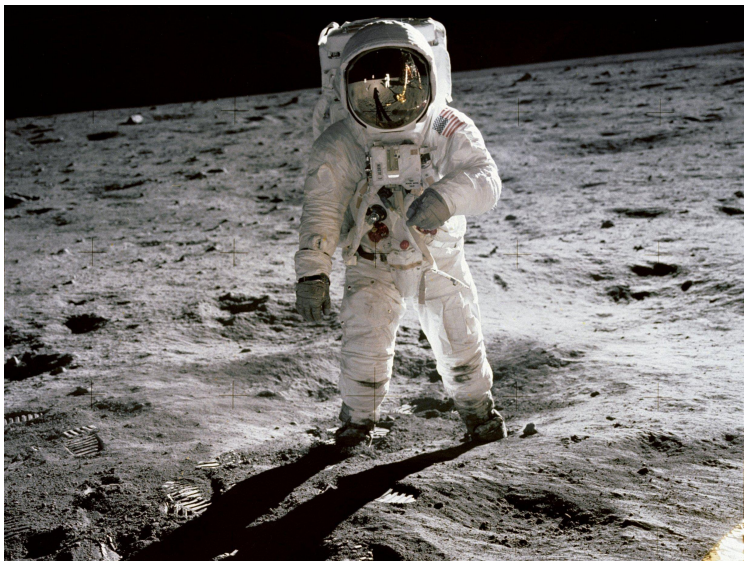
Labs for Foundations of Applied Mathematics

Volume 3
Modeling with Uncertainty and Data

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

A. Oveson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

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

Contents

Part I

Labs

1 Information Theory and Wordle

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

Wordle

Wordle is a word game[1] where you have 6 guesses to guess a five-letter word. Every time a guess is made, you receive some information about how close your guess is to the correct answer. Letters in the guess that are in the correct location are colored green; letters that are present in the secret word but not in the correct location are colored yellow; and letters that aren't present in the secret word are colored gray. An example game is given in Figure 1.1.

Figure 1.1: An example game of Wordle. Here, the secret word is "train." Green tiles mean the letter is in the correct location; yellow tiles mean the letter is in the secret word but not at that location; and grey tiles mean the letter is not in the secret word.

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

[1] It was particularly popular on the internet in 2022.

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

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

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

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

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

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

Computing Guess Results

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

As part of this algorithm, we will need to use the guess results for every pair of possible secret word and allowed guess. Rather than computing these each time, we will compute these once and save the results in an array. For convenience, we have provided this array for you in `all_guess_results.npy`, which can be loaded using the function `np.load()`.

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

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

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

Information and Entropy

Entropy is the expected amount of information we would gain by knowing the result of a random variable. A natural way to define the information of an event A is as $-\log_2 P(A)$.[2] The entropy of a random variable X, which we denote $H(X)$, is then defined as

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

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

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

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

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

These two approaches are in fact equivalent, as

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

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

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

Taking the expectation of both sides implies that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note

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

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

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

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

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

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

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

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

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

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

- Repeat until the secret word has been guessed

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

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

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

2 K-Means Clustering

Lab Objective: *Clustering is the one of the main tools in unsupervised learning—machine learning problems where the data comes without labels. In this lab we implement the k-means algorithm, a simple and popular clustering method, and apply it to geographic clustering and color quantization.*

Clustering

In this lab, we will analyze a few different datasets from Scikit-Learn's library and use the K-means algorithm. Figure 2.1 is a graph of the iris dataset. As a human, it is easy to identify the two distinct groups of data. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*. The K-means algorithm is a simple way of helping computers see the group distinctions.

The objective of clustering is to find a partitions of the data such that points in the same subset will be "close" according to some metric. The metric used will likely depend on the data, but some obvious choices include Euclidean distance and angular distance. Throughout this lab, we will use the metric $d(x, y) = \|x - y\|_2$, the Euclidean distance between x and y, unless we specify a different metric to be used.

More formally, suppose we have a collection of $\mathbb{R}^K$-valued observations $X = \{x_1, x_2, \dots, x_n\}$. Let $N \in \mathbb{N}$ and let $\mathcal{S}$ be the set of all N-partitions of X, where an N-partition is a partition with exactly N nonempty elements. We can represent a typical partition in $\mathcal{S}$ as $S = \{S_1, S_2, \dots, S_N\}$, where

$$X = \bigcup_{i=1}^{N} S_i$$

and

$$|S_i| > 0, \qquad i = 1, 2, \dots, N.$$

We seek the N-partition S^* that minimizes the within-cluster sum of squares, i.e.

$$S^* = \arg\min_{S \in \mathcal{S}} \sum_{i=1}^{N} \sum_{x_j \in S_i} \|x_j - \mu_i\|_2^2,$$

where μ_i is the mean of the elements in S_i, i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

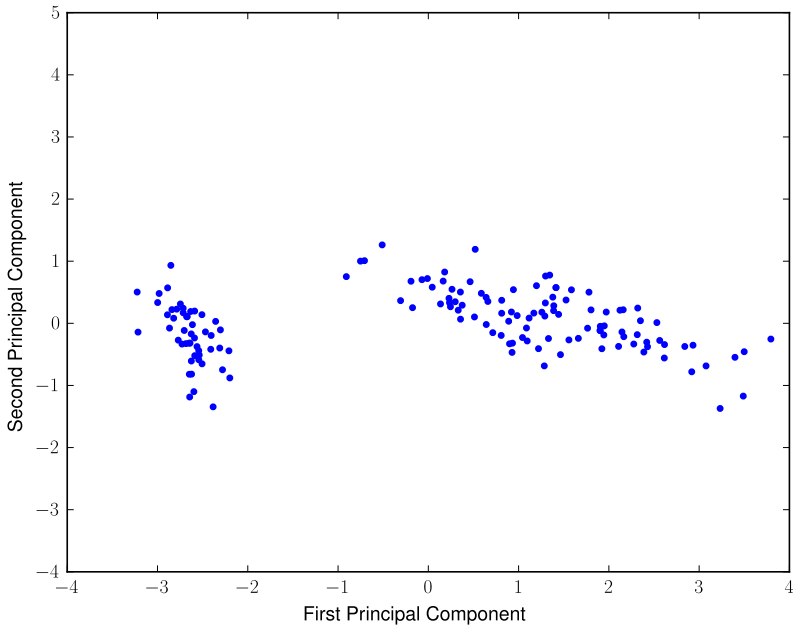


Figure 2.1: The first two principal components of the iris dataset.

The K-Means Algorithm

Finding the global minimizing partition S^* is generally intractable since the set of partitions can be very large indeed, but the *k-means* algorithm is a heuristic approach that can often provide reasonably accurate results.

We begin by specifying an initial cluster mean $\mu_i^{(1)}$ for each $i = 1, \cdots, N$. This can be done by random initialization, or according to some heuristic. For each iteration, we adopt the following procedure. Given a current set of cluster means $\mu^{(t)}$, we find a partition $S^{(t)}$ of the observations such that

$$S_i^{(t)} = \{x_j \ : \ \|x_j - \mu_i^{(t)}\|_2^2 \leq \|x_j - \mu_l^{(t)}\|_2^2, \ \ l = 1, \cdots, N\}.$$

We then update our cluster means by computing for each $i = 1, \cdots, N$. We continue to iterate in this manner until the partition ceases to change.

Figure 2.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

The algorithm can be summarized as follows.

1. From the data points, choose k initial cluster centers.
2. For $i = 0, \ \ldots, \texttt{max\_iter}$,
   (a) Assign each data point to the cluster center that is closest, forming k clusters.
   (b) Recompute the cluster centers as the means of the new clusters.

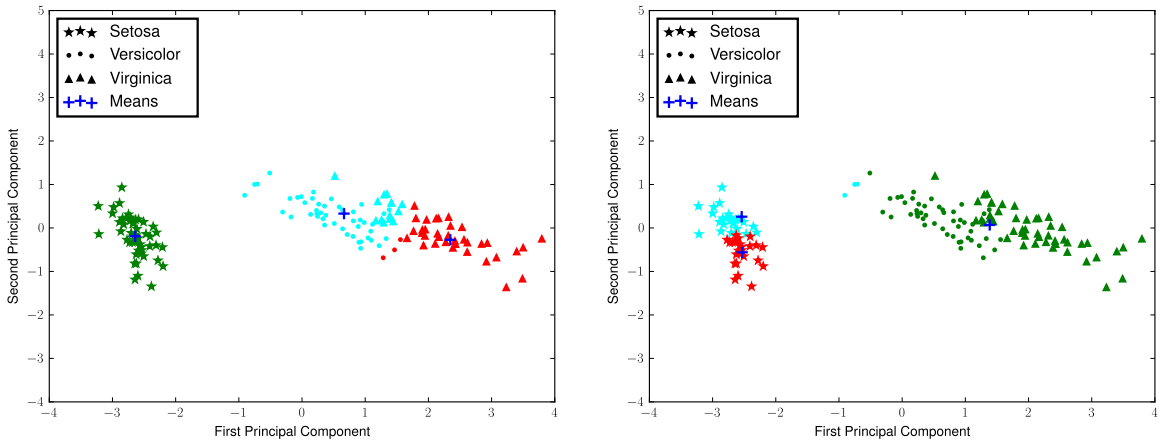


Figure 2.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

(c) If the old cluster centers and the new cluster centers are sufficiently close, terminate early.

Problem 1. Write a `KMeans` class for doing basic k-means clustering. Implement the following methods.

1. `__init__()`: Accept a number of clusters k, a maximum number of iterations, and a convergence tolerance. Store these as attributes.

2. `fit()`: Accept an $m \times n$ matrix X of m data points with n features. Choose k random rows of X as the initial cluster centers. Run the k-means iteration until consecutive centers are within the convergence tolerance, or until iterating the maximum number of times. Save the cluster centers as attributes.

   If a cluster is empty, reassign the cluster center as a random row of X.

3. `predict()`: Accept an $l \times n$ matrix X of data. Return an array of l integers where the ith entry indicates which cluster center the ith row of X is closest to.

4. `plot()`: Accept an $l \times n$ matrix X of l data points and an array y of l integers representing the labels. Plot each data point from the matrix, colored by cluster, along with the cluster centers. Note that in this case, $n = 2$.

Test your class on the iris data set after reducing the data to two principal components. Plot the data, coloring by cluster.

Fire Station Placement

When urban planners are making plans for a city, there are many city elements to consider. One of which is the locations of the fire stations that will service the city. When choosing a suitable location for the city, urban planners look at the current building locations, the roads nearby each location, prior traffic history and the areas of potential growth. We will simplify this complex problem by only taking into account the distances from each building to the nearest fire station (see Additional Material for a harder version of this problem).

Using another data set from SKLearn, we can get the data from the 1990 US Census for California housing based on the blocks of the residents. This has been saved in `sacramento.npy` and can be accessed by using the np.load() function. This file contains demographic data for each block in Sacramento and nearby cities. The eight columns in the file are: median block income, median house age in the block, average number of rooms, average number of bedrooms, average house occupancy, latitude and longitude.

There are couple ways for a fire station to be optimally placed. The stations could be placed to minimize the average distance to each house. Another option is to minimize the distance to the farthest house in each group. For this problem, minimize the distance to the farthest house in each group.

Problem 2. Using the Methods you wrote in Problem 1, add a parameter, p, to your class that denotes the norm and defaults to 2. Save p as an attribute to be used in your fit() and predict() functions. Using the latitude and longitude data in `sacramento.npy` find the optimal placement for 12 fire stations. Plot the longitude and latitudes, the centers, and color them by cluster. Make plots for different values for `p` to find the optimal locations for the fire stations. As the initial centers are chosen at random, make sure to run the `predict()` function several times. In a Markdown cell report which norm was the best at keeping the maximum distance small.

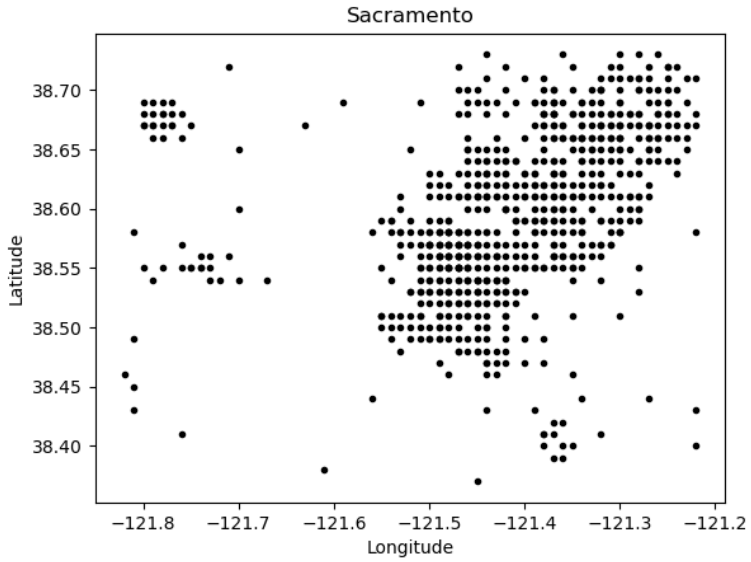


Figure 2.3: Sacramento Housing Data (1990 US Census).

Detecting Active Earthquake Regions

Suppose we are interested in learning about which regions are prone to experience frequent earthquake activity. We could make a map of all earthquakes over a given period of time and examine it ourselves, but this, as an unsupervised learning problem, can be solved using our k-means clustering tool.

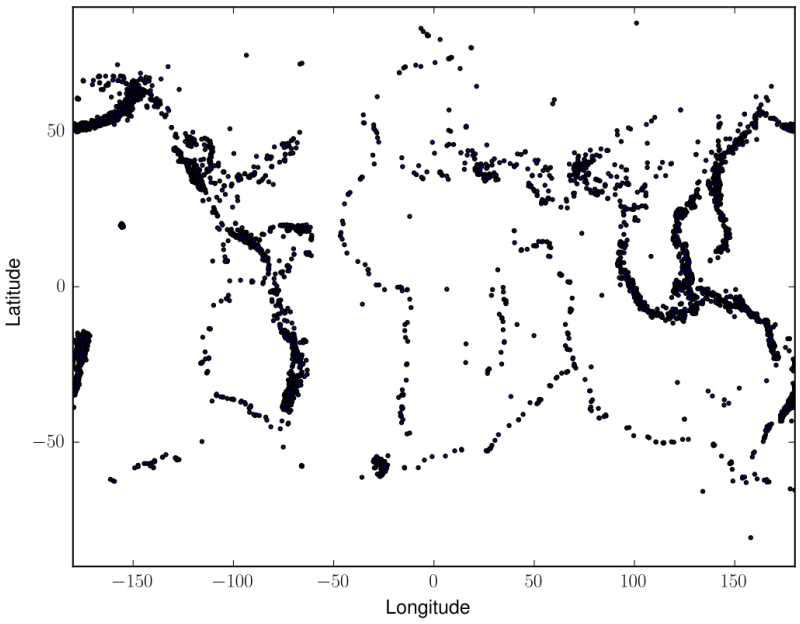


Figure 2.4: Earthquake epicenters over a 6 month period.

The file `earthquake_coordinates.npy` contains earthquake data throughout the world from January 2010 through June 2010. Each row represents a different earthquake; the columns are scaled longitude and latitude measurements. We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in $\mathbb{R}^2$ with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. Instead, latitude and longitude should be viewed in *spherical coordinates* in $\mathbb{R}^3$, which could then be clustered.

A simple way to accomplish this transformation is to first transform the latitude and longitude values to spherical coordinates, and then to Euclidean coordinates. Recall that a spherical coordinate in $\mathbb{R}^3$ is a triple (r, θ, φ), where r is the distance from the origin, θ is the radial angle in the xy-plane from the x-axis, and φ is the angle from the z-axis. In our earthquake data, once the longitude is converted to radians it is an appropriate θ value; the latitude needs to be offset by $90°$ degrees, then converted to radians to obtain φ. For simplicity, we can take $r = 1$, since the earth is roughly a sphere. We can then transform to Euclidean coordinates using the following relationships.

$$\theta = \frac{\pi}{180}\,(\text{longitude}) \qquad \varphi = \frac{\pi}{180}\,(90 - \text{latitude})$$

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} & x &= r\sin\varphi\cos\theta \\ \varphi &= \arccos\frac{z}{r} & y &= r\sin\varphi\sin\theta \\ \theta &= \arctan\frac{y}{x} & z &= r\cos\varphi \end{aligned}$$

There is one last issue to solve before clustering. Each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. Therefore, the cluster centers should also have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth, and the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors.

Problem 3. Add a keyword argument `normalize=False` to your `KMeans` constructor. Modify `fit()` so that if `normalize` is `True`, the cluster centers are normalized at each iteration.

Cluster the earthquake data in three dimensions by converting the data from raw data to spherical coordinates to euclidean coordinates on the sphere.

1. Convert longitude and latitude to radians, then to spherical coordinates.
   (Hint: `np.deg2rad()` may be helpful.)
2. Convert the spherical coordinates to euclidean coordinates in $\mathbb{R}^3$.
3. Use your `KMeans` class with normalization to cluster the euclidean coordinates.
4. Translate the cluster center coordinates back to spherical coordinates, then to degrees. Transform the cluster means back to latitude and longitude coordinates.
   (Hint: use `numpy.arctan2()` for arctan, so that that correct quadrant is chosen).
5. Plot the data, coloring by cluster. Also mark the cluster centers.

With 15 clusters, your plot should resemble the Figure 2.5.

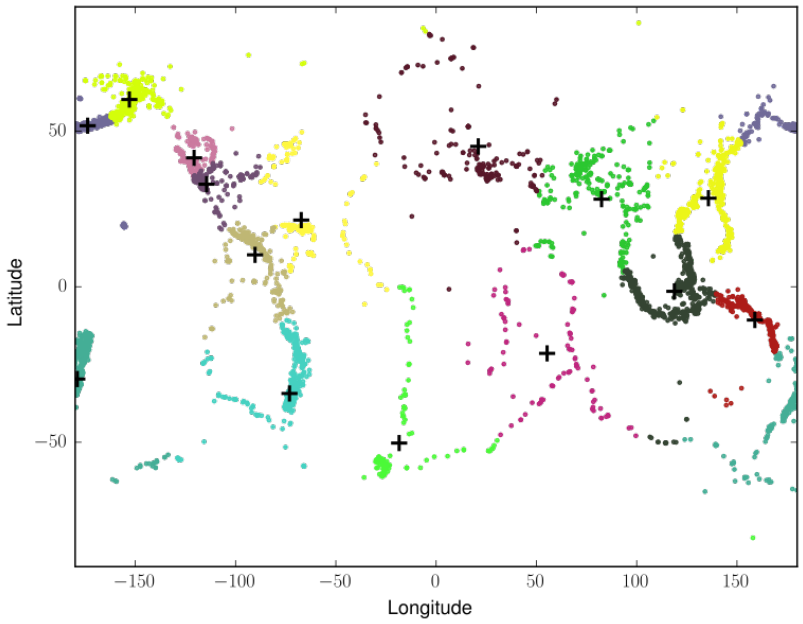


Figure 2.5: Earthquake epicenter clusters with $k = 15$.

Color Quantization

The k-means algorithm uses the euclidean metric, so it is natural to cluster geographic data. However, clustering can be done in any abstract vector space. The following application is one example.

Images are usually represented on computers as 3-dimensional arrays. Each 2-dimensional layer represents the red, green, and blue color values, so each pixel on the image is really a vector in $\mathbb{R}^3$. Clustering the pixels in RGB space leads a one kind of image segmentation that facilitate memory reduction.

Reading: `https://en.wikipedia.org/wiki/Color_quantization`

Problem 4. Write a function that accepts an image array (of shape $(m, n, 3)$), an integer number of clusters k, and an integer number of samples S. Reshape the image so that each row represents a single pixel. Choose S pixels to train a k-means model on with k clusters. Make a copy of the original picture where each pixel has the same color as its cluster center. Return the new image. For this problem, you may use `sklearn.cluster.KMeans` instead of your `KMeans` class from Problem 1.

Test your function on some of the provided NASA images.

Additional Material

Spectral Clustering

We now turn to another method for solving a clustering problem, namely that of Spectral Clustering. It can cluster data not just by its location on a graph, but can even separate shapes that overlap others into distinct clusters. It does so by utilizing the spectral properties of a Laplacian matrix. Different types of Laplacian matrices can be used. In order to construct a Laplacian matrix, we first need to create a graph of vertices and edges from our data points. This graph can be represented as a symmetric matrix W where w_{ij} represents the edge from x_i to x_j. In the simplest approach, we can set $w_{ij} = 1$ if there exists an edge and $w_{ij} = 0$ otherwise. However, we are interested in the similarity of points, so we will weight the edges by using a *similarity measure*. Points that are similar to one another are assigned a high similarity measure value, and dissimilar points a low value. One possible measure is the *Gaussian similarity function*, which defines the similarity between distinct points x_i and x_j as

$$s(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

for some set value σ.

Note that some similarity functions can yield extremely small values for dissimilar points. We have several options for dealing with this possibility. One is simply to set all values which are less than some ε to be zero, entirely erasing the edge between these two points. Another option is to keep only the T largest-valued edges for each vertex. Whichever method we choose to use, we will end up with a weighted *similarity matrix* W. Using this we can find the diagonal *degree matrix* D, which gives the number of edges found at each vertex. If we have the original fully-connected graph, then $D_{ii} = n - 1$ for each i. If we keep the T highest-valued edges, $D_{ii} = T$ for each i.

As mentioned before, we may use different types of Laplacian matrices. Three such possibilities are:

1. The *unnormalized Laplacian*, $L = D - W$
2. The *symmetric normalized Laplacian*, $L_{sym} = I - D^{-1/2}WD^{-1/2}$
3. The *random walk normalized Laplacian*, $L_{rw} = I - D^{-1}W$.

Given a similarity measure, which type of Laplacian to use, and the desired number of clusters k, we can now proceed with the Spectral Clustering algorithm as follows:

- Compute W, D, and the appropriate Laplacian matrix.
- Compute the first k eigenvectors $u_1, \cdots, u_k$ of the Laplacian matrix.
- Set $U = [u_1, \cdots, u_k]$, and if using L_{sym} or L_{rw} normalize U so that each row is a unit vector in the Euclidean norm.
- Perform k-means clustering on the n rows of U.
- The n labels returned from your `kmeans` function correspond to the label assignments for $x_1, \cdots, x_n$.

As before, we need to run through our k-means function multiple times to find the best measure when we use random initialization. Also, if you normalize the rows of U, then you will need to set the argument `normalize = True`.

You can use the following function declaration to implement the Spectral Clustering Algorithm by calling your `kmeans` function.

```
def specClus(measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    ----------
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -------
    labels : ndarray of shape (n,)
        The i-th entry is an integer in [0,n_clusters-1] indicating
        which cluster the i-th row of data belongs to.
    """
    pass
```

We now need a way to test our code. The website http://cs.joensuu.fi/sipu/datasets/ contains many free data sets that will be of use to us. Scroll down to the “Shape sets" heading, and download some of the datasets found there to use for trial datasets.

You can use the following function declaration to create a function that will return the accuracy of your spectral clustering implementation.

```
def test_specClus(location,measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    ----------
    location : string
        The location of the dataset to be tested.
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
```

```
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -------
    accuracy : float
        The percent of labels correctly predicted by your spectral
        clustering function with the given arguments (the number
        correctly predicted divided by the total number of points.
    """
    pass
```

Fire Station Placement II

In problem 2 we looked at choosing the best location for a fire station. However, because we looked at the city of Sacramento where the geography doesn't role in choosing a location, we didn't need to double check that there is a place for the station. The `sanfrancisco.npy` data is organized the same way as `sacramento.py`, as this also comes from the SKLearn California Housing Module. Doing the same method as before will give us groups of houses, however, the group centers may be in the middle of the bay. When implementing this problem, perform a check on the centers to make sure they are not in water. The file `bayboundary.npy` gives a rough outline of where the bay is. The `bayboundary.npy` has only 2 columns, longitude and latitude. Using the boundaries set, make sure that the chosen centers are on land and not on water.

As an additional exercise, import and parse the data from the `bayboundary.npy` and the `sanfrancisco.npy` files. Using either the algorithm that you wrote in problem 1 or the k-means algorithm in the SK Learn library, find the optimal locations for the 12 fire stations.

After the algorithm has finished running, check to see if the new coordinates are on land. Return the graph of the clusters, the centers (the fire station locations) as different colors.

3 PCA, LSI, and Scikit-Learn

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

Principal Component Analysis

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

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

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

$$\frac{\sigma_i^2}{\sum_{j=1}^{k} \sigma_j^2} \tag{3.1}$$

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

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

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

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

Write a function that performs PCA on the breast cancer dataset using the SVD as described above. Graph the first two principal components, with the first along the x-axis. Your graph should resemble Figure 3.1 below. Include the proportion of the total variance that the first two principal components capture in the graph title, calculated with Equation 3.1.

You can load this dataset using the following code:

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

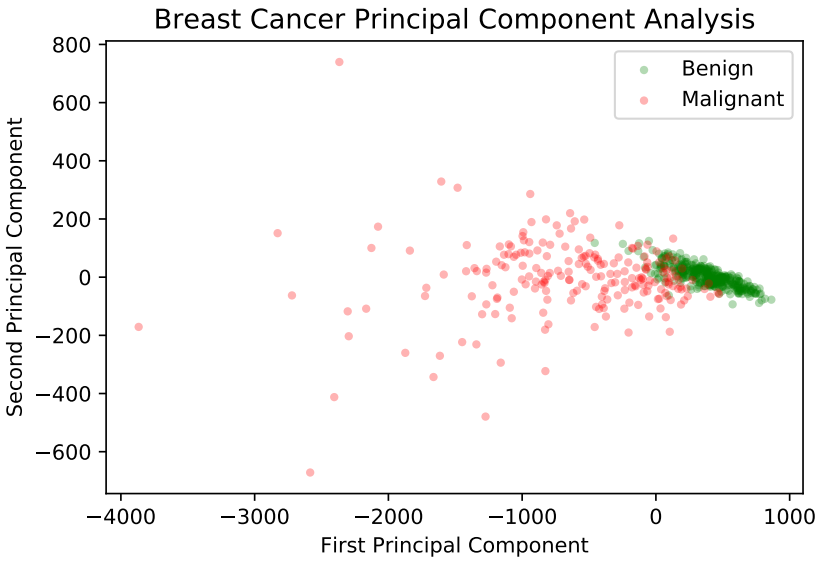


Figure 3.1: First two principal components of the transformed breast cancer data

Latent Semantic Indexing

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

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

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

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

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

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

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

$$\frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

To find the document most similar to document i, we simply compute

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

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

Hint: `np.argsort` may be useful for finding which are the most and least similar. Note that every document will have a similarity score of 1 with itself, so be careful not to return a document as its own closest document.

