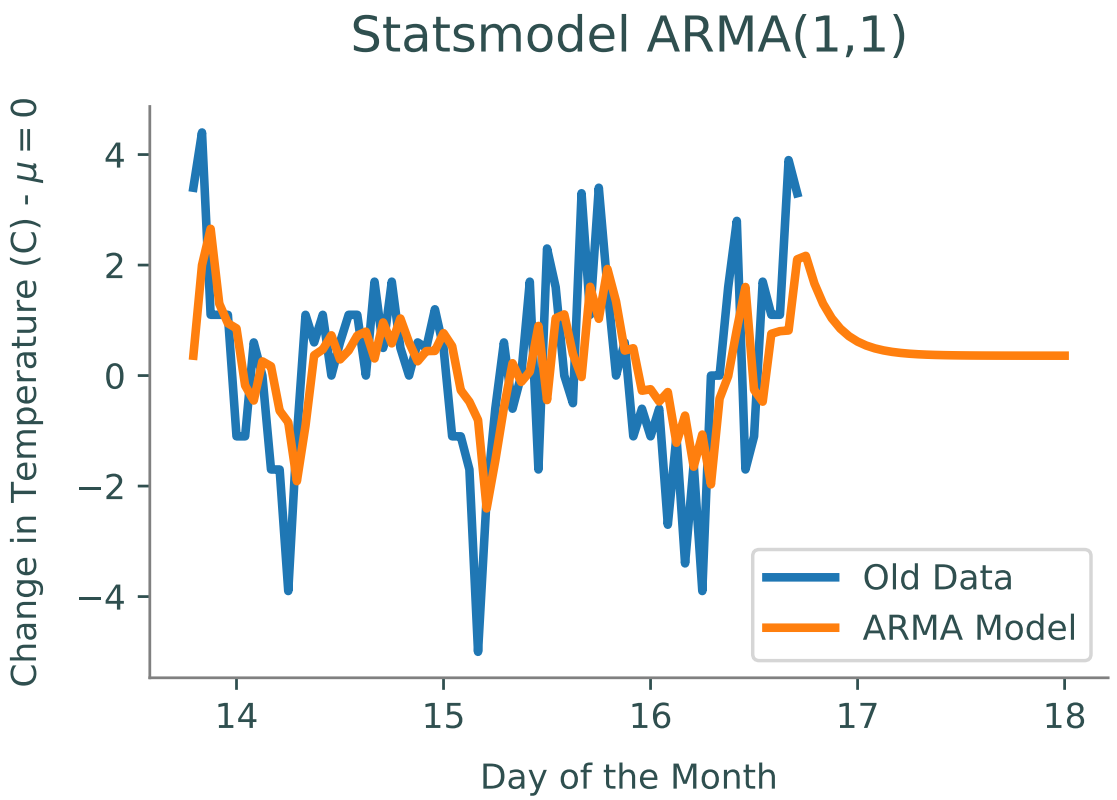


Figure 10.4: Statsmodel ARMA(3,1) forecast on `weather.npy`.

Statsmodel VARMA

Until now we have been dealing with univariate ARMA models. Multivariate ARMA models are used when we have multiple time series that can be useful in predicting one another. For example say we have two time series $z_{t,1}$ and $z_{t,2}$. The multivariate ARMA(1,1) model is as follows:

$$z_{t,1} = c_1 + \phi_{11} z_{t-1,1} + \phi_{12} z_{t-1,2} + \theta_{11} \varepsilon_{t-1,1} + \theta_{12} \varepsilon_{t-1,2} \tag{10.21}$$
$$z_{t,2} = c_1 + \phi_{21} z_{t-1,1} + \phi_{22} z_{t-1,2} + \theta_{21} \varepsilon_{t-1,1} + \theta_{22} \varepsilon_{t-1,2} \tag{10.22}$$

This can be written in matrix form as shown in equation 10.1. The module `statsmodels` contains a package that includes an VARMAX model class which can be used to create a multivariate ARMA model. This stands for Vector Autoregression Moving Average with Exogenous Regressors. An exogenous regressor is a time series that affects the model but is not affected by it. In the example below we have two time series corresponding to the price of copper and aluminum. Since aluminum is a substitute for copper, it is reasonable to assume the price of aluminum may help us predict the price of copper and vice versa. Note that when fitting a VARMAX model setting the parameter `ic` to `aic` selects parameters based on AIC criterion.

```
>>>from statsmodels.tsa.api import VARMAX
>>>import statsmodels.api as sm

>>> # Load in world copper data
>>> data = sm.datasets.copper.load_pandas().data
>>> # Create index compatible with VARMAX model
>>> idx = pd.period_range(start='1951', end='1975',freq = 'Y')
>>> data.index = idx

>>> # Initialize and fit model
>>> mod = VARMAX(data[['ALUMPRICE', 'COPPERPRICE']])
>>> mod = mod.fit(maxiter=1000, disp=False, ic = 'aic')
>>> # Predict until the price of aluminium and copper until 1985
>>> pred = mod.predict('1951','1985')

>>> # Get confidence intervals
>>> forecast_obj = mod.get_forecast('1981')
>>> all_CI = forecast_obj.conf_int(alpha=0.05)
>>> all_CI

>>> # Plot predictions against true price
>>> pred.plot()
>>> plt.plot(data['ALUMPRICE'],'r--', label = 'ALUMPRICE prediction')
>>> plt.plot(data['COPPERPRICE'],'r--',label = 'COPPERPRICE prediction')
>>> plt.legend()
>>> plt.title('VARMA Predictions for World Copper Market Dataset')
```

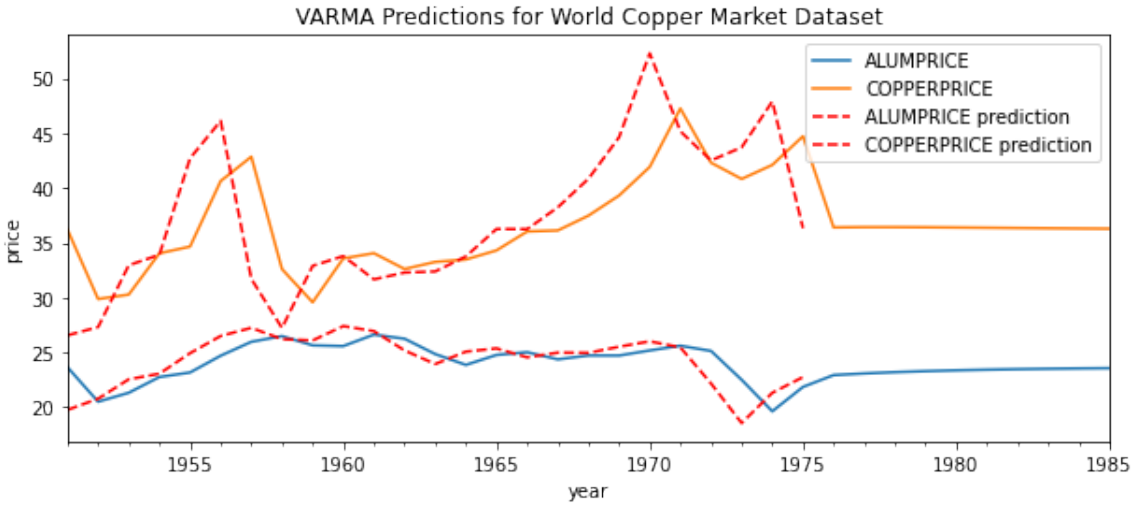


Figure 10.5: Statsmodel VAR(1) forecast.

Problem 6. Write a function `sm_varma()` that accepts start and end dates for forecasting. Use the statsmodels VARMAX class to forecast on macroeconomic data between the start and end dates. Use AIC as the criterion for model selection when fitting the model. Plot the prediction, original data and a 95% confidence interval (2 standard deviations away from the mean) around the future observations. Return the AIC of the chosen model. The plot should be similar to Figure 10.6.

The following code shows how to obtain the data.

```
>>> # Load in data
>>> df = sm.datasets.macrodata.load_pandas().data
>>> # Create DatetimeIndex
>>> dates = df[['year', 'quarter']].astype(int).astype(str)
>>> dates = dates["year"] + "Q" + dates["quarter"]
>>> dates = dates_from_str(dates)
>>> df.index = pd.DatetimeIndex(dates)
>>> # Select columns used in prediction
>>> df = df[['realgdp','realcons','realinv']]
```

In the dataset 'realgdp' is the real gross domestic product, 'realcons' is real personal consumption expenditures, and 'realinv' is real gross private domestic investment. Since personal consumption and domestic investment are components of gross domestic product it is reasonable to assume these time series will be useful in predicting one another.