To test your code, use the following matrix:

```
X = np.array([
    [0.78, 0.14, 0.12, 0.  ],
    [0.64, 0.97, 0.  , 0.  ],
    [0.  , 0.  , 0.63, 0.46],
    [0.  , 0.84, 0.6 , 0.  ],
    [0.29, 0.89, 0.51, 0.  ],
    [0.77, 0.  , 0.27, 0.2 ],
    [0.86, 0.47, 0.  , 0.06],
    [0.89, 0.  , 0.  , 0.  ]
```

```
])
```

With `i=4` your function should output the following:

```
>>> print(similar(4, X))
(7, 3)
```

Application: State of the Union

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

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

```
# Get list of filepaths to each text file in the folder.
folder = "./Addresses/"
paths = sorted([folder+p for p in os.listdir(folder) if p.endswith(".txt")])

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

# Initialize vocab set, then read each file and add to the vocab set.
vocab = set()
for p in paths:
    with open(p, 'r', encoding="utf8") as infile:
        for line in infile:
            vocab.update(extractWords(line)) # Union sets together
```

We now have a set containing all of the unique words in the corpus. However, many of the most common words do not provide important information. We call these *stop words*. Examples in English include *the, a, an, and, I, we, you, it, there*, etc; a list of common English stop words is given in `stopwords.txt`. We remove the stop words from our vocabulary set as follows and then fix an ordering to the vocabulary by creating a dictionary with key-value pairs of the form (word, index).

```
# Load stopwords
with open("stopwords.txt", 'r',  encoding="utf8") as f:
    stops = set([w.strip().lower() for w in f.readlines()])
```

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

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

```
from collections import Counter

counts = []      # holds the entries of X
doc_index = []   # holds the row index of X
word_index = []  # holds the column index of X

# Iterate through the documents
for doc, p in enumerate(paths):
    with open(p, 'r', encoding="utf8") as f:
        # Create the word counter.
        ctr = Counter()
        for line in f:
            ctr.update(extractWords(line))
        # Iterate through the word counter, storing counts.
        for word, count in ctr.items():
            if word in vocab:
                word_index.append(vocab[word])
                counts.append(count)
                doc_index.append(doc)

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

Problem 3. Applying the techniques of LSI discussed above to the word count matrix X, created with the `document_converter()` function, and keeping the first 7 principal components, write a function that takes in the path to a single State of the Union address `speech` and returns a tuple of the addresses that are most and least similar to `speech`.

For Ronald Reagan's 1984 speech, the input would be `'./Addresses/1984-Reagan.txt'`, and your output should be `('1985-Reagan', '1946-Truman')`. Be sure to format the strings properly. Also run your algorithm on Clinton's 1993 speech, and display your output.

Since X is a sparse matrix, you will need to use the SVD method found in `scipy.sparse.linalg`. This method operates slightly differently than the SVD method found in `scipy.linalg`, so be sure to read the documentation. Also pass the argument `random_state=28` into this function for consistency.

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

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

We then let

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

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

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

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

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

Hint: the function `np.log1p`, which calculates $\log(1+x)$, can be applied to a sparse matrix without losing sparsity.

Scikit-Learn

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

Transformers

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

NOTE

A *hyperparameter* is not dependent on data. Hyperparameters are declared in the constructor `__init__()`, before data is even passed in. Parameters set during the `fit()` method are often called *model parameters* and do depend on specific data. For example, a `StandardScaler` transformer shifts and scales data to have a mean of 0 and a standard deviation of 1.

Scikit-learn's transformers have three main methods: `fit_transform()`, which fits model parameters and also transforms given data; `fit()`, which sets model parameters but does not perform a transformation; and `transform()`, which transforms data according to pre-fitted model parameters. Model parameters are fitted according to training data, and they are not refitted to testing data, so a `StandardScaler` will shift and scale testing data according to the mean and variance of the training data; the transformed test data likely will not have mean 0 and variance 1.

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

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

For our particular application, however, ince our matrix is sparse and Scikit-learn's `PCA` class does not accept sparse matrices, we need to use their `TruncatedSVD` class instead. The syntax for this class is identical to the above.

Note

An interesting observation can be made by repeating Problem 1 using scikit-learn's PCA package. The resulting graph will have the x-axis flipped, because a matrix's singular value decomposition is unique up to multiplication by -1.

Problem 5. Repeat Problem 3 using your weighted document converter function and scikit-learn's `sklearn.decomposition.TruncatedSVD` class. For consistency, also pass the argument `random_state=74` in the constructor to this class.

For Bill Clinton's 1993 speech, your code should return `('1994-Clinton', '1946-Truman')`. Also run the algorithm on Reagan's 1984 speech, and display the results.

Validation Tools

We now turn our attention from transformers to classifiers. A *classifier* is trained to predict how a new sample should be classified or labeled. Knowing how to determine whether or not a classifier performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the possible labels are only `0` or `1`.

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

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

With this terminology, we define the following metrics.

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

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

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

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

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

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

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

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

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

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

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

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

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

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

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

Grid Search

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

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

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

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

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

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

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

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

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

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

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

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

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* chains together one or more transformers and one estimator (such as a classifier) into a single object, complete with `fit()` and `predict()` methods. This simplifies and automates the machine learning process so that when you get new data or make changes to various functions and features, you can easily rerun the new version from beginning to end.

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

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

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

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

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

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

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

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

```
0.971830985915493
```

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

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

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

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

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

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

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

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

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

4 Random Forests

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

Classification Trees

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

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

A classification tree is composed of many *nodes*, which ask a question (i.e. "Is income $\geq$ 85?") and then split the data based on the answers. If the response is `True`, then the sample is "pushed" down the tree to the left child node. If the response is `False`, then the sample is "pushed" down the tree to the right child node. A *leaf* node is a node that has no child node. Upon arrival at a leaf, an unlabeled sample is labeled with the classification that matches the majority of labeled samples at that leaf.

Table 4.1 includes information about 10 individuals and an indicator of whether or not they made a certain purchase. To simplify construction of the tree, all data is numeric, so 1=Yes and 0=No for yes/no questions.

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

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

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

Table 4.1: Customer data with 3 features (Married, Children, Income) and a label (Purchase) indicating whether or not the customer bought the item.

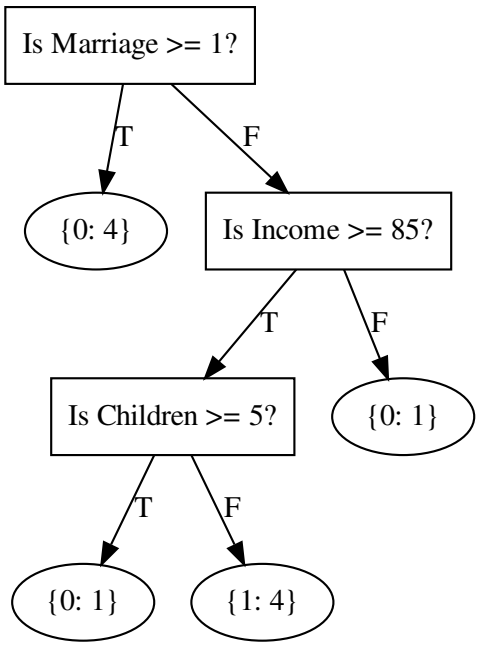


Figure 4.1: A classification tree built using Table 4.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

Problem 1. At each node in a classification tree, a question determines which branch a sample belongs to. The `Question` class has attributes `column` and `value`. Write a `match` method for the `Question` class that accepts a sample and returns either `True` or `False`. A sample will be in the form of an array, so in the example above, a single college student with no children making \$20,000 would be represented by the array $[0, 0, 20]$. The method should determine if the sample's feature located at index `column` is greater than or equal to `value`. Notice that this method will only handle one feature of one sample at a time.

Next, write a `partition()` function that partitions the samples (rows) of a dataset for a given `Question` into two `numpy` arrays, `left` and `right`, returned in that order. The array `left` will contain the samples that the `match` method returned as `True`, and the array `right` will contain the samples that the `match` method returned as `False`. If `left` or `right` is empty, still return them as (2-D size zero) arrays.

Hint: if `n` is the length of each sample, `left.reshape(-1,n)` (and similar for `right`) will make the final arrays the correct size even if they're empty.

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

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

# initialize question and test partition function
>>> question = Question(column=1, value=3, feature_names=features)
>>> left, right = partition(animals, question)
>>> print(len(left), len(right))
62 38

>>> question = Question(column=1, value=75, feature_names=features)
>>> left, right = partition(animals, question)
>>> print(len(left), len(right))
0 100
```

Optimal Split

To use the `partition()` function from Problem 1, we need to know which question to ask at each node. Usually, the question is determined by the split that maximizes either the Gini impurity or the information gain (which itself uses the Gini impurity). For this lab, we will use the information gain.

Gini impurity measures how often a sample would be mislabeled based on the distribution of labels. It is a measure of homogeneity of labels, so it is 0 when all samples at a node have the same label.

Definition 4.1. *Let D be a dataset with K different class labels and N different samples. Let N_k be the number of samples labeled class k for each $1 \leq k \leq K$. We define the* Gini impurity *to be*

$$G(D) = 1 - \sum_{k=1}^{K} \left(\frac{N_k}{N} \right)^2.$$

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

Definition 4.2. *Let $s_D(p, x) = D_1, D_2$ be a partition of data D. We define the* information gain *of this partition to be*

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

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

The provided function `info_gain()` can be used to compute the information gain of a partiton of a dataset. The optimal split of data at a node can be chosen by finding the question whose partition maximizes the information gain.

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

To find the optimal split, begin by instantiating `best_gain` to 0 and `best_question` to `None`. Iterate through each feature (column) of the dataset, and then through each unique value (row) that the feature takes on. You must iterate through the columns and then the rows; doing the opposite will yeild the wrong answer! Instantiate a `Question` object with the column and value, and then use `partition()` to split the dataset into left and right partitions. If either the left or right partition has less samples than the smallest allowable leaf size (called `min_samples_leaf`), then discard this partition and iterate to the next one. If the left and right partitions are ok, then calculate the `info_gain()` of these two partitions with the Gini impurity of the dataset. If this `info_gain()` is greater than `best_gain`, then set this `info_gain()` and its corresponding `Question` equal to `best_gain` and `best_question`, respectively. Once you iterate through every column and row, return `best_gain` and `best_question`.

Problem 2. Write a function `find_best_split()` that computes the optimal split of a dataset by following the directions given above. Recall that the final column of the dataset contains the class labels, which has no questions associated with it, so do not iterate through the final column. Include a minimum leaf size argument `min_samples_leaf` defaulting to 5.

Return the information gain and corresponding question for the best split, in that order. If two splits have the same information gain, choose the first split. If no partitions are possible due to `2*min_samples_leaf`, return `None` for the question.

You should get the following output for the animals dataset.

```
# Test your function
>>> find_best_split(animals, features)
(0.12259833679833687, Is # legs/tentacles >= 2.0?)
```

Building the Tree

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

Problem 3. The class `Leaf` is instantiated with data containing samples in that leaf. In the constructor, save an attribute `prediction` as a dictionary of how many samples in the leaf belong to each unique class label.

Hint: remember the provided function `class_counts()`.

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

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

Problem 4. Write a function `build_tree()` that uses your previous functions to build a classification tree. Include a minimum leaf argument defaulting to 5 and a maximum depth argument defaulting to 4. Start counting depth at 0. For comparison, the tree in Figure 4.1 has depth 3. We will build this tree recursively as follows:

- If the number of samples (rows) in the given data is less than twice `min_samples_leaf` (i.e., it can't be split again), then return the data as a `Leaf` and that's it.
- Otherwise, the data can be split, so find the optimal gain and corresponding question using the function `find_best_split()`.

- If the optimal gain is 0 (i.e., if the partition is already optimal), or if `current_depth` is greater than or equal to `max_depth` (i.e., the tree is already too deep), return the data as a `Leaf` and that's it.
- If the node isn't a `Leaf`, then it must be a `Decision_Node`.
  - Use `partition()` to split the data into left and right partitions.
  - Next, recursively define the right branch and left branch of the tree by calling `build_tree()` on each of the left and right partitions with `current_depth` incremented by 1.
  - Finally, return a `Decision_Node` object using the optimal question found earlier and the left and right branches of the tree.

The function `draw_tree()` is provided to allow you to save a pdf image to view a specified trained tree. In order to use this function, you must successfully install the `graphiz` package.

`graphviz` has two parts: an external program, and a Python package that interfaces with that program. To install both of these parts, refer to the section in Additional Materials. (Both of these are installed by `install_dependencies.sh`.)

With `graphviz` installed, you can test your `build_tree` function as follows:

```
>>> my_tree = build_tree(animals, features)
>>> draw_tree(my_tree)
```

The resulting tree should have 8 question nodes, 9 leaf nodes, and a total of 5 rows of nodes (including the lowermost leaves). If `draw_tree` returns an error about pdf being an unrecognized file type, try running the command `dot -c` in your terminal.

Predicting

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

Problem 5. Write a function `predict_tree()` that returns the predicted class label for a single new sample given a trained tree called `my_tree`. This function will be implemented recursively in order to traverse the branches and reach a `Leaf` node. Use the `isinstance()` function to determine if the current node (`my_tree`) is of type `Leaf`; if it is, return the label that corresponds with the most samples in the `Leaf`.

If the given tree is not a `Leaf`, then it is a `Decision_Node` with left and right children nodes. If the `my_tree.question.match` method is `True` with the given sample, then recursively call `predict_tree()` with `my_tree.left`. Otherwise, recursively call `predict_tree()` with `my_tree.right`.

Hint: an easy way to get the most common value at a `Leaf` node is to call `max()` on `my_tree.prediction` using the keyword argument `key=my_tree.prediction.get`. Remember to return something from both branches of your function.

Next, write a function `analyze_tree()` that accepts a labeled dataset (with the labels in the last column, as in `animals.csv`) and a trained classification tree. The function should compare the actual label of each sample (row) with its predicted label using `predict_tree()`, and it should return the proportion of samples that the tree labels correctly.

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

Random Forest

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

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

Problem 6. Add an argument `random_subset` to `find_best_split()` and `build_tree()`, defaulting to `False`, that indicates whether or not the tree should be trained randomly. When `True`, each node should be restricted to a random combination of $\sqrt{n}$ (rounded down) features to use in its split, where n is the total number of features (note that class labels are not features). This will require the function `find_best_split()` to be altered so that it only iterates through a random combination of $\sqrt{n}$ features (columns).

Next, write a function `predict_forest()` that accepts a new sample and a trained forest (as a list of trees). It should iterate through each tree, finding the label assigned to the sample by calling `predict_tree()`. Then, it should return the label predicted by the majority of the trees.

Finally, write a function `analyze_forest()` that accepts a labeled dataset (with the labels in the last column, as in `animals.csv`) and a trained forest. The function should compare the actual label of each sample (row) with its predicted label using `predict_forest()`, and it should return the accuracy of the forest's predictions.

Test your functions on the `animals.csv` dataset. Visualize your trees using `draw_tree()`, verifying that they are different every time. Compare the results to the non-randomized version.

Scikit-Learn

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

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

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

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

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

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

Problem 7. The file `parkinsons.csv` contains annotated speech data from people with and without Parkinson's Disease. The first column is the subject ID, columns 2-27 are various features, and the last column is the label indicating whether or not the subject has Parkinson's. You will need to remove the first column so your forest doesn't use participant ID to predict class labels. Feature names are contained in the file `parkinsons_features.csv`.

Write a function to compare your forest implementation to the package from scikit-learn. Because of the size of this dataset, we will only use a small portion of the samples and build a very simple forest. Randomly select 130 samples. Use 100 in training your forest and 30 more in testing it. Build 5 trees for your forest (as a list) using `min_samples_leaf=15` and `max_depth=4` for each tree. Time how long it takes to train and analyze your forest.

Repeat this with scikit-learn's package, using the same 100 training samples and 30 test samples. Set `n_estimators=5`, `min_samples_leaf=15`, and `max_depth=4`.

Then, using scikit-learn's package, run the whole `parkinsons.csv` dataset, using the default parameters. Use 80% of the data to train the forest and the other 20% to test it.

Return three tuples: the accuracy and time of your implementation, the accuracy and time of scikit-learn's package, and then the accuracy and time of scikit-learn's package using the entire dataset.

Additional Materials

Installing Graphviz

The Python package can be installed using `pip install graphviz`. To install the external program, on Windows in the WSL terminal and on Linux use the following:

```
sudo apt-get install graphviz
```

On Mac:

```
brew install graphviz
```

For additional options for the external program, refer to the instructions at `https://graphviz.org/download/`.

5 Linear Regression

Lab Objective: *This section will introduce the basics of Linear Regression, feature selection methods, and regularization.*

Introduction to Linear Regression

One of the first skills taught in basic algebra is to effectively plot the line $y = mx + b$ which can be done with two points. But what if we want to find the line that best fits a set of points?

In this case, we can use the simplest form of linear regression: *Ordinary Least Squares* (*OLS*). Given data as a set of points $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$ we wish to find the line that best fits the data. The line is given by $y = mx + b$ where m and b are unknown constants and x and y are the independent and dependent variables respectively. Using OLS, let

$$y_i = mx_i + b + \varepsilon_i$$

describe the *ith* point in D for each $i \in \{1, \dots, n\}$. Note that ε_i is the vertical distance from the *ith* point to the line given by $y = mx + b$ and is often called the *residual* or the *error*.

The n equations for each point in D can be written in vector notation. Let the x and y coordinates of D be represented by column vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. In statistical science, the intercept (b) and slope (m) are denoted as β_0 and β_1 respectively and

$$\begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \boldsymbol{\beta}.$$

Additionally, the residuals are represented by a column vector $\boldsymbol{\varepsilon}$ and $\mathbf{1}$ is a column vector of ones. So we have

$$\boldsymbol{y} = m\boldsymbol{x} + \mathbf{1}b + \boldsymbol{\varepsilon} = [\mathbf{1}, \boldsymbol{x}] \cdot \begin{bmatrix} b \\ m \end{bmatrix} + \boldsymbol{\varepsilon}.$$

Denoting $X = [\mathbf{1}, \boldsymbol{x}]$, we have our final equation given as

$$\boldsymbol{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon}.$$

This notation may seem excessive, but suppose we wanted to fit a model of the form $y = ax^3 + bx^2 + cx + d$ A little work can show that $X = [\mathbf{1}, \boldsymbol{x}, \boldsymbol{x}^2, \boldsymbol{x}^3]$ and $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, \beta_3]^T$, which is very easy to work with. Thus, this notation is actually the ideal way to generalize linear regression, especially when working with higher degree polynomials.

The solution to OLS is straight forward with some important assumptions. Sparing you the algebraic details and assuming that $\boldsymbol{y} \sim \mathcal{N}(X\boldsymbol{\beta}, \sigma^2\boldsymbol{I})$ and $\boldsymbol{\varepsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2\boldsymbol{I})$ and $\boldsymbol{I}$ is the identity matrix, the least squares estimator for $\boldsymbol{\beta}$ is given as

$$\widehat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \boldsymbol{y}. \tag{5.1}$$

Problem 1. Write a function that takes as input `X` and `y`. In your function, add a column of ones to `X` to account for β_0. Call this function `ols`. This function should return the least squares estimator for $\boldsymbol{\beta}$ as a numpy array.

Hint: Try rearranging equation 1.1 and use `np.linalg.solve()` to avoid inverting $(X^T X)$.

Problem 2. Use the following code to generate random data.

```
n = 100 # Number of points to generate
X = np.arange(100) # The input X for the function ols
eps = np.random.uniform(-10,10, size=(100,)) # Noise to generate random y ↩
    coordinates
y = .3*X + 3 + eps # The input y for the function ols
```

Find the least squares estimator for $\boldsymbol{\beta}$ using this random data. Produce a plot showing the random data and the line of best fit determined by the least squares estimator for $\boldsymbol{\beta}$. Your plot should include a title, axis labels, and a legend.

Rank-Deficient Models

Notice that in order to find the least squares estimator $\widehat{\boldsymbol{\beta}}$, we need $X^T X$ to be invertible. However, when X does not have full rank, the product $X^T X$ is singular and not invertible. We can no longer use the previous solution for the least squares estimator, but we can use the SVD and still compute a solution.

Recall that if $X \in M_{n\times d}$ has rank r, then the compact form of the SVD of X is

$$X = U\Sigma V^H$$

where $U \in M_{n\times r}$ and $V \in M_{r\times d}$ have orthonormal columns and $\Sigma \in M_{r\times r}$ is diagonal. In addition, if X is real, then the factors U, Σ, and V^H are also real. In this lab we assume X is real. As described in Volume 1, there is a unique solution for the least squares estimator given by

$$\widehat{\boldsymbol{\beta}} = V\Sigma^{-1} U^T \boldsymbol{y}. \tag{5.2}$$

Problem 3. Write a function that finds the least squares estimator for rank-deficient models using the SVD. The function should still take `X` and `y` as inputs. In your function, add a column of ones to `X` to account for β_0. Call the function `svd_ols` and return the least squares estimator for $\boldsymbol{\beta}$ as a numpy array.

Hint: Use `np.linalg.svd` to factor `X` and use the argument `full_matrices=False`. Consider solving for Σ^{-1} using the command, `np.diag(1/s)`, where `s` is the second thing being returned in `np.linalg.svd`.

Problem 4. Use the following code to generate random data:

```
x = np.linspace(-4, 2, 500)
y = x**3 + 3*x**2 - x - 3.5
eps = np.random.normal(0, 3, len(y)) # Create noise
y += eps # Add noise to randomize data
```

Now use your function `svd_ols` to find the least squares estimator for a cubic polynomial. This can be done by passing in $X = [\boldsymbol{x}, \boldsymbol{x}^2, \boldsymbol{x}^3]$ into `svd_ols`. Create a plot that shows a scatter plot of the data and a curve using the least squares estimator. Your plot should include a title, axis labels, a legend, and should look similar to Figure 5.1.

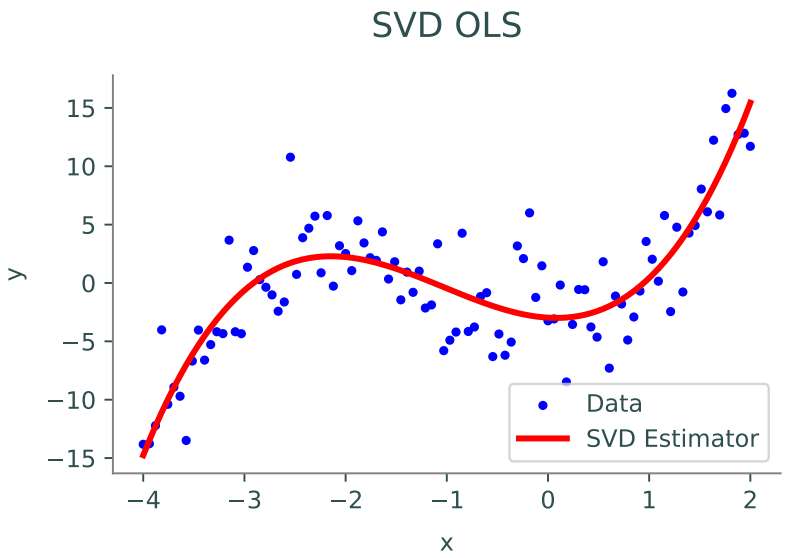


Figure 5.1: This is the SVD estimator for a cubic polynomial.

Model Accuracy

Residual Sum of Squares

The *Residual Sum of Squares* (RSS) is a common choice of measure for the quality of a model. The formula for RSS is given by

$$RSS = ||\boldsymbol{y} - X\widehat{\boldsymbol{\beta}}||_2^2.$$

Notice that the RSS measures the variance in the error of the model. So relative to other models, a smaller RSS value indicates a more accurate model.

Coefficient of Determination

Another method of model accuracy is the *Coefficient of Determination*, denoted R^2. In the case of linear regression,

$$R^2 = 1 - \frac{RSS}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the sample mean of $\boldsymbol{y}$. The intuition of R^2 is that the ratio of the average residual and biased sample variance of $\boldsymbol{y}$ is approximately the total variance explained by the model. A larger R^2 corresponds to a model that fits better. However, R^2 comes with flaws such as being able to take negative values, rewarding overfitting, and punishng under-fit models. Because of this, we typically want to use other methods for model accuracy.

Python Example

There are various python packages that can be used to calculate R^2, but we will use `statsmodels` in this lab. Below is an example of how to build a model and extract R^2 using `statsmodels`.

```
import statsmodels.api as sm
data = pd.read_csv("/filepath") # Read in data as pandas dataframe
y = data["dependent_variable"] # Extract dependent variable
temp_X = data[["var_1", ..., "var_n"]] # Extract independent variables
X = sm.add_constant(temp_X) # Add column of 1's
model = sm.OLS(y, X).fit() # Fit the linear regression model
print(model.rsquared) # Print the R squared value
```

Problem 5. The file `realestate.csv` contains transaction data from 2012-2013. It has columns for transaction data, house age, distance to nearest MRT station, number of convenience stores, latitude, longitude, and house price of unit area. [a] Each row in the array is a separate measurement. As independent variables, use `house age`, `distance to the nearest MRT station`, `number of convenience stores`, `latitude`, and `longitude`.

Find the combination of independent variables that builds the model with the best R^2 value when predicting `house price of unit area`, the dependent variable. Use `statsmodels` to build each model and calculate R^2. Using the same combination of variables, time the methods `ols`, `svd_ols`, and `statsmodels`. Return a list with the first element being a tuple of times for each method and the second element being the best R^2 value from the first part of the problem.

Note that R^2 cannot get worse by adding more columns and also rewards overfitting, so solving for the R^2 value isn't the greatest in practice. The purpose of this problem is to explore the issues of using R^2.

Hint: The `combinations` method from the `itertools` package will be very helpful for finding all feature combinations.

[a] See `https://www.kaggle.com/datasets/quantbruce/real-estate-price-prediction?resource=download`.

Feature Selection

Every regression model consists of features or variables used to predict a dependent variable or result. An important question to ask when building regression models is, which features are the most important in predicting the dependent variable? In addition to being used for model accuracy, R^2 can also be used in feature selection, as it was in Problem 5. It still has the same pitfalls of rewarding overfitting and punishing under-fit models, but it can be a useful tool used in conjunction with the following tools for feature selection. While there are other methods for implementing feature selection, most incorporate the p-value and are not included in this lab.

Akaike's Information Criterion (AIC)

A simple motivation for AIC is based on balancing goodness of fit and prescribing a penalty for model complexity. A more rigorous motivation for AIC is given in Volume 3 using the *Kullback-Leibler* (KL) divergence. Given two models, f and g, the KL divergence is given by

$$KL(f,g) = \int f(z) \log\left(\frac{f(z)}{g(z)}\right) dz$$

and it measures the amount of information lost when g is used to model f. Thus, a lower AIC value indicates a better model. Additionally, AIC penalizes the size of the parameter space with a coefficient of 2 which allows for slightly more complex models.

Bayesian Information Criterion (BIC)

Instead of estimating the KL-divergence between the model in question and the true model, BIC has the property of being minimized precisely when the posterior probability of a model, given the data, is maximized. The equations for AIC and BIC only differ with one term: the coefficient weighting the size of the parameter space. The coefficient for BIC is $log(n)$ which is generally much larger than 2. As a result, BIC penalizes complex models more than AIC. The difference in AIC and BIC values will grow from having more data points.

When using AIC or BIC for feature selection, you need to consider how you want to penalize features in your model. If you want to exclude irrelevant features, then use BIC. If you want to keep all features that are relevant, then use AIC. In other words, BIC is more likely to choose too small a model, and AIC is more likely to choose too large a model.

Python Example

There are multiple ways to calculate AIC and BIC with various python packages. We will use the package `statsmodels` for the following problem. When constructing X for `statsmodels`, do not add the column of 1's manually because `statsmodels` has a method that will do this for us.

```
import statsmodels.api as sm
data = pd.read_csv("/filepath") # Read in data as pandas dataframe
y = data["dependent_variable"] # Extract dependent variable
temp_X = data[["var_1", ..., "var_n"]] # Extract independent variables
X = sm.add_constant(temp_X) # Add column of 1's
model = sm.OLS(y, X).fit() # Fit the linear regression model
print(model.aic) # or print(model.bic)
```

Problem 6. Use the file `realestate.csv` and the Python Example above as a template for constructing `y` and `X` and calculating model AIC and BIC. For the dependent variable, use `house price of unit area`. For the independent variables, use `house age`, `distance to the nearest MRT station`, `number of convenience stores`, `latitude`, and `longitude`. Loop through all of the combinations of these variables and create `OLS` models for each of these combinations. Solve for the `AIC` and `BIC` for each of these models.

Find the model that has the lowest AIC and the model that has the lowest BIC. Are they the same model? Print the features of the model with the lowest AIC as a list.

Hint: The `combinations` method from the `itertools` package will be very helpful for finding all feature combinations.

Regularization

Up to this point, we have been solving the problem

$$\min_{\boldsymbol{\beta}} ||X\boldsymbol{\beta} - \boldsymbol{y}||_2^2.$$

However, we have also assumed independence among the features used to predict the dependent variable. The pitfall of multicollinearity arises when the features of X have dependence and X becomes nearly singular. As a result, the least squares estimator is susceptible to random noise or error. Multicollinearity typically occurs when data is collected with poor experimental design. It is important to have good experimental design, but regularization can be used to mitigate poor design. Another issue OLS faces is feature selection. While there are feature selection methods available, regularization can be used to minimize non-zero coefficients.

Ridge Regularization Regression

The problem posed by *Ridge Regularization* is

$$\min_{\boldsymbol{\beta}} ||X\boldsymbol{\beta} - \boldsymbol{y}||_2^2 + \alpha||\boldsymbol{\beta}||_2^2$$

where $\alpha \geq 0$. This essentially penalizes the size of the coefficients. The larger α is, the more the model resists multicollinearity.

Lasso Regularization Regression

The problem posed by *Lasso Regularization* is

$$\min_{\boldsymbol{\beta}} \frac{1}{n}||X\boldsymbol{\beta} - \boldsymbol{y}||_2^2 + \alpha||\boldsymbol{\beta}||_1.$$

Note that α provides the same functionality here as it does in Ridge Regularization. However, the use of the 1-norm often results in sparse solutions. As a result, Lasso Regularization can be used for feature selection since it only includes the most important features.

Python Example

Since α is not a fixed value in Ridge and Lasso Regularization, it is best practice to perform a Grid-Search to find the best parameter value. The example below goes over the syntax for implementing Ridge Regularization. Note that the syntax for Lasso Regularization is similar.

```
>>> from sklearn import linear_model
>>> y = # dependent variable data
>>> X = # independent variable data with no column of ones
>>> reg = linear_model.RidgeCV(alphas=np.logspace(-6, 6, 13)) # Range for grid ↩
    search
>>> reg.fit(X, y) # Fit the model
>>> reg.alpha_ # Best parameter value
```

Problem 7. Use Ridge and Lasso Regression to model `house price of unit area` from the file `realestate.csv`. First, do a grid search for the model parameter for both the Ridge and the Lasso models, separately. Note that the objects `RidgeCV` and `LassoCV` have built in cross-validation. Be sure to pass in `alphas=np.logspace(-6, 6, 13)` as grid parameters for each of the models. Then use the grid search result to fit each model. Once you have fit the model, you can use the `score` method to get R^2. Print R^2 for each model as a tuple.

6 Logistic Regression

Lab Objective: *Understand the basic principles of Logistic Regression and binary classifiers. Apply this to a dataset.*

Linear regression is unsuitable for predicting probabilities, because the resulting model may take values in any fixed interval in $\mathbb{R}$, but a probability-predicting model can only take values in the interval $[0, 1]$. *Logistic regression* is a form of regression that always takes its values in the interval $[0, 1]$ and as such, is a popular method for predicting probabilities and for constructing *classifiers*. As in linear regression, in a classification problem we have a random variable Y, conditioned on an input $X \in \mathbb{R}^d$. However, in *binary classification* problems the random variable Y is binary, that is, $Y \in \{0, 1\}$. A *binary classifier* is any function f taking values in $\{0, 1\}$. For example, $\mathbf{x} \in \mathbb{R}^d$ could be the pixel intensities of an image, and the classifier f gives 1 if the image is a picuture of a duck and 0 otherwise. The goal of a classification problem is to choose a classifier $\widehat{f}$ so that $(X, \widehat{f}(X))$ is a good approximation for (X, Y).

Logistic Regression

Logistic regression relies heavily on the *logistic function*, also known as the *sigmoid function*, $\text{sigm} : \mathbb{R} \to (0, 1)$ given by

$$\text{sigm}(x) = \frac{1}{1 + e^{-x}}. \tag{6.1}$$

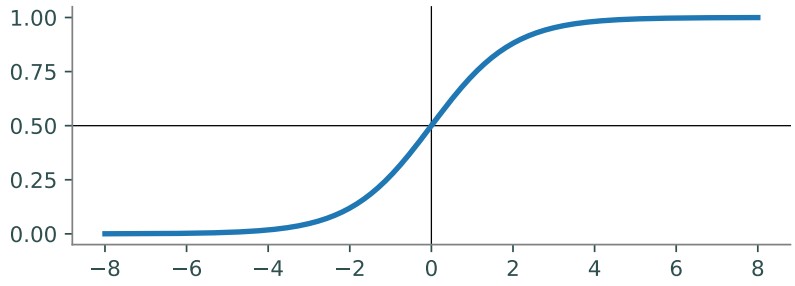


Figure 6.1: Sigmoid Function

This function works well for classifying objects based on probabilities, because it has some key properties that translate well into probability theory. Of particular note, the graph can translated by adding a constant, giving the form $\text{sigm}(\beta_1 t + \beta_0)$. A larger value of β_1 makes the ramp up from 0 to 1 steeper, while a smaller value of β_1 makes it less steep. The trick behind logistic regression is to find the values of β_i such that the resulting sigmoid function best classifies the data.