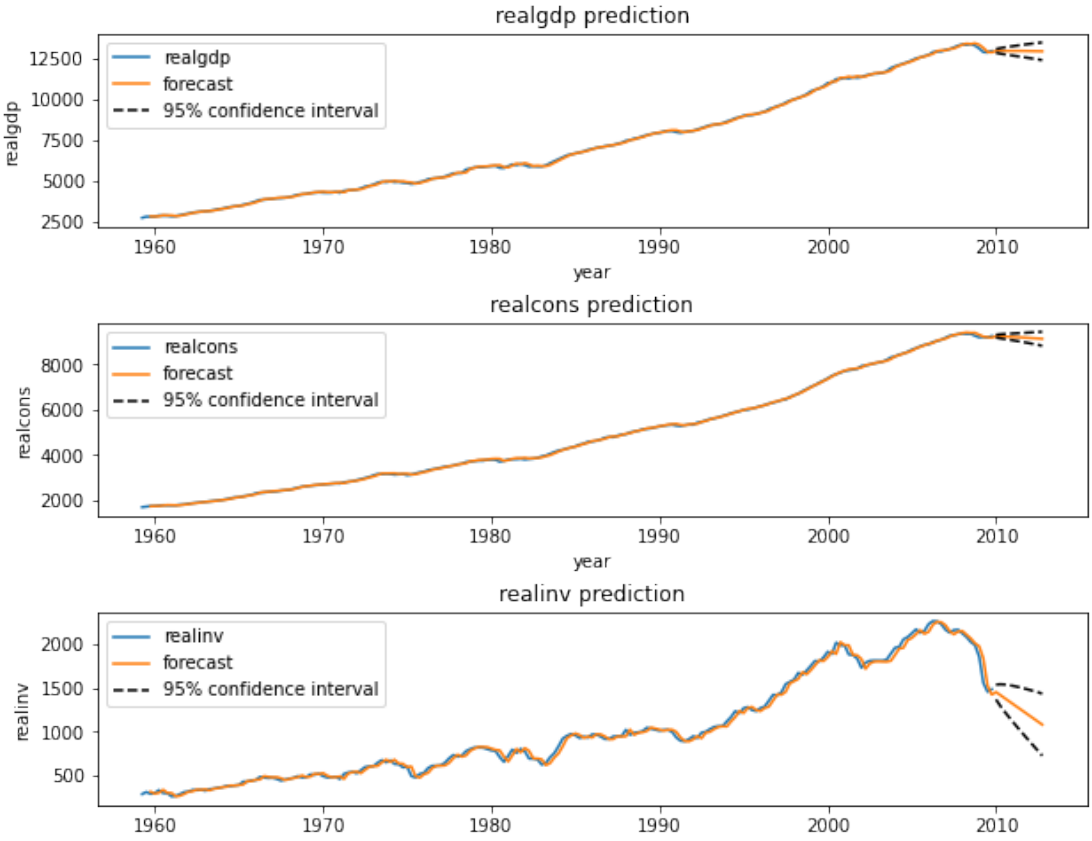


Figure 10.6: Macroeconomic data is forecasted 12 years in the future using statsmodels.

Optional

The `statsmodels` package can help us perform model identification. The method `arma_order_select_ic` will find the optimal order of the ARMA model based on certain criteria. The first parameter `y` is the data. The data must be a NumPy array, not a Pandas DataFrame. The parameter `ic` defines the criteria trying to be minimized. The method will return a dictionary, where the minimal order of each criteria can be accessed.

```
>>> import statsmodels.api as sm
>>> from statsmodel.tsa.stattools import arma_order_select_ic as order_select
>>> import pandas as pd

>>> # Get sunspot data and give DateTimeIndex
>>> sunspot = sm.datasets.sunspots.load_pandas().data
>>> sunspot.index = pd.Index(pd.date_range("1700", end="2009", freq="A-DEC"))
>>> sunspot.drop(columns = ["YEAR"],inplace = True)

>>> # Find best order where p < 5 and q < 5
>>> # Use AICc as basis for minimization
>>> order = order_select(sunspot.values,max_ar=4,max_ma=4,ic=['aic','bic'],↩
    fit_kw={'method':'mle'})
```

```
>>> print(order['aic_min_order'])
(4,2)
>>> print(order['bic_min_order'])
(4,2)

>>> # Fit model
>>> # Note that we need to set the dimensionality to zero in order to have an ↩
    ARMA model.
>>> model = ARIMA(sunspot,order = (4,0,2)).fit(method='innovations_mle')

>>> # Predict values from 1950 to 2012.
>>> prediction = model.predict(start='1950',end='2012')

>>> # Plot the prediction along with the sunspot data.
>>> fig, ax = plt.subplots(figsize=(13,7))
>>> plt.plot(prediction)
>>> plt.plot(sunspot['1950':'2009'])
>>> ax.set_title('Sunspot Dataset')
>>> ax.set_xlabel('Year')
>>> ax.set_ylabel('Number of Sunspots')
>>> plt.show()
```