In logistic regression models we have a random variable Y with support $\{0, 1\}$, where Y is conditioned on another random variable X, with support in $\mathbb{R}^d$. The distribution of Y, given X, is assumed to be Bernoulli

$$Y \mid X \sim \text{Bernoulli}(\text{sigm}(X^\mathsf{T}\boldsymbol{\beta})),$$

so that

$$P(Y \mid X) = \text{sigm}(X^\mathsf{T}\boldsymbol{\beta}) = \frac{1}{1 + \exp(-X^\mathsf{T}\boldsymbol{\beta})}.$$

As in the case of linear regression, we usually add a constant feature $X_0 = \mathbf{1}$ to X and a corresponding coefficient β_0 to $\boldsymbol{\beta}$, so that $X^\mathsf{T}\boldsymbol{\beta} = \beta_0 + \beta_1 X_1 + \cdots + \beta_d X_d$. Given a draw of length n of the form $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ we wish to estimate $\boldsymbol{\beta}$. The maximum likelihood estimator is a good choice. To find this estimator, first observe that the likelihood of $\boldsymbol{\beta}$, given the data, is

$$\begin{aligned} L(\boldsymbol{\beta} \mid D) &= \prod_{i=1}^{n} P(Y = y_i, X = \mathbf{x}_i \mid \boldsymbol{\beta}) \\ &= \prod_{i=1}^{n} P(Y = y_i \mid X = \mathbf{x}_i, \boldsymbol{\beta}) P(X_i). \end{aligned}$$

which is equivalent to maximizing

$$\prod_{i=1}^{n} P(Y = y_i \mid X = x_i, \boldsymbol{\beta}) = \prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1 - y_i}.$$

where

$$p_i = P(Y = 1 \mid \mathbf{x}_i, \boldsymbol{\beta}) = \text{sigm}(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}) = \frac{1}{1 + \exp(-\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})}.$$

Taking the negative logarithm turns this into a convex minimization problem, and a little math shows that

$$\ell(\boldsymbol{\beta} \mid D) = \sum_{i=1}^{n} \left(y_i \log(1 + \exp(-\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})) + (1 - y_i) \log(1 + \exp(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})) \right). \tag{6.2}$$

The convexity of this problem implies there is a unique minimizer $\widehat{\boldsymbol{\beta}}$ of $\ell(\boldsymbol{\beta} \mid D)$.

Problem 1. Write a `fit()` method in the Python `LogiReg` classifier that accepts an $(n \times 1)$ array `y` of binary labels (0's and 1's) as well as an $(n \times d)$ array `X` of data points that uses equation 6.2 to find and save the optimal $\widehat{\boldsymbol{\beta}}$. Save `X`, `y`, and $\widehat{\boldsymbol{\beta}}$ as attributes. Remember to add a column of ones to `X` before implementing the `fit` algorithm.

Once the maximum likelihood estimate $\widehat{\boldsymbol{\beta}}$ is found, we have an estimate for the probability

$$P(Y = 1 \mid \mathbf{x}) \approx \text{sigm}(\mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\beta}}).$$

From this, we can construct a classifier $\widehat{f}$ by setting $\widehat{f}(x) = 1$ if $P(Y = 1 \mid \mathbf{x}) \geq \frac{1}{2}$ and $\widehat{f}(x) = 0$ otherwise.

Problem 2. Write a method called `predict_prob()` for your classifier that accepts an $(n \times d)$ array `x_test` and returns $P(Y = 1 \mid$ `x_test`$)$. Also write a method called `predict()` that calls `predict_prob()` and returns an array of predicted labels (0's or 1's) for the given array `x_test`. Remember to add a column of ones to `x_test` in your `predict` function like you did in the previous problem with `X`.

Problem 3. To test your classifier, create training arrays `X` and `y` as well as a testing array `X_test`. The code to generate `X`, `y` and `X_test` is provided below. Both `X` and `X_test` have 100 random draws from a 2-dimensional multivariate normal distribution centered at $(1, 2)$, and another 100 draws from one centered at $(4, 3)$.

Train your classifier on `X` and `y`. Then generate a list of predicted labels using your trained classifier and `X_test`, and use it to plot `X_test` with a different color for each predicted label. Your plot should look similar to Figure 6.2. If you didn't add a constant column in Problem 1, go back and do that. This will allow the decision boundary between the two classes to not intersect the origin.

```
>>> import numpy as np

>>> data = np.column_stack((
        # draw from 2 2-dim. multivariate normal dists.
        np.concatenate((
        np.random.multivariate_normal(np.array([1,2]), np.eye(2), 100),
        np.random.multivariate_normal(np.array([4,3]), np.eye(2), 100)
        )),
        # labels corresonding to each distribution
        np.concatenate(( np.zeros(100), np.ones(100) )) ))
>>> np.random.shuffle(data)
>>> # extract X and y from the shuffled data
>>> X = data[:,:2]
>>> y = data[:,2].astype(int)

>>> X_test = np.concatenate((
        # draw from 2 identical 2-dim. multivariate normal dists.
        np.random.multivariate_normal(np.array([1,2]), np.eye(2), 100),
        np.random.multivariate_normal(np.array([4,3]), np.eye(2), 100)
        ))
>>> np.random.shuffle(X_test)
```

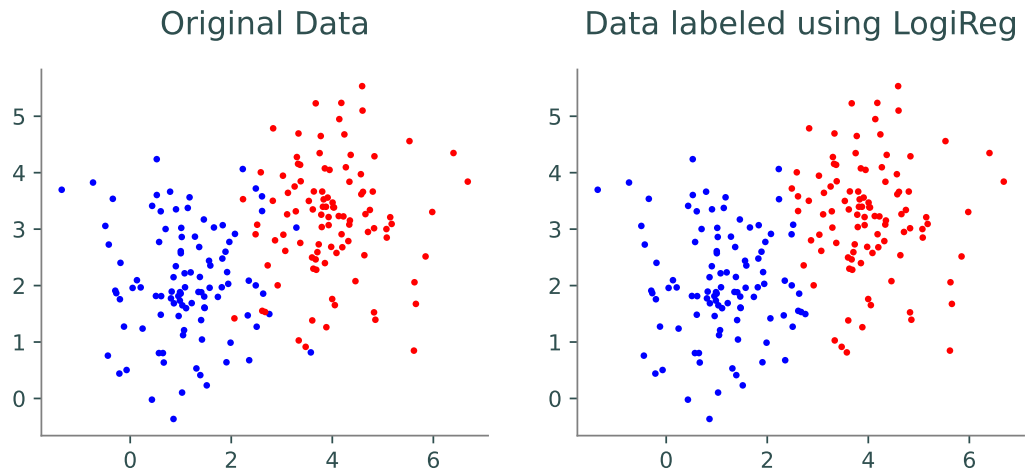


Figure 6.2: In reality, these two distributions overlap a little, but the logistic regression model makes a clean divide between the two.

Statsmodels and Sklearn

The module `statsmodels` contains a package that includes a logistic regression class called `Logit`. A simple example of this class being implemented is as follows.

```
>>> import statsmodels.api as sm

>>> model = sm.Logit(y, X).fit(disp=0)  # setting disp=0 turns off printed info
>>> probs = model.predict(X_test)  # list of probabilities, not labels
```

`Logit` does *not* add a constant feature (column of 1's) to `X` by default, so in order to do so, you must apply the function `sm.add_constant()` to both `X` and `X_test`. In addition, the `.fit()` method does not regularize the problem by default, which may lead to some errors involving singular matrices. To fix this, you can use the `.fit_regularized()` method instead of `.fit()`.

The module `sklearn` also has a package for logistic regression called `LogisticRegression`, which can be implemented as follows.

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

>>> model = LogisticRegression(fit_intercept=True).fit(X, y)  # X before y
>>> labels = model.predict(X_test)  # predicted labels of X_test
```

`LogisticRegression` already regularizes the problem by default. The parameter `fit_intercept` (which defaults to `False`) indicates whether you want to add a constant feature (column of 1's) to `X` and `X_test`.

You can also use `sklearn` to score a logistic regression model. After fitting an `sklearn` model, you can call `<model>.score(X_test, y_test)` to find the percentage of accuracy of the model's prediction for `X_test`, given the true labels in `y_test`. Alternatively, you can use `sklearn.metrics.accuracy_score` to find the percentage of accuracy between a list of predicted labels and the list of true labels.

```
>>> from sklearn.metrics import accuracy_score

>>> true_labels = [0, 1, 2, 3, 4]
>>> pred_labels = [0, 2, 2, 2, 4]  # predicted labels from logistic regression
>>> accuracy_score(true_labels, predicted_labels)
0.6
```

Support Vector Machines

Support Vector Machines (SVM) are another type of classifier. It uses what is called the 'kernel trick' to handle nonlinear input spaces. It classifies data by finding an optimal hyperplane to split the data.

```
>>> from sklearn import svm

>>> svm = svm.SVC(kernel = 'linear').fit(X, y)
>>> labels = svm.predict(X_test)          # predicted labels of X_test
```

Problem 4. The code to generate arrays `X`, `y`, `X_test`, and `y_test` is provided below. `X` and `X_test` are each composed of 200 draws from two 20-dimensional multivariate normal distributions, one centered at **0**, and the other centered at **2**.

Using each of `LogiReg`, `statsmodels`, `sklearn.LogisticRegression`, and `sklearn.svm`, train a logistic regression classifier on `X` and `y` to generate a list of predicted labels for `X_test`. Then, using `y_test`, print the accuracy scores for each trained model. Compare the accuracies and training/testing time for all four classifiers. Be sure to add a constant feature with each model.

```
>>> # predefine the true beta
>>> beta = np.random.normal(0, 7, 20)

>>> # X is generated from 2 20-dim. multivariate normal dists.
>>> X = np.concatenate((
        np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
        np.random.multivariate_normal(np.ones(20)*2, np.eye(20), 100)
        ))
>>> np.random.shuffle(X)
>>> # create y based on the true beta
>>> pred = 1. / (1. + np.exp(-X @ beta))
>>> y = np.array( [1 if pred[i] >= 1/2 else 0
```

```
            for i in range(pred.shape[0])] )

>>> # X_test and y_test are generated similar to X and y
>>> X_test = np.concatenate((
        np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
        np.random.multivariate_normal(np.ones(20), np.eye(20), 100)
        ))
>>> np.random.shuffle(X_test)
>>> pred = 1. / (1. + np.exp(-X_test @ beta))
>>> y_test = np.array( [1 if pred[i] >= 1/2 else 0
            for i in range(pred.shape[0])] )
```

Hint: Consider using the command `fit_regularized(disp = 0)` for the `statsmodels` case.

Multiclass Logistic Regression

Sometimes we may want to classify data into more than two categories, but so far we've only used logistic regression as a binary classifier. The good news is that we can extend logistic regression to classify more than just two categories.

The more popular method for doing this is to generalize the logistic regression model to a multiclass setting. This method is called *multinomial logistic regression* or sometimes *softmax regression*. While standard logistic regression was based on the sigmoid function, multinomial logistic regression is based on the *softmax function* $\mathscr{S} : \mathbb{R}^k \to (0,1)^k$, which is a multivariate version of the sigmoid function, given by

$$\mathscr{S}(t_1, \ldots, t_k) = \left(\frac{e^{t_1}}{\sum_{j=1}^k e^{t_j}}, \ldots, \frac{e^{t_k}}{\sum_{j=1}^k e^{t_j}} \right). \tag{6.3}$$

We will assume that $Y \mid X$ is categorically distributed as

$$\mathrm{Cat}(p_1(X), \ldots, p_k(X)) = \mathrm{Cat}(\mathscr{S}(X^\mathsf{T}\boldsymbol{\beta}_1, \ldots, X^\mathsf{T}\boldsymbol{\beta}_k))$$

for some choice of vectors $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_k$, which we will estimate from the data. Here

$$p_i(X) = P(Y = i \mid X) = \frac{e^{X^\mathsf{T}\boldsymbol{\beta}_i}}{\sum_{j=1}^k e^{X^\mathsf{T}\boldsymbol{\beta}_j}} = \frac{\mathrm{sigm}(X^\mathsf{T}\boldsymbol{\beta}_i)}{\sum_{j=1}^k \mathrm{sigm}(X^\mathsf{T}\boldsymbol{\beta}_j)}.$$

Given a draw of length n of the form $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, we wish to compute $\boldsymbol{\theta} = (\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_k)$ where, without loss of generality, we may assume $\boldsymbol{\beta}_k = \mathbf{0}$. The maximum likelihood estimate of $\boldsymbol{\theta}$ is computed in a manner similar to the way it was for standard logistic regression. A bit of math shows that

$$\begin{aligned} \ell(\boldsymbol{\theta} \mid D) &= -\sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log(p_j(\mathbf{x}_i)) \\ &= -\sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log\left(\frac{e^{\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}_j}}{\sum_{m=1}^k e^{\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}_m}} \right) \end{aligned}$$

where

$$\delta_{c_j}(y_i) = \begin{cases} 1 & \text{if } y_i = c_j, \text{ the jth class} \\ 0 & \text{otherwise.} \end{cases}$$

This is a convex minimization problem with unique minimizer $\widehat{\boldsymbol{\theta}}$. Once $\widehat{\boldsymbol{\theta}} = (\widehat{\boldsymbol{\beta}_1}, \ldots, \widehat{\boldsymbol{\beta}_k})$ is found, we have an estimate for the probability

$$P(Y = y \mid \mathbf{x}) \approx \frac{e^{\mathbf{x}^\intercal \widehat{\boldsymbol{\beta}}_y}}{\sum_{j=1}^{k} e^{\mathbf{x}^\intercal \widehat{\boldsymbol{\beta}}_j}}.$$

From this, we can construct a classifier $\widehat{f}$ by setting $\widehat{f}(\mathbf{x}) = \text{argmax}_j P(Y = c_j \mid \mathbf{x})$.

Conveniently, `sklearn` has a very simple implementation of multinomial logistic regression that simply requires the argument `multi_class='multinomial'` when initiating a `LogisticRegression` model.

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

>>> model = LogisticRegression(
                multi_class='multinomial',
                fit_intercept=True).fit(X, y) # add constant feature
```

Problem 5. The Iris Dataset contains information taken from 150 samples of 3 different types of iris flowers (Setosa, Versicolor, and Virginica). The columns contain measurements for sepal length, sepal width, pedal length, and pedal width. Import the Iris Dataset and perform a train-test split on only the first two columns of the data with `test_size=0.4`. Train a multinomial logistic regression model using the training data with an added constant feature, and generate prediction labels for the test data. Plot the test data by color using your prediction labels. Also, print the model score. Reference code below for accessing the Iris Dataset and for using `train_test_split`.

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split

>>> iris = datasets.load_iris()

>>> X = iris.data[:, :2]  # we only take the first two features.
>>> y = iris.target

>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size↩
    =0.4)
```

Your plot should reflect Figure 6.3

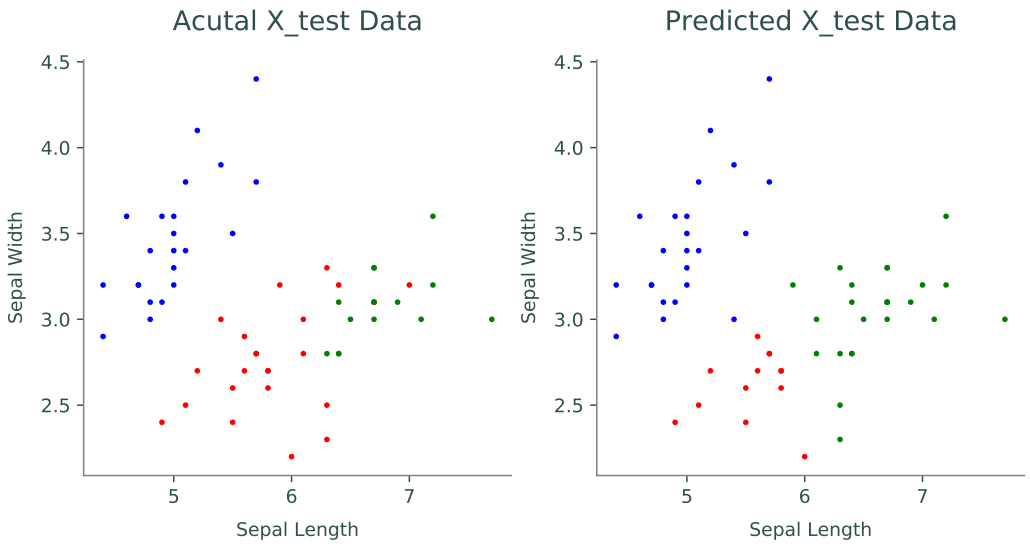


Figure 6.3: Multinomial logistic regression attempt to categorize the Iris Dataset.

7 Naive Bayes

Lab Objective: *In this lab, we will create a Naïve Bayes Classifier and use it to make an SMS spam filter.*

Naïve Bayes Classifiers

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

While naïve Bayes classifiers are most easily seen as applicable in cases where the features have, ostensibly, well defined probability distributions (such as classifying sex given physical characteristics), they are applicable in many other cases. In this lab, we will apply them to the problem of spam filtering. While it is generally a bad idea to assume independence, naïve Bayes classifiers can still be very effective, even when we can be confident that features are not independent.

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

$$\text{argmax}_{k\in K} P(C = k|\mathbf{x}), \tag{7.1}$$

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

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

Now we will examine each feature individually and use the chain rule to expand the new conditional probability:

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

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

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

Therefore we have that

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

which reforms Equation 7.1 as

$$\text{argmax}_{k\in K}P(C=k|\mathbf{x})=\text{argmax}_{k\in K}P(C=k)\prod_{i=1}^{n}P(x_i|C=k). \tag{7.2}$$

We can drop the $P(\mathbf{x})$ in the denominator since it does not depend on k. In this form, the problem is computationally tractable, since we can use the training data to find approximations of $P(C=k)$ and $P(x_i|C=k)$ for each i and k. Something to note here is that we are actually maximizing $P(C=k|\mathbf{x})$ by computing and maximizing $P(C=k,\mathbf{x})$. This means that naïve Bayes is a generative classifier, and not a discriminative classifier.

Spam Filters

A spam filter is just a special case of a classifier with two classes: spam and not spam (often called ham). Spam filtering is a situation where naive Bayes classifiers perform relatively well. Throughout the lab, we will use the SMS spam dataset contained in `sms_spam_collection.csv`. The messages in this dataset have already been cleaned by converting to lowercase and removing all punctuation. To load the dataset, use the following code:

```
import pandas as pd
df = pd.read_csv('sms_spam_collection.csv')

# separate the data into the messages and labels
X = df.Message
y = df.Label
```

Before we can construct a naive Bayes classifier, we need to choose a probability distribution for the x_i. Two common choices are categorical distributions and Poisson distributions. We will first create a classifier using a categorical distribution. In this case, each feature x_i represents the i-th word of a message. The probability $P(x_i|C=k)$ then represents the probability that a specific word in the message is the word that we observed, given that the category is k. For simplicity, we assume that these values do not change with respect to i, so the probability of observing a specific word is the same for every position in a message.

Suppose we have a labeled training dataset. To train the model, we need to just find values for $P(C=k)$ and $P(x_i|C=k)$. In this case, a reasonably good choice is the maximum likelihood estimator, which in this case is just

$$P(C=k)=\frac{N_{\text{samples in }k}}{N_{\text{samples}}}$$

$$P(x_i|C=k)=\frac{N_{\text{occurrences of }x_i\text{ in class }k}}{N_{\text{words in class }k}}$$

However, this choice leads to some issues. For example, if a certain word x_j occurs only in spam messages and never in ham messages in our training dataset, then our classifier will predict $P(x_i|C = \text{ham}) = 0$ for any message that contains x_j. This is not desirable, but to make matters worse, the same situation could happen for both classes within a single message, leading our model to predict $P(x_i|C = k) = 0$ for all classes. This makes our classifier unable to classify such samples. To circumvent this issue, we will use *Laplace add-one smoothing*, which consists of adding 1 to the numerator of $P(x_i|C = k)$ and 2 to its denominator. So, the probabilities we will use are the following:

$$P(C = \text{spam}) = \frac{N_{\text{messages in spam}}}{N_{\text{samples}}}, \tag{7.3}$$

$$P(C = \text{ham}) = \frac{N_{\text{messages ham}}}{N_{\text{samples}}}, \tag{7.4}$$

$$P(x_i|C = k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k} + 1}{N_{\text{total words in class } k} + 2}. \tag{7.5}$$

The result of Laplace add-one smoothing is equivalent to the maximum likelihood estimators if a certain Bayesian prior is used for the probabilities. We don't use this for the $P(C = k)$, since it is not really needed for those probabilities and does not lead to any particular benefit. Lastly, note that the denominator in Equation 7.5 is not the number of unique words in class k, but the total number of occurrences of any word in class k.

Problem 1. Create a class `NaiveBayesFilter`. Implement a method `fit()` that accepts the training data `X` and the training labels `y`. In this method, compute the probabilities $P(C = \text{spam})$ and $P(C = \text{ham})$ as in Equations (7.3) and (7.4). Also compute the probabilities $P(x_i|C = k)$ for each word in both the spam and ham classes, thereby training the model. Store these computed probabilities in dictionaries called `self.spam_probs` and `self.ham_probs`, where the key is the word and the value is the associated probability.

For example, `self.ham_probs['out']` will give the computed value for $P(x_i = \text{“out”}, C = \text{ham})$ value:

```
# Example model trained on the first 300 data points
>>> nb = NaiveBayesFilter()
>>> nb.fit(X[:300], y[:300])

# Check spam and ham probabilities of 'out'
>>> nb.ham_probs['out']
0.003147128245476003
>>> nb.spam_probs['out']
0.004166666666666667
```

Hint: be sure you count the number of occurrences of a word, and not simply of a string. For example, when searching the string `'find it in there'` for the word `'in'`, make sure you get 1 and not 2 (because of the `'in'` in `'find'`). The methods `pd.Series.str.split()` and `count()` may be helpful. When using `split()`, call it without any arguments, as otherwise you may accidentally add empty strings to your data.

Predictions

Now that we have trained our model, we can predict the class of a message by calculating

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

for each class k, then choosing the class that maximizes this probability. As discussed above, this is equivalent to maximizing the probability $P(C = k|\mathbf{x})$; however, be aware that we are not actually computing those. The probabilities we compute here will not sum to 1, since they are actually the values $P(C = k, \mathbf{x})$.

However, directly computing this probability as a product can lead to an issue: underflow. If $\mathbf{x}$ is a particularly long message, then, since we are multiplying lots of numbers between 0 and 1, it is possible for the computed probability to *underflow*, or become too small to be machine representable with ordinary floating-point numbers. In this case the computed probability becomes 0. This is particularly problematic because if underflow happens for a sample for one class, it will likely also happen for all of the other classes, making such samples impossible to classify. To avoid this issue, we will work with the logarithm of the probability:

$$\ln P(\mathbf{x}, C = k) = \ln \left(P(C = k) \right) + \sum_{i=1}^{n} \ln \left(P(x_i | C = k) \right). \tag{7.6}$$

This has the same maximizer as before, so our predictions are unaffected, while also avoiding any issues with underflow.

Problem 2. Implement the `predict_proba()` method in your naïve Bayes classifier. It should take as an argument `X`, the data that needs to be classified, and it will compute the log probabilities as given in Equation 7.6 for each message in `X`. In the case we have some word x_u that is not found in the training set, use the value $P(x_u|C = k) = \frac{1}{2}$ for both classes.

The method should return an `(N, 2)` array, where `N` is the length of `X`, whose entries are the log probabilities of each message `x` in `X` belonging to each category. The first column corresponds to $\ln P(\mathbf{x}, C = \text{ham})$, and the second to $\ln P(\mathbf{x}, C = \text{spam})$.

Your code should produce the following output with the example from Problem 1:

```
>>> nb.predict_proba(X[800:805])
array([[ -30.8951931 ,  -35.42406687],
       [-108.85464069,  -91.70332157],
       [ -74.65014875,  -88.71184709],
       [-164.94297917, -133.8497405 ],
       [-127.17743715, -101.32098062]])
```

Hint: The dataframe `X`'s index might not be in order or consecutive integers, so accessing its rows as `X[i]` may lead to errors later. Using `for row in X` or similar to iterate will work better.

Problem 3. Implement the method `predict()`. Accept as an argument `X`, the data to be classified, and return an array with the same shape holding the predicted class for each sample in `X`. The entries of this array should be strings `'spam'` and `'ham'`. Use your method `predict_proba()` to compute the log probabilites of each class. In case of a tie, predict `'ham'`.

Your code should produce the following output with the example from Problem 1:

```
>>> nb.predict(X[800:805])
array(['ham', 'spam', 'ham', 'spam', 'spam'], dtype=object)
```

Problem 4. We now test our spam filter. Use the `sklearn`'s `train_test_split` function with the default parameters to split the data into training and test sets. Train a `NaiveBayesFilter` on the train set, and have it predict the labels of each message in the test set. Compute the answer to the following questions:

- What proportion of the spam messages in the test set were correctly identified by the classifier?
- What proportion of the ham messages were incorrectly identified?

Return the answers to these questions as a tuple.

The Poisson Model

Now that we've examine one way to constructing a naïve Bayes classifier, let us look at one more method. In the Poisson model we assume that each word in the vocabulary is Poisson random variable, occurring with potentially different frequencies among spam and ham messages. Because each of the messages is a different length, we can reparameterize the Poisson PMF to the following

$$P(n_i = x) = \frac{(rn)^x e^{-rn}}{x!} \tag{7.7}$$

where n_i is the number of times word i occurs in a message, n is the length of the message, and $\lambda = rn$ is the classical Poisson rate. In this case r represents the number of events per unit time/space/etc. We will again use maximum likelihood estimation to find the values of r for each word and class.

Training the Model

Similar to the other classifier that we made, training the model amounts to using the training data to determine how $P(x_i|C = k)$ is computed, as well as compting $P(C = k)$. For the Poisson model we must find a value for r for each word that appears in the training set. To do this we will use maximum likelihood estimation. We will label the chosen value of r for the i-th word and class k as $r_{i,k}$. In this case, since we are using a Poisson distribution (7.7) for each word, we will solve the following problem for both the spam class and the ham classes

$$r_{i,k} = \text{argmax}_{r\in[0,1]} \frac{(rN_k)^{N_{i,k}} e^{-rN_k}}{N_{i,k}!}, \tag{7.8}$$

where $N_k = N_{\text{total words in class } k}$ is the total number of words in class k and $N_{i,k} = N_{\text{occurences of word } i \text{ in class } k}$ is the number of times the i-th word occurs in class k. We have $r \in [0, 1]$ because a word cannot occur more than once per word in the message. It can then be shown that the maximizing value is

$$r_{i,k} = N_{\text{occurences of word } i \text{ in class } k} / N_{\text{total words in class } k}.$$

However, observe that in Equation (7.7), if we have $r = 0$ then $P(n_i = x) = 0$ whenever $x > 0$. This leads to the exact same issue we saw with the categorical approach, and will lead to predicted probabilities being 0. To resolve this issue, we will again use Laplace add-one smoothing and instead use the values

$$r_{i,k} = \frac{N_{\text{occurences of word } i \text{ in class } k} + 1}{N_{\text{total words in class } k} + 2}. \tag{7.9}$$

As before, this can be interpreted as the maximum likelihood estimator if we start with a certain Bayesian prior for r.[1]

Making predictions with this model is exactly the same as we did earlier, albeit with slightly different equations. Specifically, if we write Equation 7.6 with the Poisson probability, our prediction is given by

$$\text{argmax}_{k \in K} \ln\left(P(C = k)\right) + \sum_{i \in \text{Vocab}} \ln\left(\frac{(r_{i,k} n)^{n_i} e^{-r_{i,k} n}}{n_i!}\right), \tag{7.10}$$

with n_i the number of times the i^{th} word occurs in the message, n the total number of words in the message, and $r_{i,k}$ the Poisson rate of the i^{th} word in class k.

Problem 5. Create a new class called `PoissonBayesFilter`, with three methods called `fit()`, `predict_proba()`, and `predict()`, analogous to those in the `NaiveBayesFilter` class.

Implement `fit()` by finding the MLE found in Equation 7.9 to predict r for each word in both the spam and ham classes, thereby training the model. Store these computed rates in dictionaries called `self.spam_rates` and `self.ham_rates`, where the key is the word and the value is the associated r.

For example, `self.ham_rates['in']` will give the computed r value for the word "in" found in ham messages.

```
# Example model trained on the first 300 data points
>>> pb = PoissonBayesFilter()
>>> pb.fit(X[:300], y[:300])

# Check spam and ham rate of 'in'
>>> pb.ham_rates['in']
0.012588512981904013
>>> pb.spam_rates['in']
0.004166666666666667
```

Implement the `predict_proba()` and `predict()` methods using equation 7.10. These methods will take the same arguments and return the same object types as the methods of the same name in the `NaiveBayesFilter` class. If a word u not in the training set is in one of the messages, use the value $r_{u,k} = 1/(N_k + 2)$. In case of a tie in the probabilities of two classes, predict `'ham'`. Your code should produce the following output with the example above:

[1] Note that these rates are exactly the same as the probabilities we computed for the categorical model.

```
>>> pb.predict_proba(X[800:805])
array([[ -37.14113097,  -38.2193235 ],
       [-112.61977379,  -83.54702923],
       [ -55.70966168,  -63.83602125],
       [-130.02471282,  -90.15687624],
       [-102.36539804,  -69.55261684]])

>>> pb.predict(X[800:805])
array(['ham', 'spam', 'ham', 'spam', 'spam'], dtype=object)
```

Hint: Most of your code will be very similar to your `NaiveBayesFilter` class. The function `np.unique` with the argument `return_counts=True` and the function `scipy.stats.poisson.logpmf` will be useful for `predict_proba()`.

Problem 6. In the function `prob6()`, use the `sklearn.model_selection.train_test_split` function to split the data into training and test sets. Train a `PoissonBayesClassifier` on the train set, and have it predict the labels of each message in the test set. Compute the answer to the following questions:

- What proportion of the spam messages in the test set were correctly identified by the classifier?
- What proportion of the ham messages were incorrectly identified?

Return the answers to these two questions as a tuple.
How do the performances of the categorical and Poisson models compare?

Naive Bayes with Sklearn

Now that we've explored a few ways to implement our own naïve Bayes classifier, we can examine some robust tools from the sklearn library that will accomplish all the things we've coded up in a very simple manner.

The first thing we need to do is create a dictionary and transform the training data, which is what our first `fit()` method did. We instantiate a `CountVectorizer` model from `sklearn.feature_extraction.text`, and then use the `fit_transform()` method to create the dictionary and transform the training data.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
train_counts = vectorizer.fit_transform(X_train)
```

Now we can use the transformed training data to fit a `MultinomialNB` model from `sklearn.naive_bayes`.

```
from sklearn.naive_bayes import MultinomialNB

clf = MultinomialNB()
clf = clf.fit(train_counts, y_train)
```

Testing data we want to classify must first be transformed by our vectorizer with the `transform()` method (not the `fit_transform()` method). We can then classify the data using the `predict()` method of the `MultinomialNB` model.

```
test_counts = vectorizer.transform(X_test)
labels = clf.predict(test_counts)
```

This naïve Bayes model uses the multinomial distribution where we have used the categorical and poisson distributions. Multinomial is very similar to the categorical implementation, as the multinomial distribution models the outcome of n categorical trials (in the same way that the binomial distribution models n Bernoulli trials).

Problem 7. Write a function that will classify messages using the SkLearn naive Bayes implementation. It will take as arguments training data `X_train` and `y_train`, and test data `X_test`. In this function use the `CountVectorizer` and `MultinomialNB` from SkLearn and return the predicted classification of the model.

8 Metropolis Algorithm

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

The Metropolis Algorithm

Sampling from a given probability distribution is an important part of many tasks throughout the sciences. When modeling real-world problems, these distributions are often very complicated, and direct sampling methods require computing high-dimensional integrals and are thus impractical. The Metropolis algorithm is an effective method to sample from many of these distributions. This algorithm only requires evaluating the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute any difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

Suppose that $h : \mathbb{R}^n \to \mathbb{R}$ is the probability density function of a distribution that is difficult to evaluate (for example, a Bayesian posterior distribution), while some function $f(\boldsymbol{\theta}) = c \cdot h(\boldsymbol{\theta})$ is easy to evaluate. The Metropolis algorithm is an MCMC sampling method which constructs a Markov chain Y whose invariant distribution is exactly the distribution associated with h. We can then use these samples as a sample from this distribution.

For the Metropolis algorithm, we need two ingredients: a *proposal function*, and an *acceptance function*. The proposal function is used to choose a potential next state. We denote this function as $Q(\mathbf{x}, \mathbf{y}) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. For each $\mathbf{y} \in \mathbb{R}^n$, $Q(\cdot, \mathbf{y})$ is the probability density function for the proposed state. This distribution needs to be easy to sample from; typical choices are uniform or normal distributions. For simplicity we also require this function to be *symmetric*, so $Q(\mathbf{x}, \mathbf{y}) = Q(\mathbf{y}, \mathbf{x})$. In words, the probability density of moving from $\mathbf{x}$ to $\mathbf{y}$ is the same as moving from $\mathbf{y}$ to $\mathbf{x}$ for all $\mathbf{x}$ and $\mathbf{y}$.

The acceptance function $A : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ gives the probability that we actually transition from the current state $\mathbf{y}$ to the proposed state $\mathbf{x}$.[1] This function is defined by

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

[1] In the Volume 3 textbook, the proposal function for continuous distributions and the acceptance function are denoted $f_{X_{t+1}|X_t=\mathbf{y}}(\mathbf{x})$ and $a_{\mathbf{x},\mathbf{y}}$ respectively. In this lab we instead write them as $Q(\mathbf{x}, \mathbf{y})$ and $A(\mathbf{x}, \mathbf{y})$ to make it clearer that they are functions of $\mathbf{x}$ and $\mathbf{y}$.

Following the proposals from Q causes us to wander around the space of allowed states. The acceptance function from A modifies this wandering so that we spend more time in more likely regions.

Algorithm 1 Metropolis Algorithm

```
procedure METROPOLIS ALGORITHM
    Choose initial point y_0.
    for t = 1, 2, ... do
        Draw x ~ Q(·, y_{t-1})
        Draw a ~ unif(0, 1)
        if a ≤ A(x, y_{t-1}) then
            y_t = x
        else
            y_t = y_{t-1}
    Return y_1, y_2, y_3, ...
```

These functions form the basis for the Metropolis algorithm. At each step, given our current state $\mathbf{y}_t$, we propose a new state according to the distribution $\mathbf{x} \sim Q(\cdot, \mathbf{y}_t)$. We then accept the proposed state with probability $A(\mathbf{x}, \mathbf{y}_t)$. If we accept the proposal, we set $\mathbf{y}_{t+1} = \mathbf{x}$; otherwise, we set $\mathbf{y}_{t+1} = \mathbf{y}_t$. Refer to Algorithm 1 for a write-up of this algorithm. Under certain conditions on Q, the Markov chain the samples $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3 \ldots$ are from will have a unique invariant distribution with density h, and any initial state will converge to this distribution.

We can consider each of the samples $\mathbf{y}_i$ as draws from the distribution of h. Most of the time we don't just want samples from the distribution, but *independent* samples. However, the samples $\mathbf{y}_t$ and $\mathbf{y}_{t+1}$ are clearly not independent. We can get around this issue by only keeping some of the samples, for example every 10th or 100th sample. While $\mathbf{y}_t$ and $\mathbf{y}_{t+1}$ aren't independent, $\mathbf{y}_t$ and $\mathbf{y}_{t+100}$ will be closer to being independent.

Finally, for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

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

Let's apply the Metropolis algorithm to an example of Bayesian analysis. Consider the exam scores in `examscores.csv`, and suppose that these scores are distributed normally with (unknown) mean μ and variance σ^2. We wish to compute the posterior distribution for μ and σ^2. Denote the data as $d_1, \ldots, d_N$ and assume the prior distributions

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

Note that IG is the inverse gamma distribution. In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 \,|\, d_1, \ldots, d_N) = \frac{p(\mu)p(\sigma^2)\prod_{i=1}^{N}\mathcal{N}(d_i \,|\, \mu, \sigma^2)}{\int_{-\infty}^{\infty}\int_{0}^{\infty} p(\mu')p(\sigma^{2\prime})\prod_{i=1}^{N}\mathcal{N}(d_i \,|\, \mu', \sigma^{2\prime})\, d\sigma^{2\prime}d\mu'}.$$

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:

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

i.e. normally distributed with mean $\mathbf{y}$ and variance uI where I is the 2×2 identity matrix and $u > 0$.

```
def proposal(y, u):
    """Returns the proposal, i.e. a draw from Q(x,y) = N(x|y,uI)."""
    return stats.multivariate_normal.rvs(mean=y, cov=u*np.eye(len(x)))

def propLogDensity(x, muprior, sig2prior, scores):
    """Calculate the log of the proportional density funciton f."""
    if x[1] <= 0:
        return -np.inf
    logprob = muprior.logpdf(x[0]) + sig2prior.logpdf(x[1])
    logprob += stats.norm.logpdf(scores, loc=x[0], scale=np.sqrt(x[1])).sum()
    return logprob

def acceptance(x, y, muprior, sig2prior, scores):
    """
    Returns the acceptance probability of moving from y to x.
    """
    return np.exp(min(0,
                  propLogDensity(x, muprior, sig2prior, scores)
                - propLogDensity(y, muprior, sig2prior, scores)
    ))
```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log probabilities $\log f(\mathbf{y}_t)$ and the proportion of proposed samples that were accepted.

We will evaluate the quality of our results by plotting the log probabilities, the μ samples, the σ^2 samples, and kernel density estimators for the marginal posterior distributions of μ and σ^2. The kernel density estimators approximate the continuous distribution of the marginal distributions. The kernel density estimator for μ should be approximately normal, and the kernel density estimator for σ^2 should be approximately an inverse gamma.

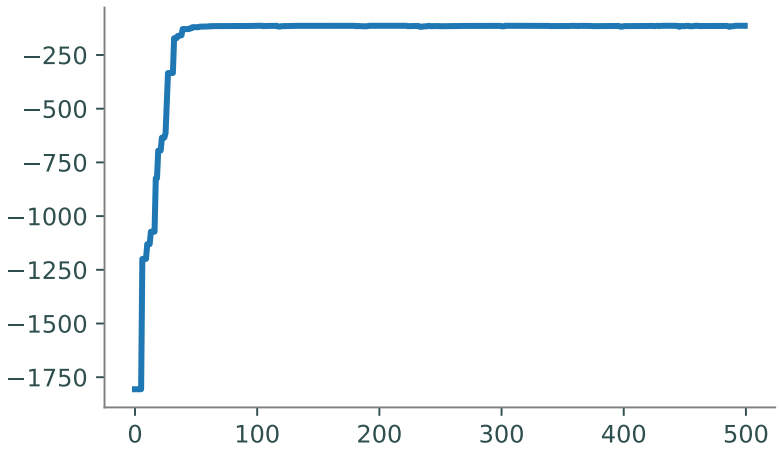


Figure 8.1: Log densities of the first 500 Metropolis samples.

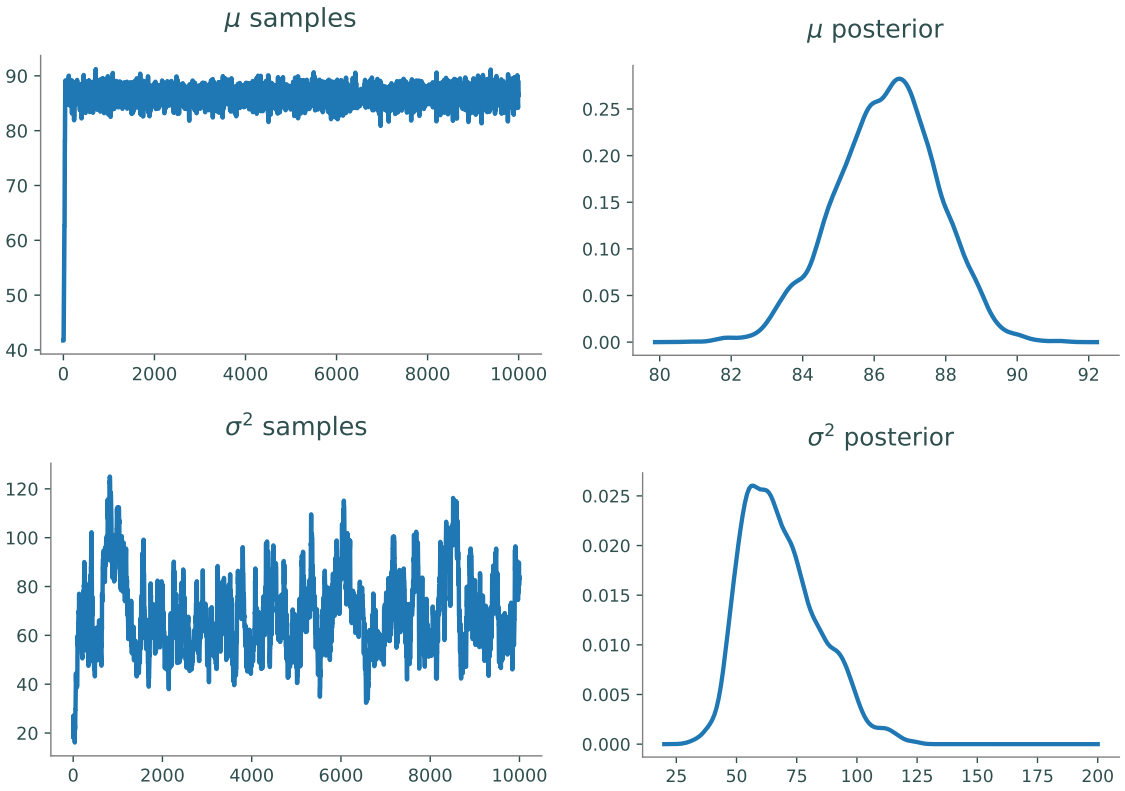


Figure 8.2: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

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 1 to complete the problem.

Your function should return an array of draws, an array of the log probabilities, and an acceptance rate. Create plots resembling Figures 8.1 and 8.2:

- Plot the log probabilities of the first 500 samples.
- Plot the samples for μ in the order they were drawn, and likewise for σ^2.
- Using `seaborn.kdeplot` plot the distribution of all samples for μ, and likewise for σ^2.

Use $u = 20$ for the parameter of the proposal function. Use the initial state $\mathbf{y}_0 = (\mu_0, \sigma_0^2) = (40, 10)$. Take 10,000 samples for both μ and σ^2.

Use the following code to load the data and initialize the priors:

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

```
# Prior sigma^2 ~ IG(alpha, beta)
>>> alpha = 3
>>> beta = 50
>>> muprior = stats.norm(loc=m, scale=sqrt(s**2))

#Prior mu ~ N(m, s)
>>> m = 80
>>> s = 4
>>> sig2prior = stats.invgamma(alpha, scale=beta)
```

The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice 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
```

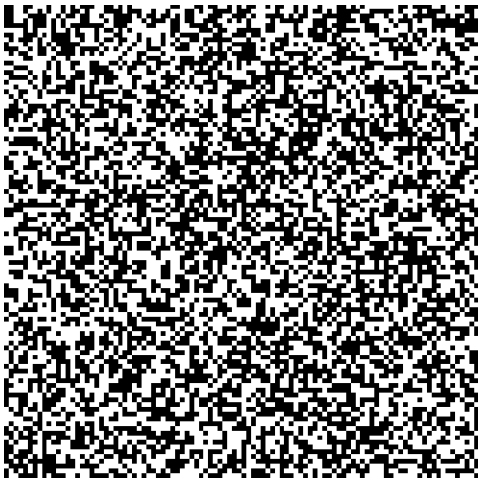


Figure 8.3: Spin configuration from random initialization.

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

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases} \tag{8.1}$$

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

$$H(\sigma^*) = H(\sigma) + 2 \sum_{j:j\sim i} \sigma_i \sigma_j. \tag{8.2}$$

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 (8.2) 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 (8.1), 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.

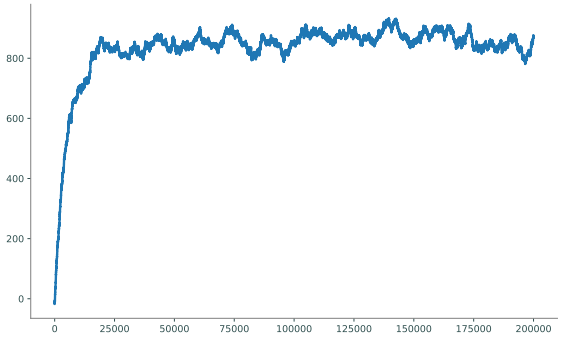
1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration σ^*.
2. Use Problem 4 to calculate the energy $H(\sigma^*)$ of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set $\sigma = \sigma^*$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

4. Track $-\beta H(\sigma)$ at each iteration (independent of acceptance).

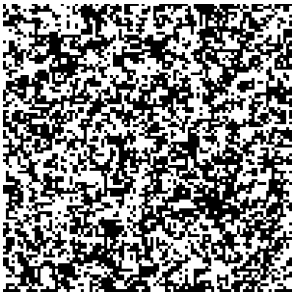
After the burn-in period, continue the iteration `n_samples` times, also recording every 100th sample (to prevent memory failure). The acceptance rate is counted after the burn-in period. Return the samples, the sequence of weighted energies $-\beta H(\sigma)$, and the acceptance rate.

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 8.4.

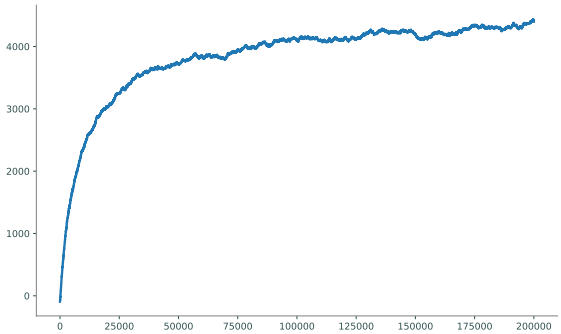
To show the spin configuration, use `plt.imshow(L,cmap='gray')`.



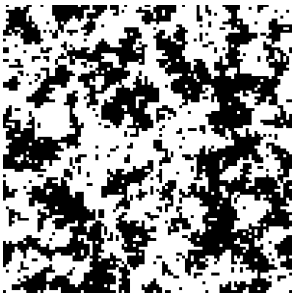
(a) Proportional log probs when $\beta = 0.2$.



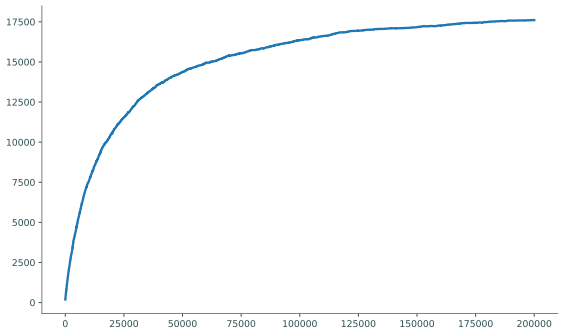
(b) Spin configuration sample when $\beta = 0.2$.



(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 8.4

9 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

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

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

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

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

Algorithm 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 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

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

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

Consider the dataset of N scores from a calculus exam in the file `examscores.npy`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean 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 $\mathbf{y} = (y_1, \ldots, y_N)$ be the set of exam scores, we would like to update our beliefs of μ and σ^2 by sampling from the posterior distribution

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

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

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

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

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

where

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

Note that μ^* and $(\sigma^*)^2$ are *not* samples and are not used to replace μ and σ^2 themselves; rather, they're parameters of the marginal distribution of μ (which happens to also be distributed normally) as shown above.

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling μ and sampling σ^2 (so using Algorithm 1, we would have $x_1 = \mu$ and $x_2 = \sigma^2$). We can sample from a normal distribution and an inverse gamma distribution as follows:

```
import numpy as np
from scipy.stats import norm
from scipy.stats import invgamma

mu = 0 # the mean
sigma2 = 9 # the variance
```

```
normal_sample = norm.rvs(mu, scale=np.sqrt(sigma))
alpha = 2
beta = 15
invgamma_sample = invgamma.rvs(alpha, scale=beta)
```

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

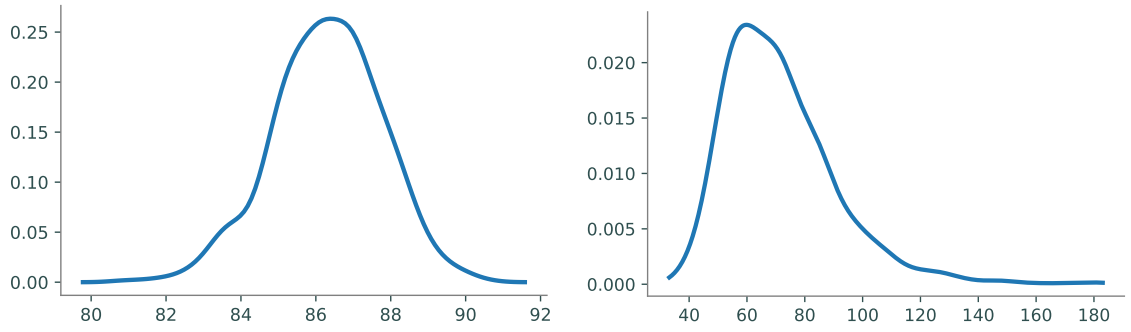
Problem 1. Write a function that accepts data $\mathbf{y}$, prior parameters ν, τ^2, α, and β, and an integer n. Use Gibbs sampling to generate n samples of μ and σ^2 for the exam scores problem.

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 versus the number of samples. They should both converge quickly, so that both plots look like "fuzzy caterpillars".

We'd like to look at the posterior marginal distributions for 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 9.1: 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*:

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

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.

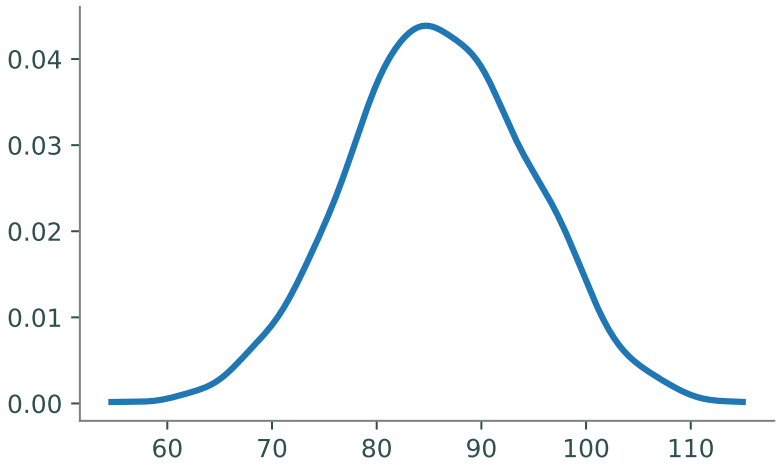


Figure 9.2: Predictive posterior distribution of exam scores.

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 9.1.

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 9.2.

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 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 v from the categorical distribution $\phi_{z_{m,n}}$. Throughout this implementation, we assume α and β are scalars[1]. In summary, we have

1. Draw $\phi_k \sim \text{Dir}(\beta)$ for $1 \leq k \leq K$.
2. For $1 \leq m \leq M$:
   (a) Draw $\theta_m \sim \text{Dir}(\alpha)$.
   (b) Draw $z_{m,n} \sim \text{Cat}(\theta_m)$ for $1 \leq n \leq N_m$.
   (c) Draw $v \sim \text{Cat}(\phi_{z_{m,n}})$ for $1 \leq n \leq N_m$.

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

This is typically depicted with graphical plate notation as in Figure 9.3.

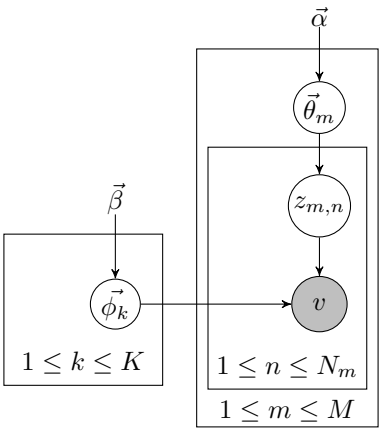


Figure 9.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables v are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each 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 $\mathbb{P}(\mathbf{z} \mid \mathbf{v}, \alpha, \beta)$, where $\mathbf{v}$ is the collection of words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$ (so as to condition on everything except the (m,n)-th entry), the conditional posterior distributions

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

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

[1]The Dirichlet distribution $\text{Dir}(x_1, ..., x_s, \alpha_1, ..., \alpha_s)$ usually requires the parameter α to be a vector of length s, but when α is a scalar, it is called the "concentration parameter" and behaves like a vector of length s whose entries are all equal to α.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k \mid \mathbf{z}_{-(m,n)}, \mathbf{v}, \alpha, \beta) \propto \frac{\left(n_{(k,m,\cdot)}^{-(m,n)} + \alpha\right)\left(n_{(k,\cdot,v)}^{-(m,n)} + \beta\right)}{n_{(k,\cdot,\cdot)}^{-(m,n)} + V\beta}$$

where

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

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

We have provided for you the structure of a Python object `LDACGS` with several methods, listed at the end of this lab. The object defines attributes `n_topics`, `alpha`, and `beta` upon initialization. The method `buildCorpus()` then defines attributes `vocab` and `documents`, where `vocab` is a list of strings (terms), and `documents` is a list of dictionaries (a dictionary for each document). For dictionary m in `documents`, each entry is of the form $n : v$, where v is the index in `vocab` of the n^{th} word in document m.

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

Problem 3. Complete the method `_initialize()` to initialize as attributes `n_words`, `n_docs`, the three count matrices, and the topic assignment dictionary `topics`.

To do this, you will need to initialize `nkm`, `nkv`, and `nk` to be zero arrays of the correct size. Matrix `nkm` corresponds to $n_{(k,m,\cdot)}$, `nkv` to $n_{(k,\cdot,v)}$, and `nk` to $n_{(k,\cdot,\cdot)}$. You will then iterate through each word found in each document. In the second of these for-loops (for each word), you will randomly assign `k` as an integer from the correct range of topics. Then, you will increment each of the count matrices by 1, given the values for `k`, `m`, and `v`, where `v` is the index in `vocab` of the n^{th} word in document m. Finally, assign `topics` as given.

The next method fully outlines a sweep of the Gibbs sampler.

Problem 4. Complete the method `_sweep()`.

To do this, iterate through each word of each document. The first part of this method will undo what `_initialize()` did by decrementing each of the count matrices by 1. Then, call the method `_conditional()` to use the conditional distribution (instead of the uniform distribution used previously) to pick a more accurate topic assignment `k`. Finally, repeat what `_initialize()` did by incrementing each of the count matrices by 1, but this time using the more accurate topic assignment.

You are now prepared to write the full Gibbs sampler.

Problem 5. Complete the method `sample()`. The argument `filename` is the name and location of a .txt file, which can be read in by the provided method `buildCorpus()` to build the corpus. Stopwords are removed if the `stopwords` argument is provided. Note that in `buildCorpus()`, **each line of `filename` is considered a document**.

Initialize attributes `total_nkm`, `total_nkv`, and `logprobs` as zero arrays. `total_nkm` and `total_nkv` will be the sums of every `sample_rate`th `nkm` and `nkv` matrix respectively. `logprobs` is of length `burnin` + `sample_rate` $*$ `n_samples` and will store each log-likelihood after each sweep of the sampler.

Burn-in the Gibbs sampler. After the burn-in, iterate further for `n_samples` iterations, adding `nkm` and `nkv` to `total_nkm` and `total_nkv` respectively, at every `sample_rate`th iteration. Also, compute and save the log-likelihood at each iteration in `logprobs` using the method `_loglikelihood()`.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on some of Ronald Reagan's State of the Union addresses, found in `reagan.txt`. Note that in `reagan.txt`, each line is an entire paragraph from one of Reagan's addresses, so your Gibbs sampler will consider each paragraph as a seperate document.

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 with the `reagan.txt` file. Use `stopwords.txt` as the stopwords file.

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

Make sure to save the `LDACGS` object, it will be used in the next problem.

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

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

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions 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 method `topterms()`, examine the topics for Reagan's addresses. If `n_topics=20` and `n_samples=10`, you should get the top 10 words that represent each of the 20 topics. Print out all 20 topics associated 10 words. For the top 5 topics, decide what their top 10 words jointly represent, and come up with a label for them.

We can use $\widehat{\theta}$ to find the documents (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 documents (paragraphs) are most centered on the Cold War. For your convenience, the provided method `toplines()` accomplishes just that by printing out the top `n_lines` documents corresponding to each topic.

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

Additional Material

LDACGS Source Code

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

    def __init__(self, n_topics, alpha=0.1, beta=0.1):
        """ Initializes attributes n_topics, alpha, and beta. """
        self.n_topics = n_topics
        self.alpha = alpha
        self.beta = beta

    def _buildCorpus(self, filename, stopwords_file=None):
        """ Reads the given filename, and using any provided stopwords,
            initializes attributes vocab and documents. In this lab,
            each line of filename is considered a document.

            vocab is a list of terms found in filename.

            documents is a list of dictionaries (a dictionary for each
            document); for dictionary m in documents, each entry is of
            the form n:v, where v is the index in vocab of the nth word
            in document m.
        """
        with open(filename, 'r') as infile:  # Create vocab
            doclines = [line.rstrip().lower().split(' ') for line in infile]
        n_docs = len(doclines)
        self.vocab = list({v for doc in doclines for v in doc})

        self.docs = doclines  # Save the documents for toplines()

        if stopwords_file:   # If there are stopwords, remove them from vocab
            with open(stopwords_file, 'r') as stopfile:
                stops = stopfile.read().split()
            self.vocab = [x for x in self.vocab if x not in stops]
            self.vocab.sort()

        self.documents = []  # Create documents
        for i in range(n_docs):
            self.documents.append({})
            for j in range(len(doclines[i])):
                if doclines[i][j] in self.vocab:
                    self.documents[i][j] = self.vocab.index(doclines[i][j])


    def _initialize(self):
        """ Initializes attributes n_words, n_docs, the three count matrices,
            and the topic assignment dictionary topics.
```

```
        Note that
        n_topics = K, the number of possible topics
        n_docs   = M, the number of documents being analyzed
        n_words  = V, the number of words in the vocabulary

        To do this, you will need to initialize nkm, nkv, and nk
        to be zero arrays of the correct size.
        Matrix nkm corresponds to n_(k,m,.)
        Matrix nkv corresponds to n_(k,.,v)
        Matrix nk corresponds to n_(k,.,.)
        You will then iterate through each word found in each document.
        In the second of these for-loops (for each word), you will
        randomly assign k as an integer from the correct range of topics.
        Then, you will increment each of the count matrices by 1,
        given the values for k, m, and v, where v is the index in
        vocab of the nth word in document m.
        Finally, assign topics as given.
    """
    self.n_words = len(self.vocab)
    self.n_docs = len(self.documents)

    # Initialize the three count matrices
    # The (k, m) entry of self.nkm is the number of words in document m ↩
        assigned to topic k
    self.nkm = np.zeros((self.n_topics, self.n_docs))
    # The (k, v) entry of self.nkv is the number of times term v is ↩
        assigned to topic k
    self.nkv = np.zeros((self.n_topics, self.n_words))
    # The (k)-th entry of self.nk is the number of times topic k is ↩
        assigned in the corpus
    self.nk = np.zeros(self.n_topics)

    # Initialize the topic assignment dictionary
    self.topics = {} # Key-value pairs of form (m,n):k

    random_distribution = np.ones(self.n_topics) / self.n_topics
    for m in range(self.n_docs):
        for n in self.documents[m]:
            # Get random topic assignment, i.e. k = ...
            # Increment count matrices
            # Store topic assignment, i.e. self.topics[(m,n)]=k
            raise NotImplementedError("Problem 3 Incomplete")


def _sweep(self):
    """ Iterates through each word of each document, giving a better
        topic assignment for each word.
```

```
            To do this, iterate through each word of each document.
            The first part of this method will undo what _initialize() did
            by decrementing each of the count matrices by 1.
            Then, call the method _conditional() to use the conditional
            distribution (instead of the uniform distribution used
            previously) to pick a more accurate topic assignment k.
            Finally, repeat what _initialize() did by incrementing each of
            the count matrices by 1, but this time using the more
            accurate topic assignment.
        """
        for m in range(self.n_docs):
            for n in self.documents[m]:
                # Retrieve vocab index for n-th word in document m
                # Retrieve topic assignment for n-th word in document m
                # Decrement count matrices
                # Get conditional distribution
                # Sample new topic assignment
                # Increment count matrices
                # Store new topic assignment
                raise NotImplementedError("Problem 4 Incomplete")


    def sample(self, filename, burnin=100, sample_rate=10, n_samples=10, ↩
        stopwords_file=None):
        """ Runs the Gibbs sampler on the given filename.

            The argument filename is the name and location of a .txt
            file, which can be read in by the provided method _buildCorpus()
            to build the corpus. Stopwords are removed if the stopwords
            argument is provided. Note that in buildCorpus(),
            each line of filename is considered a document.

            Initialize attributes total_nkm, total_nkv, and logprobs as
            zero arrays.
            total_nkm and total_nkv will be the sums of every
            sample_rate-th nkm and nkv matrix respectively.
            logprobs is of length burnin + sample_rate * n_samples
            and will store each log-likelihood after each sweep of
            the sampler.

            Burn-in the Gibbs sampler. After the burn-in, iterate further
            for n_samples iterations, adding nkm and nkv to total_nkm and
            total_nkv respectively, at every sample_rate-th iteration.
            Also, compute and save the log-likelihood at each iteration
            in logprobs using the method _loglikelihood().
        """
        self._buildCorpus(filename, stopwords_file)
        self._initialize()
```

```
        self.total_nkm = np.zeros((self.n_topics, self.n_docs))
        self.total_nkv = np.zeros((self.n_topics, self.n_words))
        self.logprobs = np.zeros(burnin + sample_rate * n_samples)

        for i in range(burnin):
            # Sweep and store log likelihood
            raise NotImplementedError("Problem 5 Incomplete")
        for i in range(sample_rate * n_samples):
            # Sweep and store log likelihood
            raise NotImplementedError("Problem 5 Incomplete")
            if not i % sample_rate:
                # Accumulate counts
                raise NotImplementedError("Problem 5 Incomplete")


    def _conditional(self, m, v):
        """ Returns the conditional distribution given m and w.
            Called by _sweep(). """
        dist = (self.nkm[:,m] + self.alpha) * (self.nkv[:,v] + self.beta) / (↩
            self.nk + self.beta * self.n_words)
        return dist / np.sum(dist)

    def _loglikelihood(self):
        """ Computes and returns the log-likelihood. Called by sample(). """
        lik = 0

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

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

        return lik

    def _phi(self):
        """ Initializes attribute phi. Called by topterms(). """
        phi = self.total_nkv + self.beta
        self.phi = phi / np.sum(phi, axis=1)[:,np.newaxis]

    def _theta(self):
        """ Initializes attribute theta. Called by toplines(). """
        theta = self.total_nkm + self.alpha
        self.theta = theta / np.sum(theta, axis=1)[:,np.newaxis]
```

```
def topterms(self, n_terms=10):
    """ Returns the top n_terms of each topic found. """
    self._phi()
    vec = np.atleast_2d(np.arange(0, self.n_words))
    topics = []
    for k in range(self.n_topics):
        probs = np.atleast_2d(self.phi[k,:])
        mat = np.append(probs, vec, 0)
        sind = np.array([mat[:,i] for i in np.argsort(mat[0])]).T
        topics.append([self.vocab[int(sind[1, self.n_words - 1 - i])]
                       for i in range(n_terms)])
    return topics

def toplines(self, n_lines=5):
    """ Print the top n_lines corresponding to each topic found. """
    self._theta()
    lines = np.zeros((self.n_topics,n_lines))
    for k in range(self.n_topics):
        args = np.argsort(self.theta[:,k]).tolist()
        args.reverse()
        lines[k,:] = np.array(args)[0:n_lines] + 1
    lines = lines.astype(int)

    for k in range(self.n_topics):
        print(f"TOPIC {k + 1}")
        for document in lines[k]:
            print(" ".join(self.docs[document]))
```

10 Gaussian Mixture Models

Lab Objective: *Understand the formulation of Gaussian Mixture Models (GMMs) and use the Expectation Maximization algorithm to estimate GMM parameters.*

Mixture models are a useful way to combine distributions together that allows us to describe much more complicated distributions than using just the standard list of named distributions. The essential idea of a mixture model is in its name: it is a mixture of several different models, or probability distributions. Each of these model is called a *component*. Each component has a certain probability associated with it, called its *weight*, that describes how likely it is for a sample from the model to come from that component. We denote the weight of the i-th component as w_i.