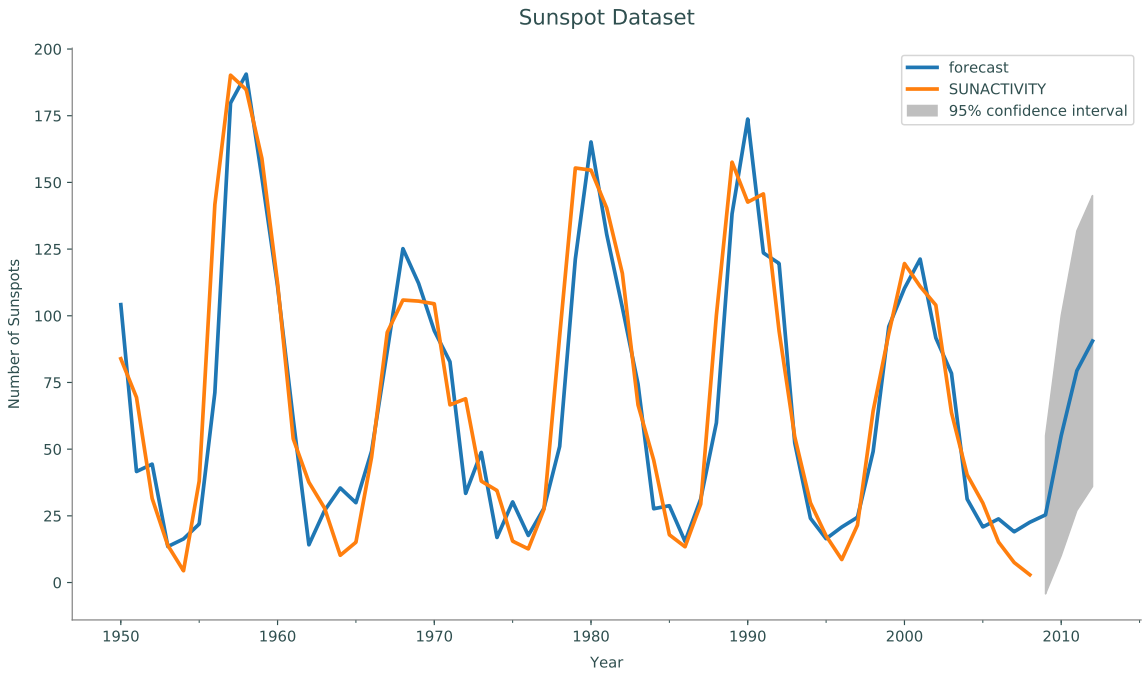


Figure 10.7: Sunspot activity data is forecasted four years in the future using `statsmodels`.

Problem 7. The dataset `manaus` contains data on the height of the Rio Negro from every month between January 1903 and January 1993. Write a function `manaus()` that accepts the forecasting range as strings `start` and `end`, the maximum parameter for the AR model `p` and the maximum parameter of the MA model `q`. The parameters `start` and `end` should be strings corresponding to a DateTimeIndex in the form `Y%M%D`, where `D` is the last day of the month.

The function should determine the optimal order for the ARMA model based on the AIC and the BIC. Then forecast and plot on the range given for both models and compare. Return the order of the AIC model and the order of the BIC model, respectively. For the range `'1983-01-31'` to `'1995-01-31'`, your plot should look like Figure 10.8.

(Hint: The data passed into `arma_order_select_ic` must be a NumPy array. Use the attribute `values` of the Pandas DataFrame.)

To get the `manaus` dataset and set it with a DateTimeIndex, use the following code:

```
>>> # Get dataset
>>> raw = pydata('manaus')
>>> # Convert to DateTimeIndex
>>> manaus = pd.DataFrame(raw.values,index=pd.date_range('1903-01','↩
    1993-01',freq='M'))
>>> manaus = manaus.drop(0,axis=1)
>>> # Set new column title
>>> manaus.columns = ['Water Level']
```

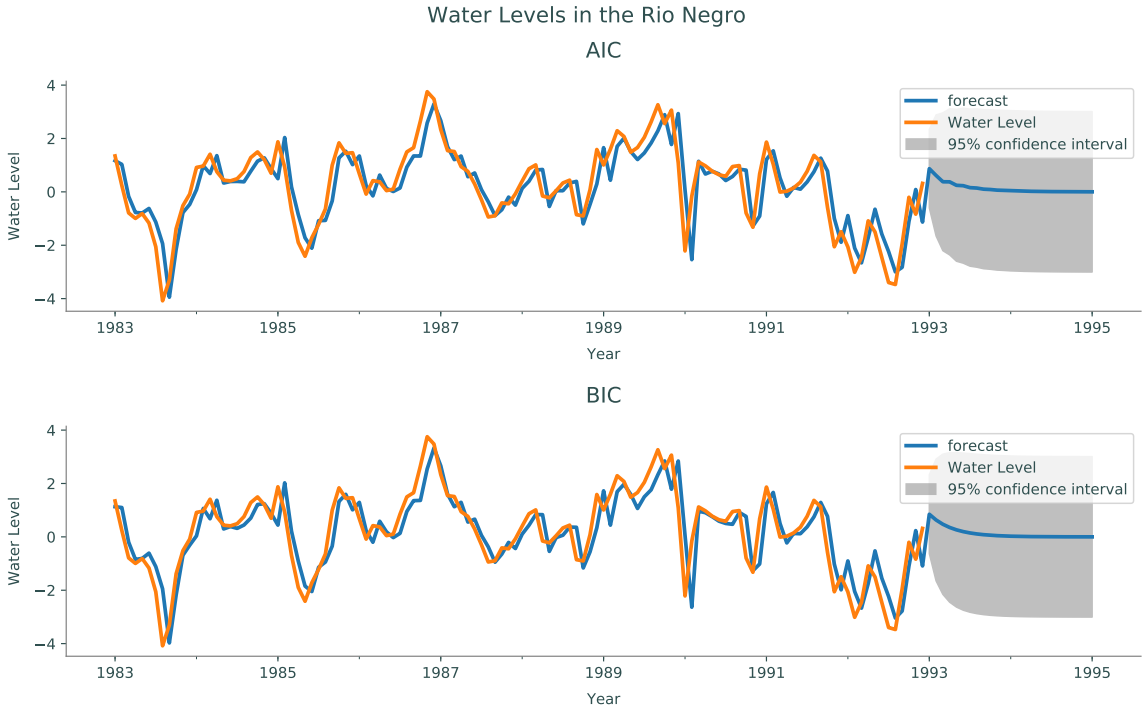


Figure 10.8: AIC and BIC based ARMA models of `manaus` dataset.

Additional Materials

Finding Error Correlation

To find the correlation of the current error with past error, the noise of the data needs to be isolated. Each data point y_t can be decomposed as

$$y_t = T_t + S_t + R_t, \tag{10.23}$$

where T_t is the overall trend of the data, S_t is a seasonal trend, and R_t is noise in the data. The overall trend is what the data tends to do as a whole, while the seasonal trend is what the data does repeatedly. For example, if looking at airfare prices over a decade, the overall trend of the data might be increasing due to inflation. However, we can break this data into individual years. We call each year a season. The seasonal trend of the data might not be strictly increasing, but have increases during busy seasons such as Christmas and summer vacations.

To find T_t, we use an M-fold method. In this case, M is the length of our season. We define the equation

$$T_t = \frac{1}{M} \sum_{-M/2<i<M/2} y_{i+t}. \tag{10.24}$$

This means for each t, we take the average of the season surrounding y_t.

To find the seasonal trend, first subtract the overall trend from the time series. Define $x_t = y_t - T_t$. The value of the seasonal trend can then be found by averaging each day of the season over every season. For example, if the season was one year, we would find the average value on the first day of the year over all seasons, then the second, and so on. Thus,

$$S_t = \frac{1}{K} \sum_{i \equiv t \pmod{M}} x_i \tag{10.25}$$

where K is the number of seasons.

With the overall and seasonal trend known, the noise of the data is simply $R_t = y_t - T_t - S_t$. To determine the strength of correlations with the current error and the past error, plot y_t vs. R_{t-i} as in Figure 10.1.

Proof of Equation 10.15

$$\sum_{i=1}^{p} \phi_i(z_{t-i} - \mu) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} = \sum_{i=1}^{p} \phi_i(H\hat{\mathbf{x}}_{t-i}) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} \tag{10.26}$$

$$= \sum_{i=1}^{r} \phi_i(x_{t-i} + \sum_{k=1}^{r-1} \theta_k x_{t-i-k}) + a_t + \sum_{j=1}^{r-1} \theta_j a_{t-j} \tag{10.27}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j \Big(\sum_{i=1}^{r} \phi_i x_{t-j-i} + a_{t-j} \Big) \tag{10.28}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j x_{t-k} \tag{10.29}$$

$$= x_t + \sum_{j=1}^{r-1} \theta_j x_{t-k} \theta_k x_{t-k} \tag{10.30}$$

$$= z_t. \tag{10.31}$$