In this lab, we focus on *Gaussian Mixture Models*, or GMMs for short. In a GMM, each component is a multivariate Gaussian (normal) distribution. Each of these is parameterized by a mean μ_i and a covariance matrix Σ_i.

A GMM with K components thus has parameters $\theta = (w_1, \dots, w_K, \mu_1, \dots, \mu_K, \Sigma_1, \dots, \Sigma_K)$. We can use the law of total probability to evaluate the density of a GMM, which is given by

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

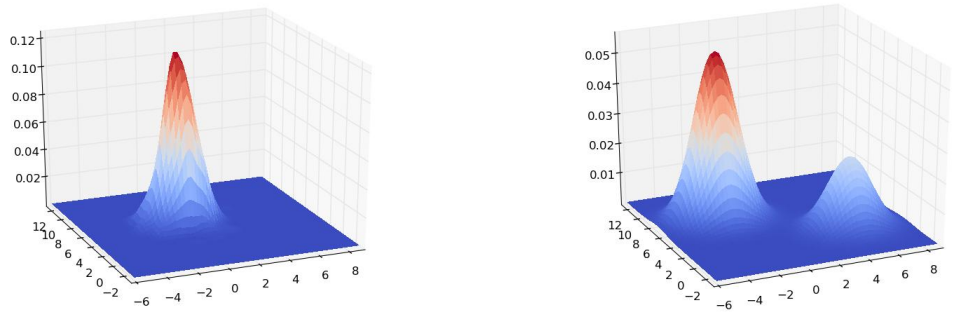
where

$$\mathcal{N}(z|\mu, \Sigma) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2}(z-\mu)^T \Sigma^{-1} (z-\mu)\right)$$

is the density function of a multivariate normal distribution.

It is important to keep in mind that a GMM does *not* arise from adding weighted multivariate normal random variables, but rather from weighting the responsibility of each multivariate normal random variable. The first case simply results in a different multivariate normal distribution. Refer to Figure 10.1 for a visualization of these two cases.

Problem 1. Throughout this lab, we will build a GMM class with various methods. Write the `__init__` method for this class. It should accept a parameter for the number of components and optional parameters for the weights, means, and covariance matrices which define the GMM, and store these.[a]



(a) Sum of weighted multivariate normal random variables.

(b) Weighted mixture of multivariate normal random variables.

Figure 10.1

If we have K components and d dimensions, then the weights should have shape `(K,)`, the means `(K,d)`, and the covariances `(K,d,d)`. The parameters for the k-th component can be found as `weights[k], means[k], covars[k]`.

[a] If we don't have a good guess for the parameters of the GMM to pass into the class, it makes more sense to initialize these from the dataset we are training on, which we will do later in the `fit` method; hence, we let the parameters be optional here.

Problem 2. Write a method `component_logpdf` for your class that accepts a component `k` and a point `z` and computes

$$\log w_k + \log \mathcal{N}(z|\mu_k, \Sigma_k),$$

the logarithm of the contribution of the k-th component of the pdf. Also write a method `pdf` that accepts a point `z` and returns the probability density of the whole GMM at that point.

Hint: `scipy.stats.multivariate_normal.pdf` and `scipy.stats.multivariate_normal.logpdf` can be used to efficiently evaluate the multivariate normal pdf.

We will use the following initialization to test the next several problems. The initialization code has been provided for you in the 'Check Section' as the function 'init_gmm':

```
gmm = GMM(n_components = 2,
        weights = np.array([0.6, 0.4]),
        means = np.array([[-0.5, -4.0], [0.5, 0.5]]),
        covars = np.array([
            [[1, 0],[0, 1]],
            [[0.25, -1],[-1, 8]],
        ]))
```

Your functions should give the following output:

```
>>> gmm.pdf(np.array([1.0, -3.5]))
0.05077912539363083
# Component 0
>>> gmm.component_logpdf(0, np.array([1.0, -3.5]))
-3.598702690175336
# Component 1
>>> gmm.component_logpdf(1, np.array([1.0, -3.5]))
-3.7541677982835004
```

Note that since this GMM is 2-dimensional, the input point must be an array of length 2.

In order to get credit for the next few problems, you must write tests in the 'Check Section' that show your functions give the proper outputs. The check for problem 2 has been written for you as an example.

In order to draw a value from a mixture model, we must first draw a variable $X \sim \text{Cat}(w_1, \ldots, w_K)$ that represents which component the sample comes from. We can then draw the sample $Z \sim \mathcal{N}(\mu_X, \Sigma_X)$. If we want to draw multiple samples, we need to repeat this process for each one (draw an X and then draw a Z.

Problem 3. Write a method `draw` for the GMM class that randomly draws from the model. If `m` points are drawn and the GMM is d-dimensional, the returned array should have shape `(m,d)`.

The function 'check_problem3()' is also provided. It will test your draw function by plotting your sample draw against your pdf function. If done correctly, running the check function should make 2 plots that look "good".

We now consider how to estimate the parameters of a GMM given some observed data $Z = z_1, \ldots, z_n$. Ordinarily, a good approach would be to try to directly maximize the log-likelihood

$$l(\theta) = \sum_{i=1}^{n} \log \sum_{j=1}^{K} w_j \mathcal{N}(z_i | \mu_j, \Sigma_j).$$

However, this expression is very difficult to deal with using standard optimization methods, particularly because of the sum inside of the logarithm. A good alternative in this case is the *expectation maximization* (EM) algorithm. This is an iterative algorithm, where each step consists of maximizing a function that is designed to approximate the log-likelihood while being much easier to maximize.

Each iteration consists of two steps, the E-step and the M-step. Suppose our estimated parameters at the t-th iteration are $\theta^t = (w_1^t, \ldots, w_K^t, \mu_1^t, \ldots, \mu_K^t, \Sigma_1^t, \ldots, \Sigma_K^t)$. Note that t is an index, not an exponent. For each data point $z_i, 1 \leq i \leq n$ and each component $1 \leq k \leq K$, the E-step consists of computing

$$\begin{aligned} q_i^t(k) &= P(X_i = k | z_i, \theta^t) \\ &= \frac{P(z_i | X_i = k, \theta^t)}{P(z_i | \theta^t)} \\ &= \frac{w_k^t \, \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^{K} w_{k'}^t \, \mathcal{N}(z_i | \mu_{k'}^t, \Sigma_{k'}^t)} \end{aligned}$$

In order to accurately compute this quantity, however, we need to be more careful. It is possible that due to floating point underflow[1] that each term $w_{k'}^t \mathcal{N}(z_i|\mu_{k'}, \Sigma_{k'})$ in the sum in the denominator becomes zero, which is a major problem. This particularly happens if the exponents in the multivariate normal densities all are large negative numbers. To avoid this problem, we can rescale the numerator and denominator. Let

$$\ell_{i,k} = \log w_k^t + \log \mathcal{N}(z_i|\mu_k^t, \Sigma_k^t),$$

the logarithm of each term in the denominator. For each data point z_i, we can find

$$L_i = \max_{k'} \ell_{i,k'},$$

the largest of these logarithms. Then, we can rewrite the quantity we want to calculate as

$$\begin{aligned} q_i^t(k) &= \frac{w_k^t \mathcal{N}(z_i|\mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^K w_{k'}^t \mathcal{N}(z_i|\mu_{k'}^t, \Sigma_{k'}^t)} \\ &= \frac{e^{\ell_{i,k}}}{\sum_{k'=1}^K e^{\ell_{i,k'}}} \\ &= \frac{e^{\ell_{i,k}} e^{-L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'}} e^{-L_i}} \\ &= \frac{e^{\ell_{i,k}-L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'}-L_i}}. \end{aligned}$$

This rescaling makes the largest term in the denominator equal to 1, so computing $q_i^t(k)$ in this way avoids underflow problems. Note that for the computation of any individual $q_i^t(k)$, the value L_i is a scalar that is the same for all components; however, you will have as many of these values as you have data points. As a reminder, i corresponds to the index of a data point and k corresponds to which componet we are comparing it to.

Problem 4. Write a method `_compute_e_step` that calculates the $q_i^t(k)$ as given by the E-step, given a collection of observations. Be sure to do the calculation in a way that avoids underflow, and use array broadcasting when possible.

Your method will accept an array of shape `(n,d)`, where `n` is the number of data points and `d` is the dimensionality of the data (i.e. each row is a data point). The array you produce should have shape `(n_components, n)` where `result[k,i]` $= q_i^t(k)$ (i.e. each row is one component, and each column is a data point). The various intermediate values should have shapes similar to the following:

- The array of $\ell_{i,k}$s should have shape `(n_components, n)`
- The array of L_is should have shape `(n,)`
- The array of the denominator values $\sum_{k'=1}^K e^{\ell_{i,k'}-L_i}$ should also have shape `(n,)`

With the GMM from the example in Problem 2, you should get the following results:

[1] As a refresher, one way that floating point numbers are limited is that they cannot represent positive numbers arbitrarily close to zero; at some point, if the number in a computation becomes too small, the computer is forced to round it to zero, which is called *underflow*. The threshhold is about 10^{-323} for the 64-bit floating point numbers used in python. Even if underflow does not occur, very small floating points have greatly reduced precision, so it is generally good to avoid using them.

```
>>> data = np.array([
        [0.5, 1.0],
        [1.0, 0.5],
        [-2.0, 0.7]
    ])
>>> gmm._compute_e_step(data)
array([[3.49810771e-06, 5.30334386e-05, 9.99997070e-01],
       [9.99996502e-01, 9.99946967e-01, 2.93011749e-06]])
```

Complete the check_problem4() function in the Check Section to show your GMM gets the correct result.

Now that we have the $q_i^t(k)$, we can perform the M-step. This step consists of maximizing the function

$$Q^t(\theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} q_i^t(k) \log w_k^t \mathcal{N}(z_i|\mu_k, \Sigma_k)$$

We then set

$$\theta^{t+1} = \underset{\theta}{\text{argmax}} Q^t(\theta)$$

and iterate until the method appears to converge. In the case of GMMs, the maximizer θ^{t+1} of $Q^t(\theta)$ is given by

$$w_k^{t+1} = \frac{1}{n} \sum_{i=1}^{n} q_i^t(k)$$

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

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

For details on the derivation of the maximizer, refer to the Volume 3 textbook.

Problem 5. Write a method `_compute_m_step` for your GMM class that takes the output of Problem 4 as the $q_i^t(k)$ values and calculates the next iteration of weights and means using the formulas above. Be sure to use array broadcasting when possible. Coding the formula for the sigma update requires a rather involved approach using devious array broadcasting and a very useful function, np.einsum.

```
# This gets the centered observations for the numerator
# by subtracting them as per the formula
obs_centered = np.expand_dims(Z,0) - np.expand_dims(new_means, 1)
# This function creates the rest of the numerator and uses array
# broadcasting to implement the denominator. The K, n, d, and D tell
# np.einsum which axes to multiply and which to sum
new_covars =
```

```
    np.einsum('Kn,Knd,KnD->KdD', q_values, obs_centered, obs_centered) / ↩
        q_sum.reshape(-1,1,1)
```

The code above has been provided in the spec file, but a general understanding of the method will be useful in a variety of other situations. Return the updated parameters (weights, means, and covariance matrices).

With the same GMM and data as in Problem 4, you should get the following results:

```
>>> gmm._compute_m_step(data)
(array([0.3333512, 0.6666488]),
 array([[-1.99983216,  0.69999044],
        [ 0.74998978,  0.75000612]]),
 array([[[ 4.99109197e-04, -2.91933135e-05],
         [-2.91933135e-05,  2.43594533e-06]],

        [[ 6.25109881e-02, -6.24997069e-02],
         [-6.24997069e-02,  6.24999121e-02]]]))
```

finish writing check_problem5() to show you have the correct outputs

Problem 6. Write a `fit` method for your GMM class.

First, if the GMM's parameters are uninitialized (set to `None`), initialize the parameters of the components. We want to do this in a way that the algorithm starts with reasonable values for the dataset. A good way to initialize the means is to randomly select points from the dataset. The covariance matrices can be initialized as diagonal matrices based on the variance of the data. Ensure that the weights you choose add up to 1.

Then, perform the expectation maximization algorithm. Use the functions you created in Problems 4 and 5 to calculate the parameters at each step. Repeat until the parameters converge. Use the following to measure the change in the parameters with each iteration:

```
change = (np.max(np.abs(new_weights - old_weights))
        + np.max(np.abs(new_means - old_means))
        + np.max(np.abs(new_covars - old_covars)))
```

The file `gmm_data.npy` contains a collection of data drawn from a two-dimensional GMM. Finish check_problem6() by creating a variable named gmm that is a GMM object initialized with 3 componnts and fitted to the data from `gmm_data.npy`.

The check function will plot the pdf of your trained gmm and a hexbin plot of the training data. They should look very similar to each other. Additionally, your class should take less than 15 seconds to train on the dataset. The check code will time your fit function and print the training time to make sure you are within that range.

Clustering with GMMs

An important use of mixture models is for *clustering*. The objective of clustering is to take an unlabeled dataset and separate it into some number of clusters, which can then be labeled. This is an instance of *unsupervised learning*, as it is a machine learning task where the the training algorithm does not need the true answers (in this case, the actual clusters).

In order to cluster a dataset using a GMM, we first need to train the GMM on that data. Then, we can assign each point a label by finding which component has the largest contribution to the pdf there. Written symbolically, for a data point z, we have

$$\text{Cluster}(z) = \text{argmax}_k w_k \mathcal{N}(z|\mu_k, \Sigma_k).$$

Note that the number of clusters (components) is a hyperparameter that must be selected before a GMM is trained. In general, cross-validation or some other method must be used to find the right number of clusters.

Problem 7. Write a `predict` method for your class. Given a set of data points, return which cluster has the highest pdf density for each data point.

The file `classification.npz` contains a set of 3-dimensional data points (`X`) and their labels (`y`). Use your class with `n_components=4` to cluster the data. Plot the points with the predicted and actual labels, and compute and return your model's accuracy. Your class should take less than 30 seconds to train on this dataset. Make sure to time your .fit() or .fit_predict() and print the time spent in training to receive credit.

Note that the labels may be permuted; for instance, your model might cluster the points correctly, but swap the labels of clusters 1 and 2 compared to the true labels. The model would still be considered accurate in this case; we only care what the clusters are, not how the model labels them. To resolve this problem, we need to find the permutation of the labels that results in the highest accuracy. The following function does this in a way that is more efficient than directly checking all permutations:

```
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import confusion_matrix

def get_accuracy(pred_y, true_y):
    """
    Helper function to calculate the actually clustering accuracy,
    accounting for the possibility that labels are permuted.
    """
    # Compute confusion matrix
    cm = confusion_matrix(pred_y, true_y)
    # Find the arrangement that maximizes the score
    r_ind, c_ind = linear_sum_assignment(cm, maximize=True)
    return np.sum(cm[r_ind, c_ind]) / np.sum(cm)
```

For convenience, a method `fit_predict` for the class is also included in the specifications file that calls both `fit` and `predict` to make the clustering process simpler.

Clustering with GMMs is closely related to the K-means algorithm. In fact, K-means can be viewed as a special case of GMMs where the covariance matrices are all the identity. How might this affect its ability to cluster? We now compare the effectiveness of GMMs for classification on this dataset with K-means, as well as comparing to sklearn's implementation.

Problem 8. The function method_comparison Initializes an instance of your GMM as well as a GMM and K-means object from Sklearn and compares the 3 based on accuracy and time to train. Run the function and observe the results, then answer in the markdown cell below why K-Means might have performed worse than the GMM's.

You may also find it interesting that sklearn's GMM is actually faster on this dataset than K-means despite GMM's being more complicated to train. This is in part because the dataset is rather low-dimensional. As the dimension of the dataset grows, GMMs suffer computationally from the curse of dimensionality much more than the K-means algorithm.

Additional Materials

Jax

Jax is a combination of both Autograd and XLA (Accelerated Linear Algebra) to provide high performance computations. It is a tool that can automatically differentiate various Python and NumPy code including `if` statement, `for` loops, recursion, and other native code types.

Jax provides a Numpy-like API to build machine learning models. Jax can only run on GPUs and TPUs which makes it more efficient than NumPy, which can only run on a CPU. The three main Jax functions include `jit`, `grad` and `vmap`.

- jit: Adding `@jax.jit` to the beginning of a function creates an optimized version of the function.
- grad: Used to compute the gradient or derivative of a function.
- vmap: This is used to vectorize your functions. Since list comprehension isn't available using Jax's version of NumPy, this can be helpful and used instead.

Dynamax

Dynamax is a library that uses Jax for probabilistic state space models (SSMs). The SSMs that Dynamax is able to compute include Hidden Markov Models (HMMs), Linear and Nonlinear Gaussian state space models, and Generalized Gaussian state space models. More information can be found at the website: `https://probml.github.io/dynamax/`.

11 Discrete Hidden Markov Models

Lab Objective: *Understand how to use discrete Hidden Markov Models.*

A common probabilistic model is the *hidden Markov model* (HMM). In an HMM, we have two sequences of random variables, $(X_t)_{t=0}^{\infty}$ and $(Z_t)_{t=0}^{\infty}$. The X_t are called the *state sequence* or *hidden state*, and the Z_t are known as the *observation sequence*. We assume that the X_t form a Markov chain, i.e. the distribution of X_t is entirely determined by the value of X_{t-1}, and that the distribution of Z_t is determined by the value of X_t. We also typically assume that we only know the values of the Z_t, not the X_t (hence the name). We denote the state space as $\mathscr{X}$ and the observation space as $\mathscr{Z}$, so that for all t we have $X_t \in \mathscr{X}$ and $Z_t \in \mathscr{Z}$. Hidden Markov models are useful in many situations where we have indirect observations of a sequential or time-based process, including speech and handwriting prediction, text analysis, gene prediction, and many other areas.

In this lab, we explore HMMs with discrete state and observation spaces. Assume the state space $\mathscr{X}$ and observation space $\mathscr{Z}$ are finite sets where $|\mathscr{X}| = n$ and $|\mathscr{Z}| = m$. For simplicity we relabel these sets as $\mathscr{X} = \{0, 1, 2, \dots, n-1\}$ and $\mathscr{Z} = \{0, 1, 2, \dots, m-1\}$ We will also assume that the HMM is temporally homogeneous, i.e. that the transition probabilities do not change with t.

In this case, we can parameterize all such HMMs by $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$ where $\boldsymbol{\pi} \in \mathbb{R}^n$ represents the distribution of X_0 (the initial state distribution), A is a $n \times n$ column-stochastic matrix describing how X_t is affected by X_{t-1} (the state transition matrix), and B is a $m \times n$ column-stochastic matrix describing how Z_t is affected by X_t (the state observation matrix). The entries of $\boldsymbol{\pi}$, A, and B specifically are the following:

$$\begin{aligned} \boldsymbol{\pi}_i &= P(X_0 = i) \\ a_{ij} &= P(X_t = i | X_{t-1} = j) \\ b_{ij} &= P(Z_t = i | X_t = j) \end{aligned}$$

Finally, we let $\boldsymbol{z} = [z_0, z_1, \dots z_{T-1}]$ be a vector of observations, where each z_t is a draw from Z_t.

Given one or both of $\boldsymbol{\theta}$ and $\mathbf{z}$, there are several questions we might want to answer:

1. What is the likelihood that our model generated the observation sequence? In other words, what is $P(\boldsymbol{z}|\boldsymbol{\theta})$?

2. Given $\boldsymbol{z}$, $\boldsymbol{\theta}$, and an integer $0 \leq k \leq T-1$, what is the most likely value for the state X_k at time k?

3. Given $\boldsymbol{z}$ and $\boldsymbol{\theta}$, what is the most likely state sequence $\boldsymbol{x}$ to have generated $\boldsymbol{z}$?

4. How can we choose the parameters $\boldsymbol{\theta}$ that maximize $P(\boldsymbol{z}|\boldsymbol{\theta})$?

The first of these is answered by the *forward pass* algorithm; the second by the *backwards pass* algorithm; the third by the *Viterbi algorithm*; and the fourth is typically approached using the *Baum-Welch algorithm*, which is the special case of expectation maximization applied to an HMM. Throughout this lab, we will use all four of these algorithms.

Problem 1. Create a class called `HMM`. Create the constructor, which accepts arguments `pi`, `A`, and `B`. Save each of these as an attribute with the same name.

The Forward Pass

The goal of the forward pass algorithm is to efficiently compute $P(\boldsymbol{z}|\boldsymbol{\theta})$. Directly expanding this out as a sum over all state sequences requires a number of computations that grows exponentially with the length of the observation sequence, and is completely impractical. Instead, the forward pass algorithm splits this probability into separate values that can be easily computed recursively.

As the first step of the algorithm, consider the values

$$\alpha_t(i) = P(z_0, \dots, z_t, x_t = i|\boldsymbol{\theta}).$$

The law of total probability gives us that

$$P(\boldsymbol{z}|\boldsymbol{\theta}) = \sum_{i \in \mathscr{X}} \alpha_{T-1}(i),$$

so finding the $\alpha_t(i)$ lets us find $P(\boldsymbol{z}|\boldsymbol{\theta})$. It can be shown that[1]

$$\alpha_0(i) = \pi_i b_{z_0,i}$$
$$\alpha_t(i) = b_{z_t,i} \sum_{j \in \mathscr{X}} \alpha_{t-1}(j) a_{ij}$$

which allows us to efficiently compute the $\alpha_t(i)$ iteratively. When we implement this algorithm, we will store the values of $\alpha_t(i)$ in a single 2D array. The (t, i)-th entry of this array will be the value $\alpha_t(i)$.

Problem 2. Create a method `forward_pass` in your HMM class to implement the forward pass algorithm. This function should accept the observation sequence `z` (with shape `(T,)`) and return the array of $\alpha_t(i)$ values (with shape `(T,n)`).

To test your code, use the following example HMM:

```
>>> pi = np.array([.6, .4])
>>> A = np.array([[.7, .4],[.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> z_example = np.array([0, 1, 0, 2])
>>> example_hmm = HMM(pi, A, B)
```

You should get the following output using the example HMM:

[1] For verification of the mathematics behind this and the other algorithms in this lab, refer to the Volume 3 textbook.

```
>>> alpha = example_hmm.forward_pass(z_example)
>>> print(np.sum(alpha[-1,:]) # the probability of the observation
0.009629599999
```

Problem 3. Consider the following (very simplified) model of the price of a stock over time as an HMM. The observation states will be the change in the value of the stock. For simplicity, we will group these into five values: large decrease, small decrease, no change, small increase, large increase, labeled as integers from 0 to 4. The hidden state will be the overall trends of the market. We'll consider the market to have three possible states: declining in value (bear market), not changing in value (stagnant), and increasing in value (bull market), labeled as integers from 0 to 2. Let the HMM modeling this scenario have parameters

$$\boldsymbol{\pi} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \quad A = \begin{bmatrix} 0.5 & 0.3 & 0 \\ 0.5 & 0.3 & 0.3 \\ 0 & 0.4 & 0.7 \end{bmatrix}, \quad B = \begin{bmatrix} 0.3 & 0.1 & 0 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.3 \\ 0.1 & 0.2 & 0.4 \\ 0 & 0.1 & 0.2 \end{bmatrix}$$

The file `stocks.npy` contains a sequence of 50 observations drawn from this HMM. What is the probability of this observation sequence given these model parameters? Use your implementation of the forward pass algorithm from Problem 2 to find the answer. Note that the answer is very small, because there are lots of possible observation sequences.

The Backward Pass

The backward pass algorithm seeks to answer the second question: given an observation sequence, parameters for an HMM, and a specific timestep, what is the most likely state at that step? As with the first question, trying to directly compute the answer via expanding into a sum over individual terms is completely impractical. The backwards pass algorithm takes a different approach.

Define the function

$$\gamma_t(i) = P(X_t = i | \mathbf{z}, \boldsymbol{\theta}).$$

The answer to the second question is then given by $\text{argmax}_{i \in \mathscr{X}} \gamma_t(i)$ for a fixed timestep t. In order to compute the $\gamma_t(i)$ efficiently, the backwards pass breaks the problem up in a clever way to allow an efficient iterative solution. Consider the values

$$\beta_t(j) = P(z_{t+1}, z_{t+2}, \ldots, z_{T-1} | X_t = j, \boldsymbol{\theta}).$$

where $\beta_{T-1}(i) = 1$. It can be shown that

$$\beta_t(j) = \sum_{i \in \mathscr{X}} a_{ij} \beta_{t+1}(i) b_{z_{t+1}, i}$$

and that

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathbf{z}|\boldsymbol{\theta})},$$

allowing efficient computation of these values. The backwards pass algorithm simply consists of iteratively computing the $\beta_t(i)$ values and using those to compute $\gamma_t(i)$. Note that the backwards pass algorithm requires running the forward pass algorithm first, and iterates over t in the opposite order.

Problem 4. Create a method `backward_pass` in your HMM class to implement the backward pass algorithm. This function should accept the observation sequence `z` (with shape `(T,)`) and return two arrays of the $\beta_t(i)$ and $\gamma_t(i)$ values (each with shape `(T,n)`).

To test your function, your code should produce the following output on the example HMM:

```
>>> beta, gamma = example_hmm.backward_pass(z_example, alpha)
>>> print(beta)
[[0.0302  0.02792]
 [0.0812  0.1244 ]
 [0.38    0.26   ]
 [1.      1.     ]]
>>> print(gamma)
[[0.18816981 0.81183019]
 [0.51943175 0.48056825]
 [0.22887763 0.77112237]
 [0.8039794  0.1960206 ]]
```

With your function and the stock model from Problem 3, answer the following question: given the observation sequence in `stocks.npy`, what is the most likely initial hidden state X_0?

Most Likely Sequence with the Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm that seeks to find the most likely sequence of hidden states $\mathbf{x} = (x_0, \dots, x_{T-1})$ that satisfies

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x}} P(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}).$$

The algorithm proceeds by considering the values

$$\eta_t(i) = \max_{x_0,\dots,x_{t-1}} P(x_0, \dots, x_{t-1}, X_t = i, z_0, \dots, z_t|\boldsymbol{\theta})$$

The Bellman optimality principal can be used to show that

$$\eta_0(i) = b_{z_0,i}\boldsymbol{\pi}_i,$$
$$\eta_t(i) = \max_{j\in\mathscr{X}} b_{z_t,i}a_{ij}\eta_{t-1}(j),$$

allowing us to compute these efficiently. The $\eta_t(i)$ give the maximizing probabilities at each timestep assuming we are in a certain hidden state. To extract the most likely sequence from the $\eta_t(i)$s, we iterate backwards as follows:

$$x^*_{T-1} = \operatorname*{argmax}_{j\in\mathscr{X}} \eta_{T-1}(j)$$
$$x^*_{t-1} = \operatorname*{argmax}_{j\in\mathscr{X}} b_{z_t,x^*_t}a_{x^*_t,j}\eta_{t-1}(j) = \operatorname*{argmax}_{j\in\mathscr{X}} a_{x^*_t,j}\eta_{t-1}(j).$$

Problem 5. Creating a method `viterbi_algorithm` in your HMM class to implement the Viterbi algorithm. This function should accept the observation sequence $\mathbf{z}$ (with shape `(T,)`) and return the most likely state sequence $\mathbf{x}^*$ (as an array with shape `(T,)`).

To test your function, it should output the following on the example HMM:

```
>>> xstar = example_hmm.viterbi_algorithm(z_example)
>>> print(xstar)
[1 1 1 0]
```

Apply your function to the stock market HMM from Problem 3. With the observaition sequence from `stocks.npy`, what is the most likely sequence of hidden states? Is the initial state of the most likely sequence the same as the most likely initial state you found in Problem 4?

Text Analysis with HMMs

We now turn to an interesting application of the Baum-Welch algorithm to train HMMs. We have coded most of the pieces needed for the Baum-Welch algorithm already in this lab. However, many of these require the computations to be done in a more careful way than we have presented here in order to prevent underflow. Instead of delving into the details of that, we will use the HMM implementation provided by the `hmmlearn` package, specifically the `hmmlearn.hmm.CategoricalHMM` class.

This class uses slightly different conventions and syntax from the HMM class that we have coded in this lab, so we will illustrate how to use this class on the data from `stocks.npy`.

```
import numpy as np
from hmmlearn import hmm
z = np.load('stocks.npy')
```

In the initializer, we specify the number of hidden states as the `n_components` argument (we will use 3):

```
h = hmm.CategoricalHMM(n_components=3)
```

To train the HMM, call the `fit` method. This method accepts the observation sequence `z`. However, it expects it to be an array with shape `(T,1)`, so reshape it as follows when you pass it in:

```
h.fit(z.reshape(-1,1))
```

Now the HMM is trained. To see the trained HMM parameters, use the following attributes: $\boldsymbol{\pi}$ is stored as `h.startprob_`, A is stored as `h.transmat_`, and B is stored as `h.emissionprob_`. However, A and B are transposed from the convention we are using, so extracting the parameters looks like the following:

```
pi = h.startprob_
A = h.transmat_.T
B = h.emissionprob_.T
```

The particular application we will use this for is to take some text and treat it as the observation sequence $\mathbf{z}$ of an HMM. Using the Baum-Welch algorithm to train an HMM with this setup can reveal interesting information about the underlying text. We will specifically use the sequence of characters (after stripping out punctuation and converting everything to lower-case) as our observation sequence.

In order to convert the raw text into data we can use with the `hmmlearn` package, we need to read and process the text and then map the characters to integer values. The following code accomplishes this task:

```
import numpy as np
import string
import codecs

def vec_translate(a, my_dict):
    # translate numpy array from symbols to state numbers or vice versa
    return np.vectorize(my_dict.__getitem__)(a)

def prep_data(filename):
    """
    Reads in the file and prepares it for use in an HMM.
    Returns:
        symbols (dict): a dictionary that maps characters to their integer values
        obs_sequence (ndarray): an array of integers representing the read-in text
    """
    # Get the data as a single string
    with codecs.open(filename, encoding='utf-8') as f:
        data=f.read().lower()  # and convert to all lower case
    # remove punctuation and newlines
    remove_punct_map = {ord(char):
                        None for char in string.punctuation+"\n\r"}
    data = data.translate(remove_punct_map)
    # make a list of the symbols in the data
    symbols = sorted(list(set(data)))
    # convert the data to a NumPy array of symbols
    a = np.array(list(data))
    # make a conversion dictionary from symbols to state numbers
    symbols_to_obsstates = {x:i for i,x in enumerate(symbols)}
    # convert the symbols in a to state numbers
    obs_sequence = vec_translate(a,symbols_to_obsstates)
    return symbols, obs_sequence.reshape(-1,1)
```

Problem 6. The file `declaration.txt` contains the text of the Declaraction of Independence.

Train an `hmmlearn.hmm.MultinomialHMM` on this data with $N = 2$ states and $M =$ `len(set(obs))` $= 27$ observation values (26 lower case characters and 1 whitespace character). Train the HMM with `n_iter=200` and `tol=1e-4` (note that both of these are arguments to the constructor, not to the `fit` function).

Once the learning algorithm converges, analyze the state observation matrix B. Note which rows correspond to the largest and smallest probability values in each column of B, and check the corresponding characters. The code below displays typical results for a well-converged HMM. (Note that the `u` before the `"` indicates that the string should be unicode, which will be useful in the next problem.)

```
>>> B = h.emissionprob_.T
>>> for i in range(len(B)):
>>>     print(u"{}, {:0.4f}, {:0.4f}"
                .format(symbols[i], *B[i,:]))
 , 0.0051, 0.3324
a, 0.0000, 0.1247
c, 0.0460, 0.0000
b, 0.0237, 0.0000
e, 0.0000, 0.2245
d, 0.0630, 0.0000
g, 0.0325, 0.0000
f, 0.0450, 0.0000
i, 0.0000, 0.1174
# ...
```

What do you notice about the columns of B? Write your observations in a markdown cell. If there is nothing apparent in your output, try re-running your HMM. Note that the order of the columns is completely arbitrary, and your code may switch the role of the two columns.

Problem 7. The file `WarAndPeace.txt` contains a portion of the Russian text of *War and Peace* by Tolstoy. Train an HMM on the text in this file with $N = 2$ states as in Problem 6. Interpret/explain your results. Which Cyrillic characters appear to be vowels?

12 Speech Recognition using CDHMMs

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

Continuous Density Hidden Markov Models

Some of the most powerful applications of hidden Markov models, speech and voice recognition, result from allowing the observation space to be continuous instead of discrete. These are called *continuous density hidden Markov models* (CDHMMs). The two most common formulations are *Gaussian HMMs* and *Gaussian mixture model HMMs* (GMMHMMs). In this lab, we will focus on GMMHMMs.

A GMMHMM is an HMM where the observation sequence variables $(Z_t)_{t=0}^{\infty}$ are distributed according to a Gaussian mixture model. To review, a Gaussian mixture model is a continuous multivariate distribution composed of K Gaussian distributions $\mathcal{N}(\mu_k, \Sigma_k)$ (where $\mu_k \in \mathbb{R}^M$ and $\Sigma_k \in \mathbb{R}^{M\times M}$) with corresponding weights c_k, for $1 \leq k \leq K$. A useful way to think of an individual GMM Z is to think of it as a pair of random variables Y and Z. The variable Y is a categorical variable on $\{1, 2, \dots, K\}$ with probabilities given by the c_k (i.e. $P(Y = k) = c_k$) and determines which Gaussian Z is drawn from. We then draw $Z \sim \mathcal{N}(\mu_Y, \Sigma_Y)$. The density function of a GMM is given as

$$f(\mathbf{z}) = \sum_{k=1}^{K} c_k \mathcal{N}(\mathbf{z};\, \mu_k, \Sigma_k).$$

For an example of what this looks like, refer to Figure 12.1. Note that this is a weighted sum of the density functions of Gaussian variables; remember that this is different from a sum of independent Gaussian random variables, which is just another Gaussian random variable!

GMMHMMs are then formulated similar to the discrete HMMs we have encountered before. We will assume that the hidden states X_t have N possible values. Then, the hidden state of the GMMHMM is parameterized by an initial state vector $\boldsymbol{\pi} \in \mathbb{R}^N$ and a state transition matrix $A \in \mathbb{R}^{N\times N}$ (both of these are the same as in the discrete case). The observation state is parameterized by one GMM per hidden state. We denote these as follows: for each hidden state $1 \leq i \leq n$, the observation state is distributed according to the GMM with component weights $\{c_{i,1}, \dots, c_{i,K}\}$, component means $\{\mu_{i,1}, \dots, \mu_{i,K}\}$, and component covariance matrices $\{\Sigma_{i,1}, \dots, \Sigma_{i,K}\}$.

To sample from a GMMHMM, follow the following process for each time step:

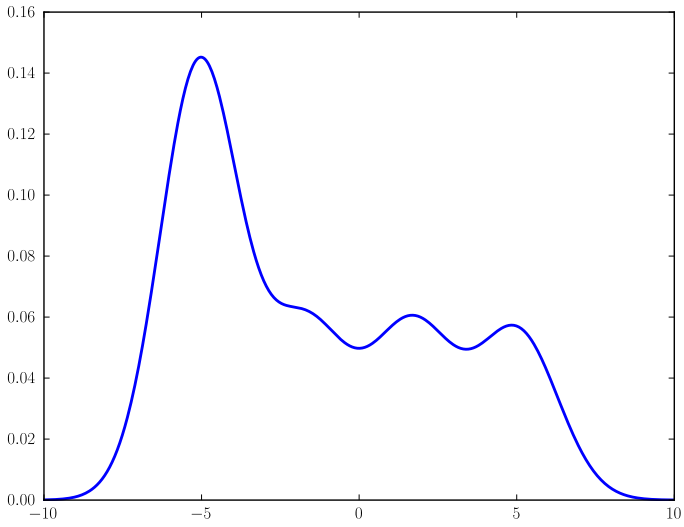


Figure 12.1: The probability density function of a mixture of Gaussians with four components.

- Determine the hidden state X_t (for X_0 this is with $\boldsymbol{\pi}$, and afterwards is determined by A and X_{t-1}).
- Determine the GMM component Y_t by drawing from a categorical variable with probabilities $c_{X_t,1}, \ldots, c_{X_t,K}$ (so $P(Y_t = j) = c_{X_t,j}$).
- Sample Z_t from the GMM by drawing from the normal distribution $\mathcal{N}(\mu_{X_t,Y_t}, \Sigma_{X_t,Y_t})$.

For an example of a sequence drawn from a GMMHMM, refer to Figure 12.2, which shows an observation sequence generated from a GMMHMM with two mixture component and two hidden states.

Problem 1. Consider the following GMMHMM with $N = 3$ states, components of dimension $M = 4$, and $K = 5$ components:

```
# NxN transition matrix
A = np.array([[.3, .3, .4], [.2, .3, .5], [.3, .2, .5]])
# NxK collection of component weights
weights = np.array([[.3, .2, .1, .2, .2], [.1, .3, .3, .2, .1],
                    [.1, .3, .2, .1, .3]])
# NxKxM collection of component means
means = np.array([np.floor(np.random.uniform(-100, 100, size = (5, 4)))
                        for i in range(3)])
# NxKx(MxM) collection of component covariance matrices
covars = np.array([[np.floor(np.random.uniform(1, 20))*np.eye(4)
                        for i in range(5)] for j in range(3)])
# (N,) ndarray initial state distribution
pi = np.array([.15, .15, .7])
```

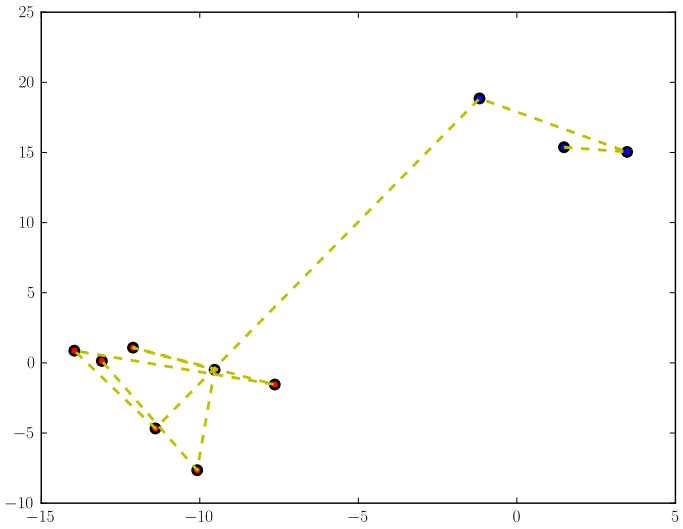


Figure 12.2: An observation sequence generated from a GMMHMM with two mixture components and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

The weight $c_{i,k}$ is `weights[i,k]`, the mean $\mu_{i,k}$ is `means[i,k,:]`, and the covariance matrix $\Sigma_{i,k}$ is `covars[i,k,:,:]`.

Write a function `sample_gmmhmm` which accepts an integer T, and draws T samples from the above GMMHMM.

Use your function to draw $T = 900$ samples. Use `sklearn.decomposition.PCA` with 2 components to plot the observations in two-dimensional space. Color the observations by state. How many distinct clusters do you see?

Hint: the function `np.random.choice` will be useful for drawing the hidden states and the GMM components, and `np.random.multivariate_normal` for the observation sequence. When plotting the samples, using the keyword argument `c` in `plt.scatter` allows you to specify the colors of the individual points.

Speech Recognition and Hidden Markov Models

The questions we asked about discrete HMMs can also be asked about CDHMMs, and the algorithms that answer these questions are virtually identical to the discrete case. However, with continuous observations it is much more difficult to implement the algorithms in a numerically stable way. We will not have you implement any of the algorithms for CDHMMs yourself; instead, we will use the `hmmlearn` package.

Hidden Markov Models are the basis of modern speech recognition systems. The essential idea is that we will use the sound data as the observation sequence. However, there are a lot of details that we won't address in this lab; a fair amount of signal processing must precede the HMM stage, and more sophisticated approaches require the use of language models.

To transform the audio data into data for our HMMs, we will use the following process:

- Divide the audio into frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these intervals.
- Transform the audio signal into mel-frequency cepstral coefficients (MFCCs). This transformation is similar to a Fourier transform in that it breaks the sound signal into frequencies.
- Keep the first $M = 10$ of these coefficients for each time frame. These will be our observation sequence in $\mathbb{R}^M$.

Problem 2. In the remainder of this lab, we will create a speech recognition system for the vocabulary of the following five words/phrases: "biology", "mathematics", "political science", "psychology", and "statistics".

The `Samples` folder contains 30 recordings for each of the words/phrases in the vocabulary. These audio samples are 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. For each of the words, create a list holding the MFCC coefficients of the recordings of that word.

The function `scipy.io.wavfile.read` can be used to load the sound files, and the function `extract` in `MFCC.py` implements the MFCC coefficient algorithm:

```
from scipy.io import wavfile
import MFCC

samplerate, sound_data = wavfile.read(filename)
coefficients = MFCC.extract(sound_data)
```

Hint: it might be helpful to use a dictionary to store the lists of coefficients.

Industrial-grade speech recognition systems train a GMMHMM for each individual English *phoneme*, or distinct sound. This allows for a lot of versatility, but also comes with its own set of challenges: it requires a lot of training data, with the audio files split into individual phonemes, as well as requiring the true words to be written in terms of their phonemes. The setup also becomes much more complicated if we try to account for multiple dialects of English.

Instead, in this lab, we will train a GMMHMM for each of the five individual words in the vocabulary.[1] Recall, however, that the training procedure can get stuck in a poor local minimum. To combat this, we will train 10 GMMHMMs for each word (using a different random initialization of the parameters each time) and keep the model with the highest log-likelihood.

For training our HMM, we will use the `hmmlearn.hmm.GMMHMM` class. We will illustrate how to use this class. Let `data` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Then the model can be trained as follows:

```
import numpy as np
from hmmlearn import hmm

# hmmlearn expects the data to be in a single array:
data_collected = np.vstack(data)
```

[1] For a larger vocabulary, this requires a ludicrous number of GMMHMMs, which is why speech recognition systems don't typically do this.

```
# To separate the sequences, it requires the length of each:
lengths = [item.shape[0] for item in data]

# Initialize and train the model
model = hmm.GMMHMM(n_components=5, covariance_type="diag")
model.fit(data_collected, lengths=lengths)

# Check the log-likelihood
log_likelihood = model.monitor_.history[-1]
```

Problem 3. For each word, randomly split the list of MFCCs into a training set of 20 samples and a test set of the remaining 10 samples.

Use the training sets to train GMMHMMs on each word in the vocabulary. For each word in the vocabulary, train 10 GMMHMMs on the training set, using `n_components=5`. Keep the model with the highest log-likelihood for each word.

We have now trained our speech recognition system. With this model, if we are given a speech sample, how do we determine which word it is?

The method we use is as follows. First, we convert the sound into MFCC coefficients. Let $\mathbf{z}$ denote these coefficients. Then, for each of the words in our vocabulary, we find $\log P(\mathbf{z} \mid \text{word})$, the log probability density of this observation sequence assuming that it came from that word's GMMHMM. This can be done with the `score` method of the `GMMHMM`. Then, whichever word's GMMHMM gives the highest log probability density will be our speech recognition model's prediction.

Problem 4. Write a `predict` function for your speech recognition model. In this function:

- Accept the MFCC coefficients of the speech sample to be predicted.
- Find the log probability density of the coefficients for each word's GMMHMM.
- Return the word with the highest probability as the speech recognition model's prediction.

Problem 5. For each of the five test sets, call your `predict` function on each sample, and find the proportion of each test set that your model predicts correctly. Display your results. How well does your model perform on this dataset?

13 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

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + G_k\mathbf{u}_k + \mathbf{w}_k \tag{13.1}$$

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

$$\mathbf{z}_k = H_k\mathbf{x}_k + \mathbf{v}_k \tag{13.2}$$

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 matrix for observation noise.
        u : ndarray of shape (n,)
            The control vector.
        """
        pass
```

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

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

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

How does the system evolve from one time step to the next? Assuming each time step is 0.1 seconds, it is easy enough to calculate the new position:

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

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 13.1 and 13.2. Bear in mind that the initial state $\mathbf{x}_0$ does not have added noise, but the initial observation and all subsequent states and observations take noise into account.

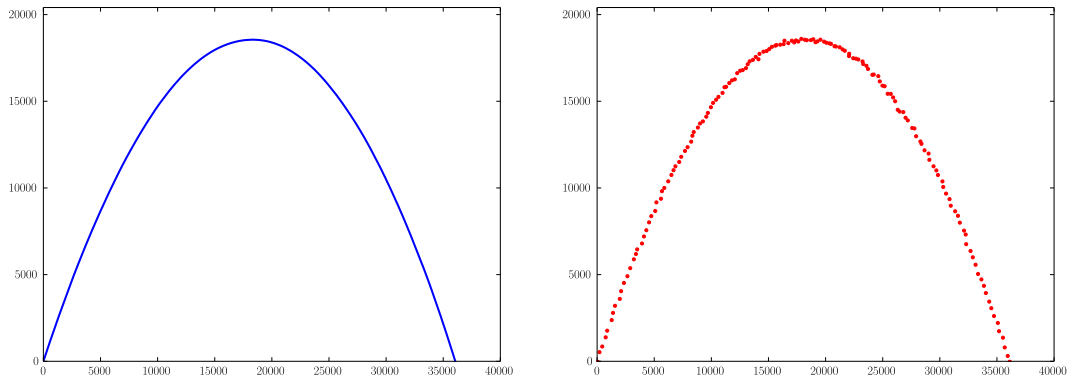


Figure 13.1: State sequence (left) and sampling of observation sequence (right).

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 ↩
        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
```

To test your code, simulate the true and observed trajectories of a projectile with initial state

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

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the y coordinate to return to 0). Your results should qualitatively match those given in Figure 13.1.

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:

Predict

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

Update

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

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.
```

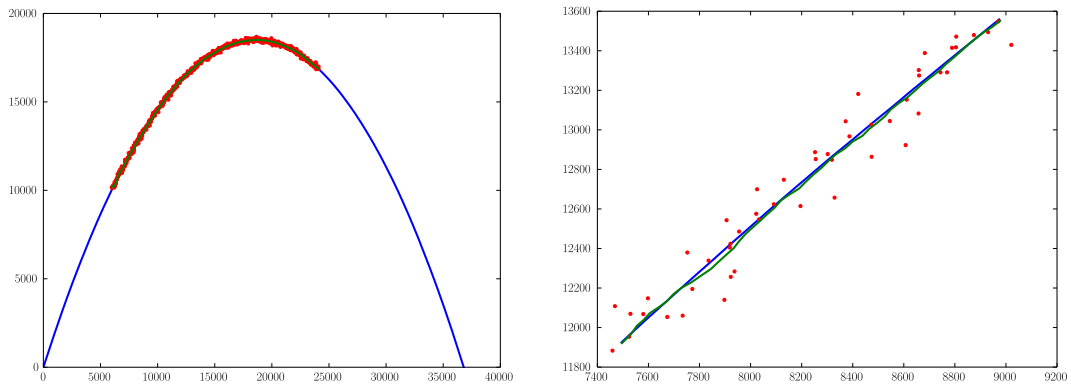


Figure 13.2: State estimates together with observations and true state sequence (detailed view on the right).

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

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

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 13.2.

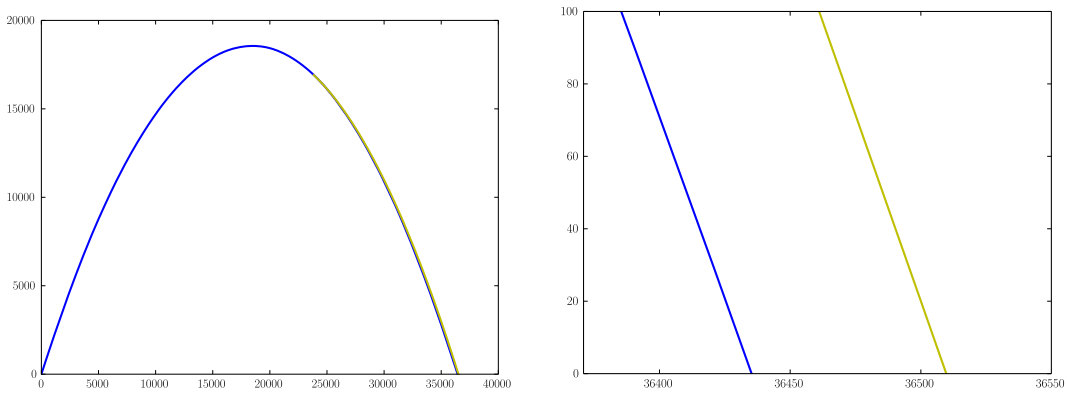


Figure 13.3: Predicted vs. actual point of impact (detailed view on right).

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

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

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 13.3. The following code will help you plot both the actual and predicted state sequences together in a nice plot.

```
>>> s,o = kal.evolve(x0,1250)
>>> predicted = kal.predict(estimates[:,-1],450)

>>> # create a dynamic xlim to always plot both sequences together
>>> x1 = s[0,:][np.where(s[1,:]>=0)][-1]
>>> x2 = predicted[0,:][np.where(predicted[1,:]>=0)][-1]
>>> plt.xlim(min(x1,x2)-50, max(x1,x2)+50)
```

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

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

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

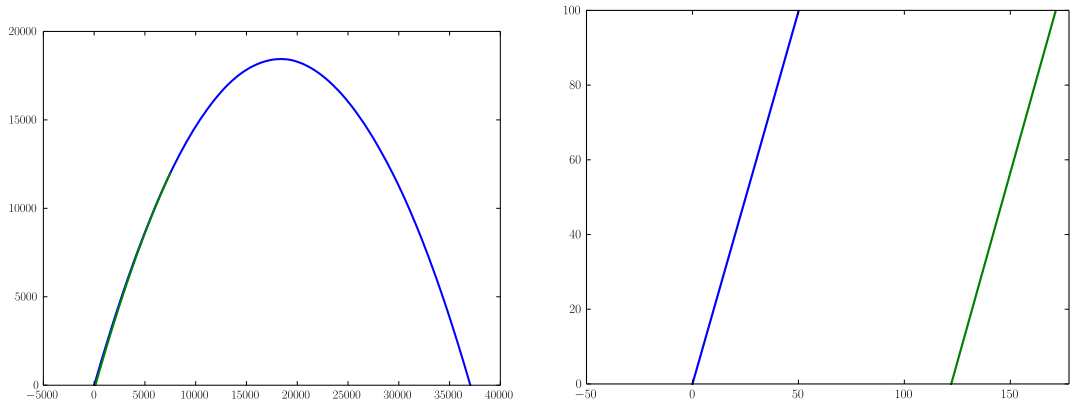


Figure 13.4: Predicted vs. actual point of origin (detailed view on right).

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 green) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 13.4.

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

14 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

$$Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i Z_{t-i}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} + \boldsymbol{\varepsilon_t} \tag{14.1}$$

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:

$$Z_t = \mathbf{c} + \sum_{i=1}^{p} \Phi_i Z_{t-i} + \boldsymbol{\varepsilon_t} \tag{14.2}$$

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.

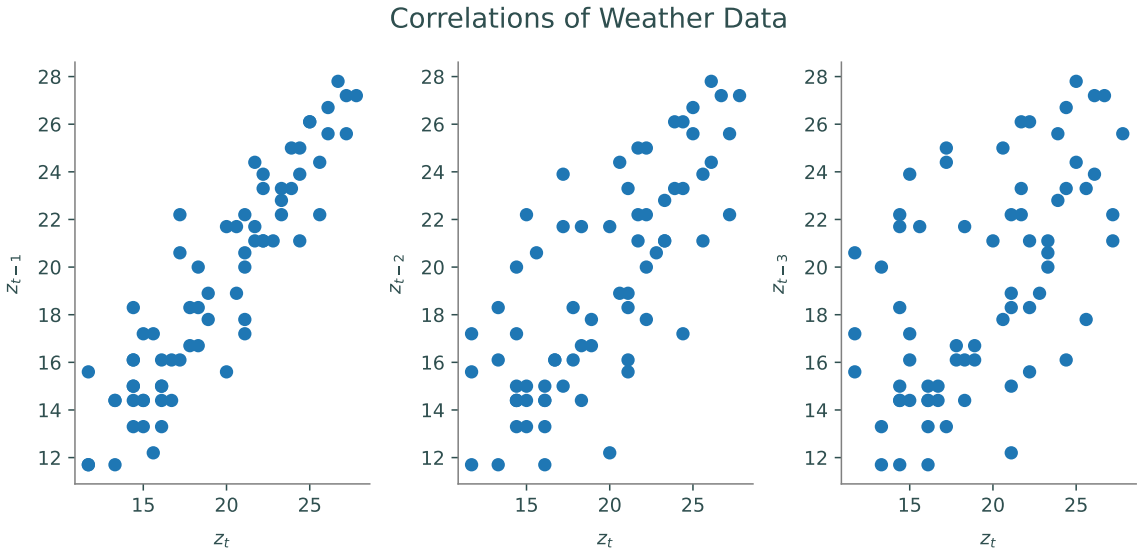


Figure 14.1: These graphs show that the weather data is correlated to its previous values. The correlation is weaker in each graph successively, showing that the further in the past the data is, the less correlated the data becomes.

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..

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

$$Z_t = \mathbf{c} + \boldsymbol{\varepsilon_t} + \sum_{i=1}^{q} \Theta_i \boldsymbol{\varepsilon_{t-i}} \tag{14.3}$$

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

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

ARIMA(p, d, q) Models

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

$$\delta^d Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i z_{t-i}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} + \boldsymbol{\varepsilon_t} \tag{14.4}$$

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 standard deviation of the error. Note that $\{\phi_i\}_{i=1}^p$ and $\{\theta_i\}_{i=1}^q$ determine the order of the ARMA model. For this lab, we will let $\mathbf{c} = 0$.

A naive way to use an ARMA model is to choose p and q based on intuition. Figure 14.1 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 14.1, 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.

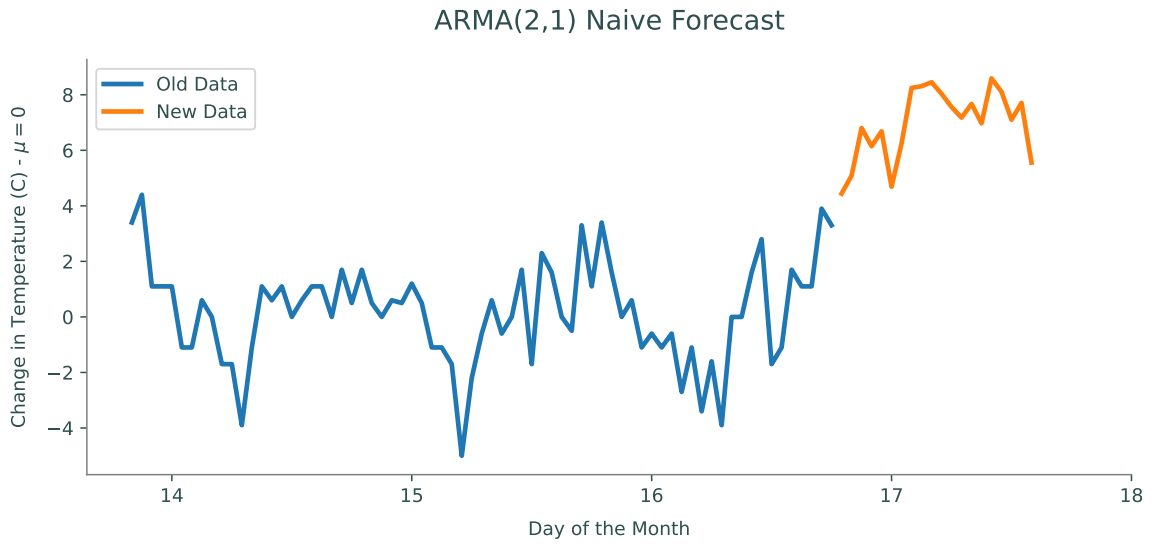


Figure 14.2: Naive forecast on `weather.npy`

Problem 1. Write a function `arma_forecast_naive()` that builds an ARMA(p,q) model. Your function should accept as parameters `p`, `q`, and `n`, where `p` is the order of the autoregressive model, `q` is the order of the moving average model, and `n` is the number of observations to predict. Assume `c`$= 0$, and let $\phi_i = .5$, $\theta_i = .1$, and $\varepsilon_i \sim \mathcal{N}(0, 1)$ for all i.

Use your function to predict the next `n` values of the time series found in `weather.npy`. This file contains data on the temperature in Provo, Utah from 7:56 PM May 13, 2019 to 6:56 PM May 16, 2019, taken every hour. This time series is NOT covariance stationary, so to make it covariance stationary, take its first difference (Hint: you might find `np.diff()` helpful). We denote the new covariance stationary time series as $\{z_t\}_{t=1}^T$. Predict the next `n` observations by iterating through Equation 14.4.

Run your code on `weather.npy`, and plot the observed differences $\{z_t\}_{t=1}^T$ followed by your predicted observations of z_t. For `p=2`, `q=1`, and `n=20`, your plot should look similar to Figure 14.2, however, due to the variance of the error ε_t, the plot will not look exactly like Figure 14.2. The predictions may be higher or lower on the y-axis.

Let $\Theta = \{\phi_i, \theta_j, \mu, \sigma\}$ be the set of parameters for an ARMA(p, q) model. Suppose we have a set of observations $\{z_t\}_{t=1}^n$. Our goal is to find the p, q, and 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

$$p(\{z_t\} \mid \Theta) = \prod_{t=1}^{n} p(z_t \mid z_{t-1}, \dots, z_1, \Theta) \tag{14.5}$$

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

$$\hat{\mathbf{x}}_{t|t-1} = \begin{bmatrix} x_{t-1} & x_{t-2} & \cdots & x_{t-r} \end{bmatrix}^T \tag{14.6}$$

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{14.7}$$

$$H = \begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_{r-1} \end{bmatrix} \tag{14.8}$$

$$Q = \begin{bmatrix} \sigma & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{14.9}$$

$$w_t \sim \text{MVN}(0, Q), \tag{14.10}$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Note that Equation 14.2 gives

$$F\hat{\mathbf{x}}_{t-1|t-2} + w_t = \begin{bmatrix} \sum_{i=1}^{r} \phi_i x_{t-i} \\ x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-(r-1)} \end{bmatrix} + \begin{bmatrix} \varepsilon_t \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{14.11}$$

$$= \begin{bmatrix} x_t & x_{t-1} & \cdots & x_{t-(r-1)} \end{bmatrix}^T \tag{14.12}$$

$$= \hat{\mathbf{x}}_{t|t-1} \tag{14.13}$$

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

Then the linear stochastic dynamical system

$$\hat{\mathbf{x}}_{t+1|t} = F\hat{\mathbf{x}}_{t|t-1} + w_t \tag{14.14}$$

$$z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{14.15}$$

describes the same process as the original ARMA model.

NOTE

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

Equation 14.15 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.

Likelihood via Kalman Filter

We assumed in Equation 14.10 that the error terms of the model are Gaussian. This means that each conditional distribution in 14.5 is also Gaussian, and is completely characterized by its mean and variance:

$$\text{mean} \quad H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{14.16}$$

$$\text{variance} \quad HP_{t|t-1}H^T \tag{14.17}$$

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

$$p(\{z_t\} \mid \Theta) = \prod_{t=1}^{n} \mathcal{N}(z_t \mid H\hat{\mathbf{x}}_{t|t-1} + \mu, HP_{t|t-1}H^T). \tag{14.18}$$

Problem 2. Write a function `arma_likelihood()` that returns the log-likelihood of an ARMA model, given a time series $\{z_t\}_{t=1}^T$. This function should accept `filename` which contains the observations, and it should accept as parameters each parameter in Θ. In this case, the time series should be the change in temperature of `weather.npy`, which is the first difference of the time series found in `weather.npy`, as was done in Problem 1. Adapt Equation 14.18 to calculate and return the log-likelihood of the ARMA(p, q) model as a **float**.

Use the provided `state_space_rep()` function to generate F, Q, and H. The function `kalman()` has also been provided to help calculate the means and covariances of each observation. Calling the function `kalman()` on a time series will return an array whose values are $\hat{\mathbf{x}}_{t|t-1}$ and an array whose values are $P_{t|t-1}$ for each $t \leq n$.

Hint: remember to subtract off the mean μ from the inputted observation when using `kalman()`.

Also, when implementing Equation 14.18, you may find it best to use `scipy.stats.distributions.norm.pdf`, but keep in mind that this method accepts standard deviations, not variances. When implemented correctly, your function should match the following output:

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

Model Identification

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

$$2k\left(1+\frac{k+1}{n-k}\right)-2\ell(\Theta) \tag{14.19}$$

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 14.18 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 `filename` containing the time series data and parameters `p_max` and `q_max` as integers. Determine which ARMA(p,q) model has the minimum AIC for all $1\leq p\leq$ `p_max` and $1\leq q\leq$ `q_max`. Then, return each parameter in Θ of that model.

Hint: when calculating the AIC using Equation 14.19, bear in mind that $-\ell(\Theta)=$ `f(sol)` where `sol` is found in the code above and explained in the following paragraph.

Your code should replicate the following output up to at least 4 decimal places.

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

Forecasting with Kalman Filter

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

$$z_{t+k} \mid z_1, \cdots, z_t \sim \mathcal{N}(z_{t+k} \mid H\hat{x}_{t+k|t} + \mu,\ HP_{t+k|t}H^T) \tag{14.20}$$

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

Predict

$$\begin{aligned} \widehat{\mathbf{x}}_{k|k-1} &= F\widehat{\mathbf{x}}_{k-1|k-1} + \mathbf{u} \\ P_{k|k-1} &= FP_{k-1|k-1}F^T + Q \end{aligned}$$

Update

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

With ARMA, we define observational noise covariance R and drift term $\mathbf{u}$ to both be 0.

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

Problem 4. Write a function `arma_forecast()` that accepts `filename` containing a time series, the parameters for an ARMA model, and the number `n` of observations to forecast. Calculate the mean and covariance of the future `n` observations using the Kalman filter.

To do this, use `state_space_rep()` to generate F, Q, and H. Then, use `kalman()` (with μ subtracted off from the covariance stationary time series $\mathbf{z}_k$) to calculate $\hat{\mathbf{x}}_{k|k-1}$ and $P_{k|k-1}$, respectively. Use the Update step on the last elements of $\mathbf{z}_k$ (with μ subtracted off), $\hat{\mathbf{x}}_{k|k-1}$, and $P_{k|k-1}$ to find $\hat{\mathbf{x}}_{k|k}$ and $P_{k|k}$. Then, iteratively use the Predict step to make future predictions of the mean and covariance. Recall that R and $\mathbf{u}$ are both defined to be 0! Also, remember that once you find a mean $\hat{\mathbf{x}}_{k|k-1}$ and covariance $P_{k|k-1}$, you must use Equations 14.16 and 14.17 to transform them back into observation space.

Plot the original observations as well as the mean of each future observation. Plot a 95% confidence interval (2 standard deviations away from the mean) around the means of future observations. Hint: the standard deviation is the square root of the covariance calculated.

The following code should create a plot similar to Figure 14.3.

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

# Forecast optimal mode
arma_forecast(filename='weather.npy', phis=phis, thetas=thetas,
                    mu=mu, std=std, n=30)
```

How does this graph compare to the naive ARMA graph from Problem 1?

Statsmodel ARMA

The module `statsmodels` contains a package that includes an ARMA model class. This is accessed through ARIMA model, which stands for Autoregressive Integrated Moving Average. This class also uses a Kalman Filter to calculate the MLE. When creating an ARIMA object, initialize the variables `endog` (the data) and `order` (the order of the model). The order is of the form (p, d, q) where d is the differences. To create an ARMA model, set $d = 0$. The object can then be fitted based on the MLE using a Kalman Filter.

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

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

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

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

Problem 5. Write a function `sm_arma()` that accepts `filename` containing a time series, integer values for `p_max` and `q_max`, and the number `n` of values to predict.

As in Problem 3, perform model identification to find the ARMA(p, q) model with the best AIC for $1 \leq p \leq$ `p_max` and $1 \leq q \leq$ `q_max`, but this time use `statsmodels`. Make sure the model is fit using the MLE.

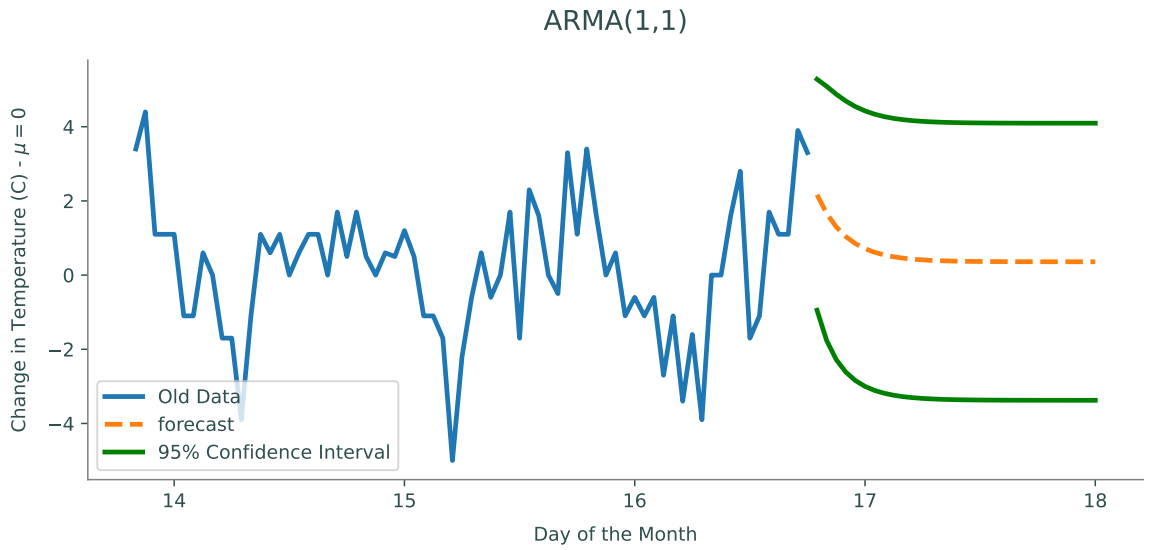


Figure 14.3: ARMA(1,1) forecast on `weather.npy`

Use the optimal model to predict `n` future observations of the time series. Plot the original observations along with the predicted observations from the beginning through `n` observations in the future, as given by `statsmodels`. **Return the AIC** of the optimal model.

For `p_max=3`, `q_max=3`, and `n=30`, your graph should look similar to Figure 14.4. How does this graph compare to Problem 1? Problem 4?

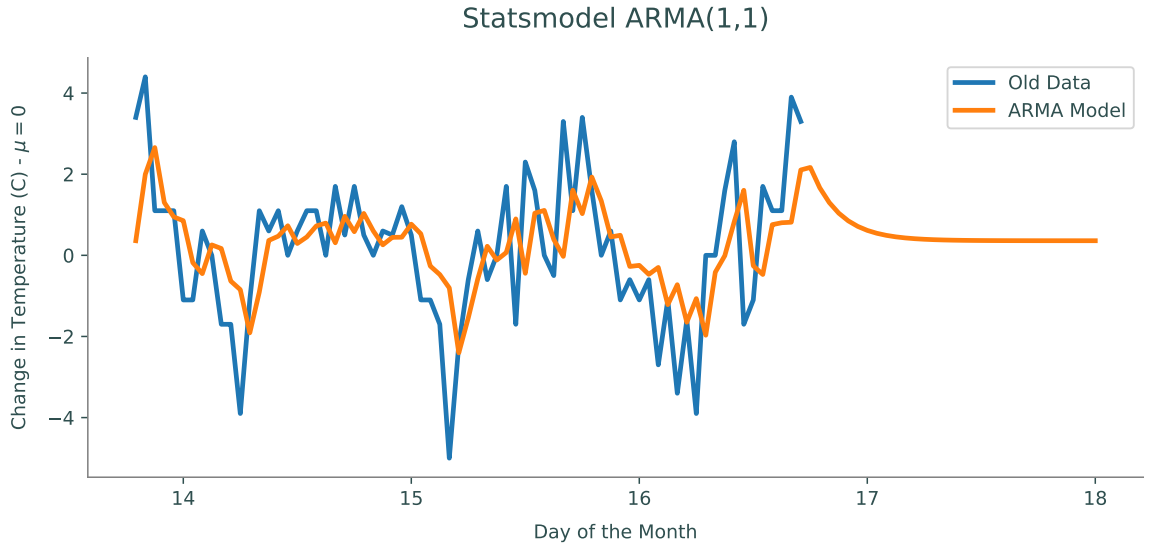


Figure 14.4: Statsmodel forecast on `weather.npy`.

Statsmodel VARMA

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

$$z_{t,1} = c_1 + \phi_{11} z_{t-1,1} + \phi_{12} z_{t-1,2} + \theta_{11} \varepsilon_{t-1,1} + \theta_{12} \varepsilon_{t-1,2} \tag{14.21}$$

$$z_{t,2} = c_1 + \phi_{21} z_{t-1,1} + \phi_{22} z_{t-1,2} + \theta_{21} \varepsilon_{t-1,1} + \theta_{22} \varepsilon_{t-1,2} \tag{14.22}$$

This can be written in matrix form as shown in Equation 14.1. The module `statsmodels` contains a package that includes a VARMAX model class which can be used to create a multivariate ARMA model. VARMAX stands for Vector Autoregression Moving Average with Exogenous Regressors. An exogenous regressor is a time series that affects the model but is not affected by it. In the example below we have two time series corresponding to the price of copper and aluminum. Since aluminum is a substitute for copper, it is reasonable to assume the price of aluminum may help us predict the price of copper and vice versa.

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

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

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

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

# Plot predictions against true price
pred.plot(figsize=(10,4))
plt.plot(data['ALUMPRICE'], label='Actual ALUMPRICE')
plt.plot(data['COPPERPRICE'], label='Actual COPPERPRICE')
plt.legend()
plt.title('VARMA Predictions for World Copper Market Dataset')
```

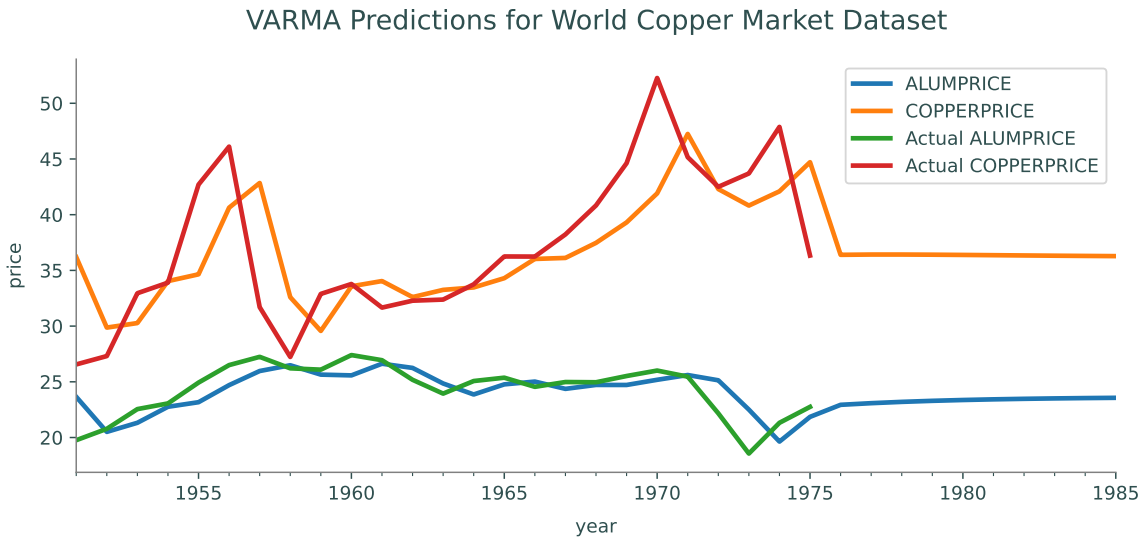


Figure 14.5: Statsmodel VAR(1) forecast.

Problem 6. Write a function `sm_varma()` that accepts start and end dates for forecasting. Use the statsmodels VARMAX class to forecast on macroeconomic data between the start and end dates. The following code shows how to obtain the data.

```
# Load in data
df = sm.datasets.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']]
```

Initialize your VARMAX model with the `df` specified above, and include the parameter `freq="Q-DEC"`. Fit your model, and predict from the start date until the end date. Then, get the model forecast until the end date. Plot the original data, prediction, and a 95% confidence interval (2 standard deviations away from the mean) around the future observations. **Return the AIC** of the chosen model. The plot should be similar to Figure 14.6.

Hint: in the example above, `mod.predict('1951','1985')` returns a dataframe of 2 columns that contain the predicted values of `'ALUMPRICE'` and `'COPPERPRICE'`, respectively, from the years 1951 to 1985. Also, `all_CI` is a dataframe where each column indicates the corresponding dataset and whether it is a lower or upper bound of a confidence interval determined by the alpha value. Thus, the column `'lower ALUMPRICE'` with `alpha=0.05` contains the lower bounds of a 95% confidence interval for the `'ALUMPRICE'` dataset.

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

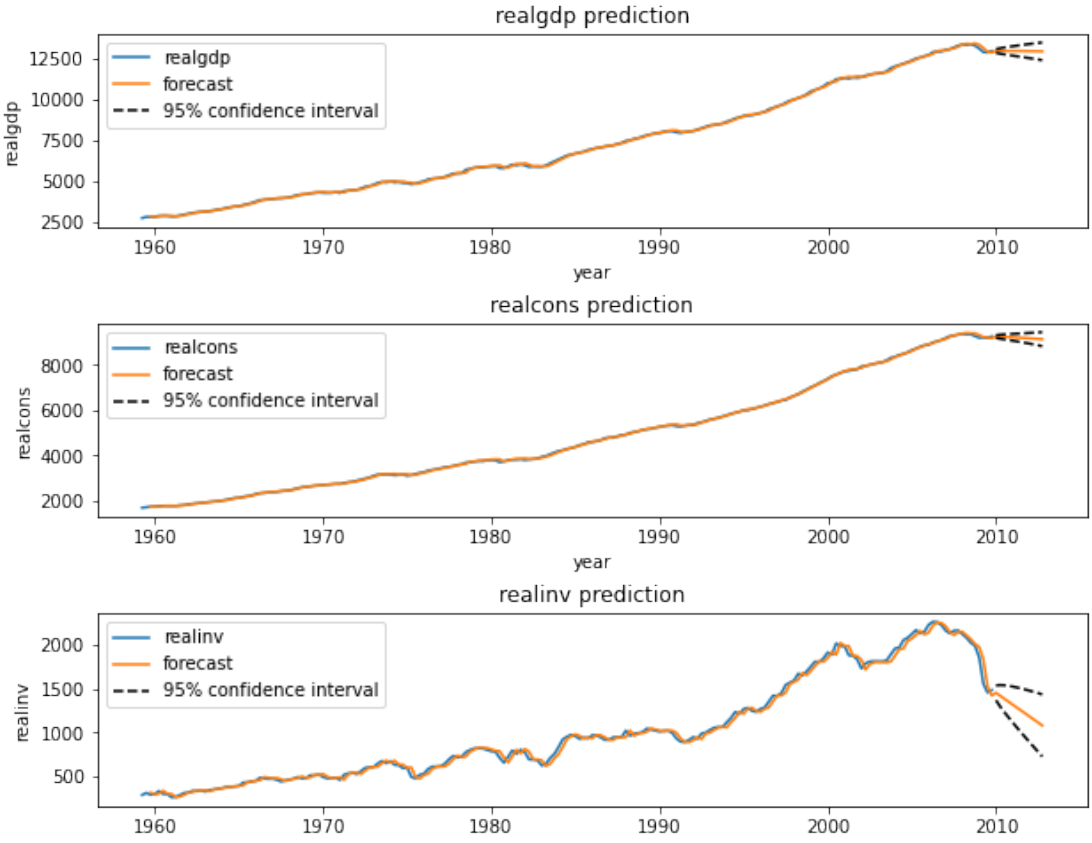


Figure 14.6: Macroeconomic data is forecasted 12 years in the future using statsmodels.

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()
```

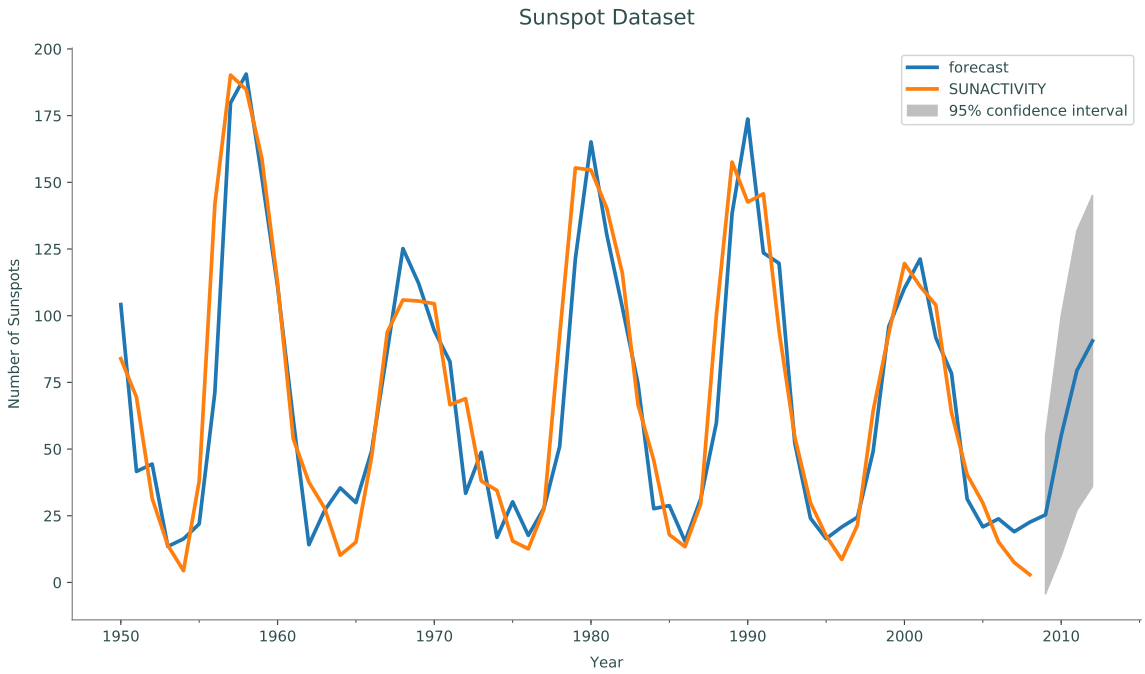


Figure 14.7: Sunspot activity data is forecasted four years in the future using `statsmodels`.

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 14.8.

(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']
```

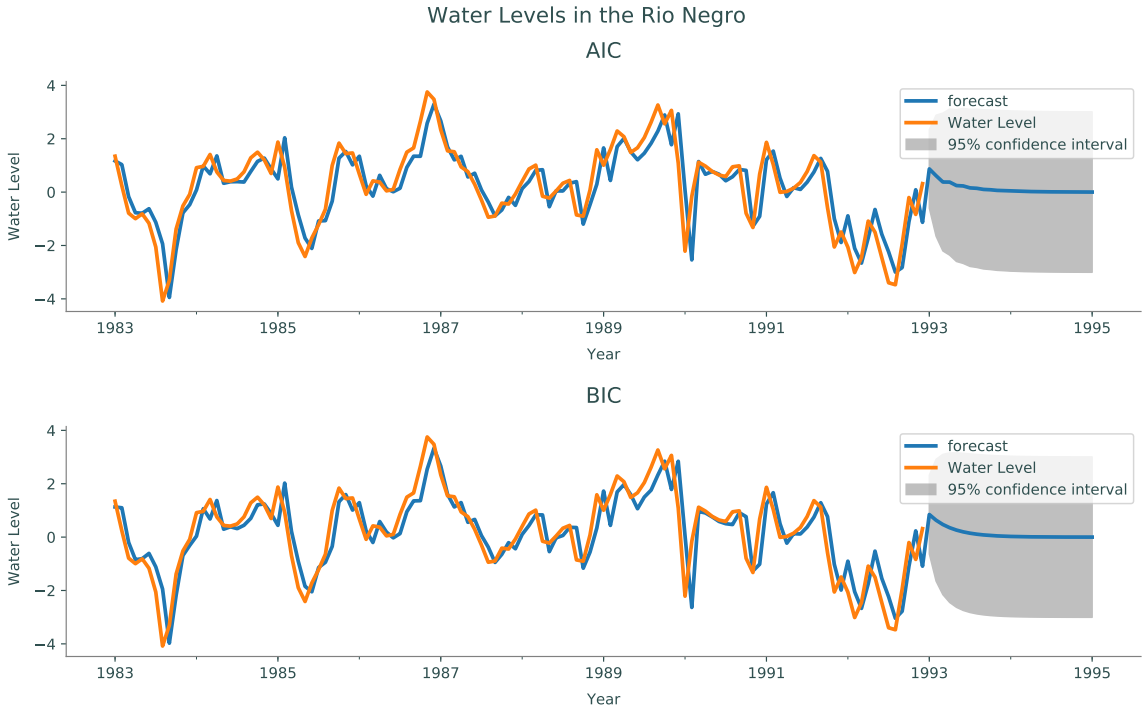


Figure 14.8: AIC and BIC based ARMA models of `manaus` dataset.

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

$$y_t = T_t + S_t + R_t, \tag{14.23}$$

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

$$T_t = \frac{1}{M} \sum_{-M/2<i<M/2} y_{i+t}. \tag{14.24}$$

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,

$$S_t = \frac{1}{K} \sum_{i \equiv t \pmod{M}} x_i \tag{14.25}$$

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 14.1.

Proof of Equation 14.15

$$\sum_{i=1}^{p} \phi_i(z_{t-i} - \mu) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} = \sum_{i=1}^{p} \phi_i(H\hat{\mathbf{x}}_{t-i}) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} \tag{14.26}$$

$$= \sum_{i=1}^{r} \phi_i(x_{t-i} + \sum_{k=1}^{r-1} \theta_k x_{t-i-k}) + a_t + \sum_{j=1}^{r-1} \theta_j a_{t-j} \tag{14.27}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j \Big(\sum_{i=1}^{r} \phi_i x_{t-j-i} + a_{t-j} \Big) \tag{14.28}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j x_{t-k} \tag{14.29}$$

$$= x_t + \sum_{j=1}^{r-1} \theta_j x_{t-k} \theta_k x_{t-k} \tag{14.30}$$

$$= z_t. \tag{14.31}$$

15 Non-negative Matrix Factorization Recommender

Lab Objective: *Understand and implement the non-negative matrix factorization for recommendation systems.*

Introduction

Collaborative filtering is the process of filtering data for patterns using collaboration techniques. More specifically, it refers to making prediction about a user's interests based on other users' interests. These predictions can be used to recommend items and are why collaborative filtering is one of the common methods of creating a recommendation system.

Recommendation systems look at the similarity between users to predict what item a user is most likely to enjoy. Common recommendation systems include Netflix's Movies you Might Enjoy list, Spotify's Discover Weekly playlist, and Amazon's Products You Might Like.

Non-negative Matrix Factorization

Non-negative matrix factorization is one algorithm used in collaborative filtering. It can be applied to many other cases, including image processing, text mining, clustering, and community detection. The purpose of non-negative matrix factorization is to take a non-negative matrix V and factor it into the product of two non-negative matrices.

For $V \in \mathbb{R}^{m\times n}, 0 \preceq W$,

$$\begin{aligned} \text{minimize} \quad & ||V - WH|| \\ \text{subject to} \quad & 0 \preceq W, 0 \preceq H \\ \text{where} \quad & W \in \mathbb{R}^{m\times k}, H \in \mathbb{R}^{k\times n} \end{aligned}$$

k is the rank of the decomposition and can either be specified or found using the Root Mean Squared Error (the square root of the MSE), SVD, Non-negative Least Squares, or cross-validation techniques.

For this lab, we will use the Frobenius norm, given by

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a|_{ij}^2}.$$

It is equivalent to the square root of the sum of the diagonal of $A^H A$

Problem 1. Create the `NMFRecommender` class, which will be used to implement the NMF algorithm. Initialize the class with the following parameters: `random_state` defaulting to 15, `tol` defaulting to $1e-3$, `maxiter` defaulting to 200, and `rank` defaulting to 2.

Add a method called `_initialize_matrices` that takes in m and n, the dimensions of V. Set the random seed so that initializing the matrices can be replicated.

```
>>> np.random.seed(self.random_state)
```

Then, using `np.random.random`, initialize W and H with randomly generated numbers between 0 and 1, where $W \in \mathbb{R}^{m\times k}$ and $H \in \mathbb{R}^{k\times n}$. Return W and H.

Finally, add a method called `_compute_loss()` that takes as parameters `V`, `W`, and `H` and returns the Frobenius norm of $V - WH$.

Multiplicative Update

After initializing W and H, we iteratively update them using the multiplicative update step. There are other methods for optimization and updating, but because of the simplicity and ease of this solution, it is widely used. As with any other iterative algorithm, we perform the step until the `tol` or `maxiter` is met.

$$H_{ij}^{s+1} = H_{ij}^{s}\frac{((W^s)^T V)_{ij}}{((W^s)^T W^s H^s)_{ij}} \tag{15.1}$$

and

$$W_{ij}^{s+1} = W_{ij}^{s}\frac{(V(H^{s+1})^T)_{ij}}{(W^s H^{s+1}(H^{s+1})^T)_{ij}} \tag{15.2}$$

Problem 2. Add a method to the `NMF` class called `_update_matrices` that takes as inputs matrices V, W, H and returns W_{s+1} and H_{s+1} as described in Equations 15.1 and 15.2.

Problem 3. Finish the `NMF` class by adding a method `fit` that finds an optimal W and H.

It should accept `V` as a numpy array, perform the multiplicative update algorithm until the loss is less than `tol` or `maxiter` is reached, and return W and H.

Call the function `_initialize_matrices()` in order to run `_update_matrices()` and `_compute_loss()`.

Finally add a method called `reconstruct` that reconstructs and returns `V` by multiplying `W` and `H`.

Using NMF for Recommendations

Consider the following marketing problem where we have a list of five grocery store customers and their purchases. We want to create personalized food recommendations for their next visit. We start by creating a matrix representing each person and the number of items they purchased in different grocery categories. So from the matrix, we can see that John bought two fruits and one sweet.

$$V = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 2 \\ 2 & 3 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 2 & 3 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

After performing NMF on V, we'll get the following W and H.

$$W = \begin{array}{c} \begin{matrix} Component1 & Component2 & Component3 \end{matrix} \\ \begin{pmatrix} 2.1 & 0.03 & 0. \\ 1.17 & 0.19 & 1.76 \\ 0.43 & 0.03 & 0.89 \\ 0.26 & 2.05 & 0.02 \\ 0.45 & 0. & 0. \end{pmatrix} \end{array} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

$$H = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0.00 & 0.45 & 0.00 & 0.43 & 1.0 & 0.9 \\ 0.00 & 0.91 & 1.45 & 1.9 & 0.35 & 0.37 \\ 1.14 & 1.22 & 0.55 & 0.0 & 0.47 & 0.53 \end{pmatrix} \end{array} \begin{array}{l} Component1 \\ Component2 \\ Component3 \end{array}$$

W represents how much each grocery feature contributes to each component; a higher weight means it's more important to that component. For example, component 1 is heavily determined by vegetables followed by fruit, then coffee, sweets and finally bread. Component 2 is represented almost entirely by bread, while component 3 is based on fruits and sweets, with a small amount of bread. H is similar, except instead of showing how much each grocery category affects the component, it shows a much each person belongs to the component, again with a higher weight indicating that the person belongs more in that component. We can see the John belongs in component 3, while Jennifer mostly belongs in component 1.

To get our recommendations, we reconstruct V by multiplying W and H.

$$WH = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0.0000 & 0.9723 & 0.0435 & 0.96 & 2.1105 & 1.9011 \\ 2.0064 & 2.8466 & 1.2435 & 0.8641 & 2.0637 & 2.0561 \\ 1.0146 & 1.3066 & 0.533 & 0.2419 & 0.8588 & 0.8698 \\ 0.0228 & 2.0069 & 2.9835 & 4.0068 & 0.9869 & 1.0031 \\ 0.0000 & 0.2025 & 0.0000 & 0.1935 & 0.45 & 0.405 \end{pmatrix} \end{array} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

Most of the zeros from the original V have been filled in. This is the `collaborative filtering` portion of the algorithm. By sorting each column by weight, we can predict which items are more attractive to the customers. For instance, Mary has the highest weight for bread at 2.9835, followed by fruit at 1.2435 and then sweets at .533. So we would recommend bread to Mary.

Another way to interpret WH is to look at a feature and determine who is most likely to buy that item. So if we were having a sale on sweets but only had funds to let three people know, using the reconstructed matrix, we would want to target Alice, John, and Jennifer in that order. This gives us more information that V alone, which says that everyone except Greg bought one sweet.

Problem 4. Use the `NMFRecommender` class to run NMF on V, defined above, with 2 components. Return W, H as matrices, and the number of people who have higher weights in component 2 than in component 1 as a float.

Sklearn NMF

Python has a few packages for recommendation algorithms: Surprise, CaseRecommender and of course SkLearn. They implement various algorithms used in recommendation models. We'll use SkLearn, which is similar to the NMFRecommender class, for the last problems.

```
from sklearn.decomposition import NMF

>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
```

As mentioned earlier, many big companies use recommendation systems to encourage purchasing, ad clicks, or spending more time in their product. One famous example of a Recommendation system is Spotify's Discover Weekly. Every week, Spotify creates a playlist of songs that the user has not listened to on Spotify. This helps users find new music that they enjoy and keeps Spotify at the forefront of music trends.

Problem 5. Read the file `artist_user.csv` as a pandas dataframe. The rows represent users, with the user id in the first column, and the columns represent artists. For each artist j that a user i has listened to, the ij entry contains the number of times user i has listened to artist j.

Identify the rank, or number of components to use. Ideally, we want the smallest rank that minimizes the error. However, this rank may be too computationally expensive, as in this situation. We'll choose the rank by using the following method. First, calculate the frobenius norm of the dataframe and multiply it by .0001. This will be our benchmark value. Next, iterate through `rank`$= 10, 11, 12, 13, \ldots$. For each iteration, run NMF using `n_components=rank` and `random_seed=0` and reconstruct the matrix V. Calculate the root mean square error (RMSE) by taking the square root of the MSE – calculated by `sklearn.metrics.mean_squared_error` – of the original dataframe and the reconstructed matrix V. If the RMSE is less than the benchmark value, stop. Return the rank and the reconstructed matrix of this rank.

Hint: the optimal rank can be found between 10 and 15, so you only actually need to iterate through `rank`$= 10, \ldots, 15$.

Problem 6. Write a function `discover_weekly` that takes in a user id and the reconstructed matrix from Problem 5, and returns a list of 30 artists to recommend as strings.

This list of strings should be sorted so that the first artist is the recommendation with the highest weight and the last artist is the least, and it should not contain any artists that the user has already listed to. Use the file `artists.csv` to match the artist ID to their name.

As a check, the Discover Weekly for user 2 should return

['Britney Spears', 'Avril Lavigne', 'Rihanna', 'Paramore', 'Christina Aguilera', 'U2', 'The Devil Wears Prada', 'Muse', 'Hadouken!', 'Ke$ha', 'Good Charlotte', 'Linkin Park', 'Enter Shikari', 'Katy Perry', 'Miley Cyrus', 'Taylor Swift', 'Beyoncé', 'Asking Alexandria', 'The Veronicas', 'Mariah Carey', 'Martin L. Gore', 'Dance Gavin Dance', 'Erasure', 'In Flames', '3OH!3', 'Blur', 'Kelly Clarkson', 'Justin Bieber', 'Alesana', 'Ashley Tisdale']

16 Deep Learning

Lab Objective: *Deep Learning is a popular method for machine learning tasks that have large amounts of data, including image recognition, voice recognition, and natural language processing. In this lab, we use PyTorch to write a convolution neural net to classify images. We also look at one of the challenges of deep learning by performing an adversarial attack on our model.*

Intro to Neural Networks

An *artificial neural network* is a machine learning tool inspired by the idea of neurons passing information between each other to learn. The network is composed of layers of neurons, usually called *nodes*, that are connected in various ways. Each connection has a *weight* based on its importance, which is used as information is passed through the network from one layer to the next. For example, in Figure 16.1, the yellow input is passed to the first layer, blue, then the green layer, and then to the final output layer.

The middle layers for a neural network are considered "hidden" because they're not directly viewable to the outside world. You can view them but they will be a mess of seemingly random numbers, not the helpful classification labels you would get from an end layer.

In a neural network, the input is often images, text, or sounds represented as a vector of real numbers. In order to evaluate a network, these values are then multiplied by the weight of each pertinent edge, and all respective values are added up to create the node value. A vector called the *bias* of the layer is added to each respective node, finalizing the linear combination step. An *activation function* takes these node values and applies a nonlinear transformation to them, which allows the model to learn complex nonlinear transformations between the input and output. Without these activation functions, the network would be linear and exactly the same as a network with no hidden layers. Mathematically, a typical neural network looks like the nested function composition

$$f_N(\mathbf{x}) = \mathbf{a}(W_k\mathbf{a}(\dots\mathbf{a}(W_2\mathbf{a}(W_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2)\dots)+\mathbf{b}_k)$$

where $\mathbf{a}$ is the activation function, and W_i and $\mathbf{b}_i$ are the weights and biases of each layer of the neural network. The model is trained by adjusting the weights and biases, typically using a variant of gradient descent, until the model output accurately matches the training labels.

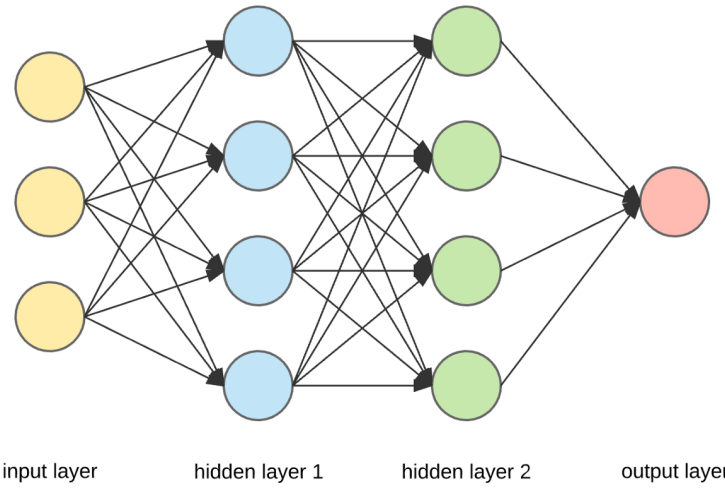


Figure 16.1: A high level diagram of an artificial neural network.

Intro to PyTorch

PyTorch is an open source machine learning library developed by Facebook AI Research. It's mainly used for fast GPU processing of deep neural networks (neural networks with many hidden layers). GPUs (graphics processing units) are designed to compute thousands of operations at once and are vital for parallelizing the operations used by neural networks, which is why they are used here. For more information and documentation on PyTorch, visit `https://pytorch.org/`

We will be working in Google's Colaboratory, `https://colab.research.google.com/notebooks/intro.ipynb`. Colab notebooks use Google's cloud servers, which have a built-in GPU. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a popup called `Notebook Settings`. Under *Hardware Settings*, select GPU.

We can verify that GPU is enabled by calling `torch.cuda.is_available()`. If this function returns `False`, a GPU is not available, and the code will be run on the CPU.

CUDA is a parallel computing platform for GPU computing. The PyTorch package `torch.cuda` interfaces with this package and allows code to be run on the GPU.

PyTorch represents vectors and arrays using *tensors*. A tensor is a data structure similar to a numpy array that is designed to be compatible with GPUs. Like a numpy array, it has a shape, data type, and can be multi-dimensional.

In order for a tensor to be used by the GPU, it must be stored on the GPU. A tensor can be sent directly to the GPU using `variable.cuda()`; however, if the GPU is not available, this will cause an error. A more flexible approach is store which deivce we are doing computations on as a variable `device`. We will prefer to use a GPU if available, but will use the CPU if a GPU is not available. Then, we can send our variables to the correct location in both cases by using `variable.to(device)`:

```
>>> import torch

>>> x = torch.tensor([3., 4.])                  # Create tensor on CPU
>>> y = torch.tensor([1., 2.]).cuda()           # Create tensor on GPU

# Create the device, choosing GPU if available
>>> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
>>> z = torch.tensor([1., 2.]).to(device)       # Create tensor on device
```

You can check which device a variable is on by displaying it. If it is on a GPU, it will list which number it is. `cuda:0` means that the device running is the default GPU. If you are using a machine that has multiple GPUs, you can set the device to be a specific GPU by changing the number. In Colab, only `cuda:0` is available.

```
>>> x
tensor([3., 4.])                                # Check location of x (CPU)

>>> x = x.to(device)                            # Move x to GPU
>>> x                                           # Check location of x (GPU 0)
tensor([3., 4.], device='cuda:0')
```

ACHTUNG!

Cross-GPU operations are not allowed. This means that the model and data must all be on the same device. If the model is called on data that is on a different device, say the model is located on the GPU and the data is on the CPU, you will get the following runtime exception:

```
RuntimeError: Input type (torch.FloatTensor) and weight type (torch.cuda.↩
    FloatTensor) should be the same.
```

If you get this error, you will need to move one the variables so that they are all on the same device.

Data

For this lab, we will be using the CIFAR10 dataset. It consists of $60,000$ images of size 32×32, represented as a $3 \times 32 \times 32$ matrix, where the 3 channels describe the colors using RGB. The images are evenly split into ten classes represented by the numbers $0 - 9$: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

For convenience, the dataset is already split into a training and a testing set; however, we will also add a *validation* set.

Using a train-validate-test split is good practice in general, because usually we will want to test and compare different models and hyperparameter choices iteratively until we arrive at one that works well for our problem. Once we are done training, we would like to use the test set to determine how well our model fits the data. However, if we use the test set to compare how well each of our models performs, it effectively becomes a second train set that we are learning by trying different models, and using the test set to determine how well our model works on the whole dataset is no longer really valid. As such, it is better practice to use a three-way split. We train each model on the train set, use the validation set to compare the models, and use the test set to determine if our final model appears to fit the data well.

The CIFAR10 dataset is split into a train set of 50,000 images and a test set of 10,000 images. We will split the original train set into a new train set of 40,000 images and a validation set of 10,000 images. To use the data, we must transform it into PyTorch tensors. We also will normalize the data, as this generally improves the results. We will normalize the values to have mean 0 and standard deviation 1 for each component. Finally, we will split the dataset and place it inside a `torch.utils.data.DataLoader` class for easier manipulation.

To load the dataset, we use the `torchvision.datasets.CIFAR10` function, which accepts a folder for the data to be stored in. Some important keyword arguments are listed in Table 16.1.

Parameter	Explanation
`train`	Whether to get the training data or the test data.
`download`	Whether to download the data. You usually only need this the first time you access the dataset. Note that restarting Google Colab will require re-downloading the data, however.
`transform`	Applies the given `transform` when loading the data. This transform always should convert the data into a PyTorch tensor.

Table 16.1: Parameters of the `datasets.CIFAR10` loading function

We can use the `transform` parameter in particular to easily normalize our data. PyTorch has a module `torchvision.transforms` to make creating these transformations easier. In this case, we want to use `transforms.ToTensor` to convert the data into tensors, and then `transforms.Normalize` to normalize the data. The `Normalize` object accepts the desired mean and standard deviation after normalization. We can use `transforms.Compose` to combine these together into a single transform object:

```
>>> from torchvision import transforms

# Transform data into a tensor and normalize
>>> transform = transforms.Compose([
...         transforms.ToTensor(),
...         transforms.Normalize((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))
...])
```

We can then load the data:

```
>>> from torchvision import datasets

# Download the CIFAR10 training data to ../data
>>> train_data = datasets.CIFAR10('../data', train=True, download=True, ↩
    transform=transform)
```

The data can then be accessed using indexing. Each data point is a tuple consisting of the $3 \times 32 \times 32$ image and its class. You can also see the specs of the dataset by calling it without an index.

```
# Get the first training data point
>>> train_data[0]
```

```
(tensor([[[ 0.2314, ..., 0.5804],
          [ 0.0627, ..., 0.4784],
          ...,
          [ 0.7059, ..., 0.3255],
          [ 0.6941, ..., 0.4824]],

         [[ 0.2431, ..., 0.4863],
          [ 0.0784, ..., 0.3412],
          ...,
          [ 0.5451, ..., 0.2078],
          [ 0.5647, ..., 0.3608]],

         [[ 0.2471, ..., 0.4039],
          [ 0.0784, ..., 0.2235],
          ...,
          [ 0.3765, ..., 0.1333],
          [ 0.4549, ..., 0.2824]]]), 6)

# Get the class of the first training data point
>>> train_data[0][1]
6

# Get the specs of the CIFAR10 training set
>>> train_data
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ../data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.0, 0.0, 0.0), std=(1.0, 1.0, 1.0))
           )
```

Problem 1. Create the `device` variable as indicated above. Download the CIFAR10 training and test datasets, transform them into tensors, and normalize them as described above.

PyTorch has a special class `DataLoader` that splits the data into batches for easy manipulation. Sending individual data points to the GPU one at a time to be processed by our model is very inefficient, as it makes it impossible for the GPU to parallelize the computations. Instead, we use *batches*, and send multiple data points together. Using larger batch sizes allows us to take advantage of GPUs, speeding up the training time. Storing all of the data on the GPU is, however, generally impossible due to memory constraints. Using too large of a batch size will cause out of memory issues, and tends to reduce the effectiveness of training. Typical batch sizes are powers of 2: 32, 64, 128, 256.

The `DataLoader` class accepts the dataset as its first argument. The dataset can be a dataset object like the one we created above, a list containing the data points, or any iterable. For the train set, we will first split the loaded data into two lists to create the actual train and validation sets. The dataset object does *not* support fancy indexing, so this step should be done with list comprehension:

```
>>> actual_train_data = [train_data[i] for i in range(40_000)]

>>> from torch.utils.data import DataLoader

# Create a DataLoader from the shuffled training data
>>> train_loader = DataLoader(actual_train_data, batch_size=36, shuffle=True)
# and similarly for the validation set
```

The data is not ordered by its classes, so directly indexing like this will put a good mixture of all of the classes into both sets. For the test set, we can just directly pass the dataset object into the `DataLoader`. Some other useful parameters of the `DataLoader` class are listed in Table 16.2.

Parameter	Explanation
`batch_size`	The size of batch to use
`shuffle`	Whether to shuffle the data
`num_worker`	The number of processes to use, in order to load the data in parallel

Table 16.2: Parameters of the `DataLoader` object

Once we have the data in the `DataLoader` class, we can iterate through it to get data points. We can turn it into an iterator using the `iter` method, and then get batches one-at-a-time using the `next` method:

```
# Get the 36 images of size 3x32x32 and labels in the first batch
>>> dataiter = iter(train_loader)
>>> images, labels = next(dataiter)
>>> images.size()
torch.Size([36, 3, 32, 32])

>>> images[0].size()
torch.Size([3, 32, 32])

>>> labels[0]
tensor(8)
```

This method is particularly useful if we just need a few images. We can also directly iterate through all of the images using a `for` loop:

```
>>> for batch, (x, y_truth) in enumerate(train_loader):
...     # Move to the GPU
...     x, y_truth = x.to(device), y_truth.to(device)
...     # ...
```

This will be a more convenient method for training.

Problem 2. Split the data into train, validate, and test sets, and create DataLoaders for each one. The train set should have 40,000 data points and the test and validate sets should each have 10,000 data points. Use a batch size of 32 for the training set and 1 for the validation and test sets. Specify `shuffle=True` for the training set, and `shuffle=False` for the validation and test sets (this is common practice in deep learning).

Neural Networks in PyTorch

Before creating a good model for this dataset, we will start with a simple model to illustrate how to set up a neural network in PyTorch. This model will use only fully-connected linear layers and activation functions. First, we need to import the `nn` module, which contains all of the classes we need for this:

```
from torch import nn
```

Simple layers

A linear layer takes an input vector x and outputs $Ax + b$ for a learned weight matrix A and bias vector b. This is implemented in Pytorch as `nn.Linear(in_features, out_features)`, where `in_features` is the length of the input vector and `out_features` is the desired length of the output vector. This is called a *fully-connected* layer, because every entry of A and b are allowed to be nonzero.

After each layer, we want to pass the values through an *activation function*. This allows the model to be nonlinear, allowing it to learn much more complicated behaviors than it would otherwise. The most commonly used activation function is the Rectified Linear Unit (ReLU) function:

$$\text{ReLu}(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

This activation function avoids many issues that other activation functions have, and is used almost universally. For the final activation function, however, we will use a different activation function: the *softmax* function

$$\text{Softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}} \right).$$

The components of the output of the softmax function are all non-negative and sum to 1. This allows the output of the final layer to be interpreted as probabilities, which is useful for classification. The component with the highest probability will be the neural network's prediction for the input image. This also enables the use of cross-entropy as a very natural loss function, which will be discussed later. These two activation functions are available as `nn.ReLU` and `nn.Softmax`.

Creating a model

To create a neural network in PyTorch, we begin by creating a class that inherits from `nn.Module`:

```
class NNExample(nn.Module):
```

The class `nn.Module` handles internals so that training is simpler, as well as providing a variety of useful methods. In the initializer of our class, we need to call the *superconstructor* `super().__init__()` to initialize the `nn.Module` itself. Then, we initialize all of the layers we want to use in our model. For this example, we will use two fully-connected linear layers with activation functions after each. Since our inputs are $3 \times 32 \times 32$ tensors and linear layers only work with vectors, we will also include an `nn.Flatten` layer.

```
    def __init__(self):
        # Initialize nn.Module
        super().__init__()

        # Create our layers
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(in_features=3*32*32, out_features=100)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(in_features=100, out_features=10)
        self.softmax = nn.Softmax(dim=1)
```

We need to set all of these layers as members of our class in order for them to be properly detected in the training process. Notice that the input dimension of the first layer (`linear1`) is equal to the dimension of the (flattened) input image ($3 \times 32 \times 32$), but from there, the output dimension is chosen arbitrarily to be 100. The second layer (`linear2`) must then have input dimension equal to the output dimension of `linear1`, but its output dimension is chosen to be 10, which is the number of classes possible in the CIFAR10 dataset.

Lastly, we define the `forward()` method, which calls all of the layers on an input image to give us the output:

```
    def forward(self, x):
        x_flat = self.flatten(x)
        x_layer1 = self.relu(self.linear1(x_flat))
        output = self.softmax(self.linear2(x_layer1))
        return output
```

Even though each layer is really a class (note how we initialize them in `__init__()`), we can call them as if they are functions. Any layer that contains *learnable parameters* (for example, the weights present in linear layers), must be called individually in the `forward()` method, as otherwise this would force it to reuse the parameters and reduce training effectiveness. However, for layers that do not have learnable parameters, such as activation functions, we can safely reuse them and call them multiple times in the `forward()` method. Hence, `nn.ReLU()` only needs to be defined once in the `__init__()` method, even when it may be called multiple times in the `forward()` method.

This neural network would likely perform very poorly on the CIFAR10 dataset, however; only using fully-connected linear layers does not work well for images. For a better method, we will turn to *convolutional neural networks*.

Convolutional neural networks (CNNs) are a type of neural network that use *convolution layers*. They also commonly use *pooling layers* They are particularly well-suited to working with images, such as the CIFAR10 dataset. We now discuss these components and how to use them in PyTorch.

Convolution Layers

A convolution layer takes a two-dimensional array of weights called a *kernel* (sometimes called a filter) and multiplies it by the input at all possible locations, "sliding" around the input. It is particularly useful when working with images, as it preserves and extracts spacial structures, unlike fully-connected linear layers.

Consider the following 5×5 input image and 3×3 kernel:

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Image

1	0	-1
1	0	-1
1	0	-1

3×3 Kernel

To get each value in the output, the kernel is multiplied element-wise by $3x3$ squares inside the input and summed. For the top left square in this example, the output is

$$2 \cdot 1 + 4 \cdot 0 + 7 \cdot (-1) + 9 \cdot 1 + 7 \cdot 0 + 1 \cdot (-1) + 8 \cdot 1 + 3 \cdot 0 + 4 \cdot (-1) = 7.$$

$2 \cdot 1$	$4 \cdot 0$	$7 \cdot (-1)$	6	2
$9 \cdot 1$	$7 \cdot 0$	$1 \cdot (-1)$	2	1
$8 \cdot 1$	$3 \cdot 0$	$4 \cdot (-1)$	5	8
4	3	3	1	2
5	2	1	5	3

5×5 Input Images

7	$\cdots$	
$\vdots$		

3×3 Output

The 7 represents a feature of the 3×3 block in the top left corner. With a trained network applied to the image, these features can represent things such lines, curves, and colors, or even more complicated objects like a nose.

The *stride* of a convolution is how much the kernel slides at a time as it passes over the input. With out example, if the kernel slides one spot over (has a stride of 1), there will be 9 submatrices inside the input image that will be used, and we would get a 3×3 matrix as output. If we used a stride of 2 instead, only the top-left, top-right, bottom-left, and bottom-right submatrices would be used.

Notice that as the kernel slides around the image, the inside values are used in more multiplications than the outside value, causing us to lose information, especially about the corners. If we want to keep more infomation about the edges of the image, we can use *padding*. Padding consists of adding a border around the input, usually filled with zeros. This can also allow the output of the layer to be the same size as the input.

2	4	7	6	2
9	7	1	2	1
8	3	4	5	8
4	3	3	1	2
5	2	1	5	3

5×5 input image

0	0	0	0	0	0	0
0	2	4	7	6	2	0
0	9	7	1	2	1	0
0	8	3	4	5	8	0
0	4	3	3	1	2	0
0	5	2	1	5	3	0
0	0	0	0	0	0	0

5×5 input image padded with 0

Each dimension of the output for a convolution layer is calculated as follows:

$$\frac{\text{input size} - \text{kernel size} + 2 \cdot \text{padding size}}{\text{stride}} + 1$$

In our example with stride 1, kernel size 3, and no padding, the output size is $(5-3+2\cdot 0)/1+1=3$. It is good to ensure that the stride always divides the numerator, as otherwise the kernel will be applied asymetrically to the image. Calculating the output dimension of a layer is necessary since the following layer will have its *input* dimension size equal to the previous *output* dimension size.

One last feature that we need to discuss is *channels*. Image data (including ours) typically has three channels, representing the red, green, and blue contents of pixels. Channels act like different "layers" of the image or of the convolutional output. In a convolutional layer, there is one kernel for each pair of input channel and output channel. If we have n input and m output channels, then we will have mn total kernels that are learned individually. Each kernel is applied to its corresponding input channel as described above. Then, each output channel is determined as the sum of the results of the convolutions of all of its kernels, plus a learned bias value.

Convolutional layers are represented in PyTorch with `nn.Conv2d`. The constructor of this class requires three parameters `in_channels`, `out_channels`, and `kernel_size`. It also has optional arguments `stride` (default 1) and `padding` (default 0). Note that for each of the kernel size, stride, and padding, only an integer needs to be specified, and it will be used for both the x and y directions. The following creates a convolutional layer that accepts an image with 3 channels, output 8 channels, and uses a 3×3 kernel with stride 1 and no padding:

```
layer = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3)
```

Pooling layers

Pooling layers are used to reduce the size of the image while retaining important information. The input image is broken into small pieces, called *pools*, each of which is condensed to a single number. The most common form of pooling is *max pooling*, where the output of each pool is just the maximum of its inputs.

4	7	6	2
7	1	2	1
3	3	1	2
2	1	5	3

4×4 Input Image

7	6
3	5

2×2 output after max pooling

Layer type	Number of Parameters
Linear	$(\texttt{in\_features} + 1) * \texttt{out\_features}$
Convolutional	$(\texttt{in\_channels} \cdot \texttt{kernel\_size}^2 + 1) * \texttt{out\_channels}$
Pooling	No parameters
Flatten	No parameters
Activation functions	No parameters

Table 16.3: Parameter counts for layer types we use in this lab

Max pooling has the particularly nice property of making the output remain similar if the input image is shifted slightly.

Max pooling layers are represented in PyTorch as `nn.MaxPool2d`. They accept a single parameter `kernel_size`; this is the size of each of the pools. Using 2×2 pools is the most common. The following creates a pooling layer that uses a 2×2 pool size:

```
layer = nn.MaxPool2d(kernel_size=2)
```

Parameters

When working with neural networks, it can be useful to know how many learnable parameters our model has. This particularly dictates the amount of space needed to store the model. The number of parameters in each layer depends on the type of layer and its input and output sizes. Table 16.3 lists how to calculate this number for the layer types we have discussed.

For example, the example neural network above would have

$$\begin{aligned} &\text{(first linear layer)} && (3 \cdot 32 \cdot 32 + 1) \cdot 100 \\ &\text{(second linear layer)} && + (100 + 1) \cdot 10 = 308310 \text{ parameters,} \end{aligned}$$

and the example convolutional layer would have

$$(3 \cdot 3^2 + 1) \cdot 8 = 224 \text{ parameters.}$$

Problem 3. Create a class for a convolutional neural network that accepts images as $3 \times 32 \times 32$ tensors and returns 1D tensors of length 10, representing its predicted probabilities of each class. Include at least the following:

- Three convolutional layers, each followed by an activation function
- A max pooling layer
- Two linear layers

Be sure that your final activation function is the softmax function. Choose the size of the layers so that your model has at least 50,000 parameters (use Table 16.3), and print out this calculation in the Jupyter notebook file. In practice, specifications of model architecture (i.e. number of layers, layer sizes, etc.) are chosen quite arbitrarily until something works. As such, you may customize your model architecture to your liking, provided your model meets the requirements specified above.

Hint: It can be very helpful to keep track of the size of the image after each step, so you know what the input size should be for the next step. The max pooling layer should occur immediately after a convolutional layer is passed through an activation function. Additionally, you will need a `nn.Flatten` layer after the convolutional layers and before the linear layers. You can check that your model works correctly by passing an (unsqueezed) image through it as demonstrated in the following code:

```
>>> model = NNExample()
>>> model(images[0].unsqueeze(0))
tensor([[0.0952, 0.1120, 0.1019, 0.0992, 0.0955, 0.0984, 0.0886, 0.1326, ↩
    0.0966, 0.0800]], grad_fn=<SoftmaxBackward0>)
```

Note that your neural network will predict different probabilities for each of the categories.

Training the Model

Now that we have data and a model all set up, we need to train the model on the data. We do this by iterating through the DataLoader, calling the model on the data, determining how well the model classified the data, and then optimizing the model weights. We use a loss function, called the *objective*, to calculate the loss, which is the difference between the model's predicted labels and the actual labels of the data. A common classification loss function is Cross Entropy Loss

$$L_{CE} = -\sum_i a_i log(p_i),$$

where i represents each data point, p_i is the Softmax probability for each data point, and a_i is the label for each data point. PyTorch's `nn.CrossEntropyLoss()` conveniently handles all of this.

Once the loss is calculated by the objective, we can use it to optimize the model weights to make the loss smaller. This can be done through *backpropagation*, which calculates the partial derivatives of the loss function with respect to each weight, and then uses gradient descent to update every weight. PyTorch has several predefined methods for optimization, but we'll use the popular `Adam` algorithm. PyTorch accumulates gradients when backpropagating, which is sometimes desireable, but in our case it would cause the loss to increase. To prevent this, we need to zero out the gradients before we perform each backpropagation. PyTorch streamlines this entire training sequence in a very clean way, as shown in the following:

```
>>> import torch.optim as optim

# Define the objective and optimizer
>>> objective = nn.CrossEntropyLoss()
>>> optimizer = optim.Adam(model.parameters(), lr=1e-4)


# For each iteration of the DataLoader, do the following
>>> optimizer.zero_grad()             # Zero out the gradients
>>> y_hat = model(x)                  # Predict labels
>>> loss = objective(y_hat, y_truth)  # Calculate loss
>>> loss.backward()                   # Backpropagate to compute gradients
>>> optimizer.step()                  # Optimize and update the weights
```

An *epoch* is a complete training sequence that trains over the entire DataLoader. To improve the model's accuracy, we can train over many epochs. A good guideline is to train the model for the number of epochs it takes for the loss to stop decreasing.

The loss is calculated using the training data, but at the end of each epoch we also want to know how well the model performs with the validation data. Before we determine the validation accuracy, we need to switch our model to evaluation mode so it doesn't continue training. This is done by the simple command `model.eval()`, but it's important to switch the model back to training mode at the start of each epoch using `model.train()` The validation accuracy is determined by simply iterating through the validation DataLoader, and seeing if the model can correctly predict each data point. The validation accuracy is then computed by dividing the number of correct predictions by the total number of data points in the validation DataLoader.

```
>>> model.eval()                        # switch to evaluation mode
>>> validation_score = 0
>>> for x, y_truth in validation_loader:
>>>     x, y_truth = x.to(device), y_truth.to(device)
>>>     y_hat = model(x)
>>>     if y_truth == y_hat.argmax(1):  # compare with greatest probability
>>>         validation_score += 1
>>> validation_accuracy = validation_score / len(validation_loader)
```

The validation accuracy does not determine the model's final accuracy. The final accuracy is computed in the same way as the validation accuracy, but this time using the testing data, and it's only computed one time, when the model finishes training entirely.

TQDM is a python package that displays the progress of a for-loop, which can help estimate the remaining time. TQDM is initialized outside the loop, then updated inside the loop, as follows:

```
>>> from tqdm import tqdm

>>> loop = tqdm(total=len(train_loader), position=0)

>>> for epoch in range(num_epochs):
>>>     loop.set_description('epoch:{}, loss:{:.4f}'.format(epoch,loss.item()))
>>>     loop.update()

>>> loop.close()

epoch:1 loss:1.8585: : 1402it [00:17, 79.69it/s]
```

Problem 4. Send your model to the device and instantiate the objective and optimizer. Train your model with a TQDM display, and calculate the Validation Accuracy after each epoch. Begin by initializing your TQDM loop, then for each epoch, do the following:

1. Set your model to training mode (`model.train()`)

2. Instantiate an empty `loss_list`
3. For each batch in `train_loader`:
   (a) Send `x` and `y_truth` to device
   (b) Zero out the gradients
   (c) Use model to predict labels of `x`
   (d) Calculate loss between predicted labels and `y_truth`
   (e) Append loss (`loss.item()`) to `loss_list` (the `.item()` feature extracts the element from a tensor with only one element)
   (f) Update TQDM loop
   (g) Backpropagate to compute gradients
   (h) Optimize and update the weights
4. Save the loss mean as the mean of the losses in `loss_list`
5. Set your model to evaluation mode (`model.eval()`)
6. Calculate and save validation accuracy

Finish the training by closing your TQDM loop. Train for 10 epochs, saving the mean loss and validation accuracy for each epoch. Plot the mean losses and validation accuracies, which should resemble Figure 16.2. Lastly, print the final test score using the testing data, as described above.

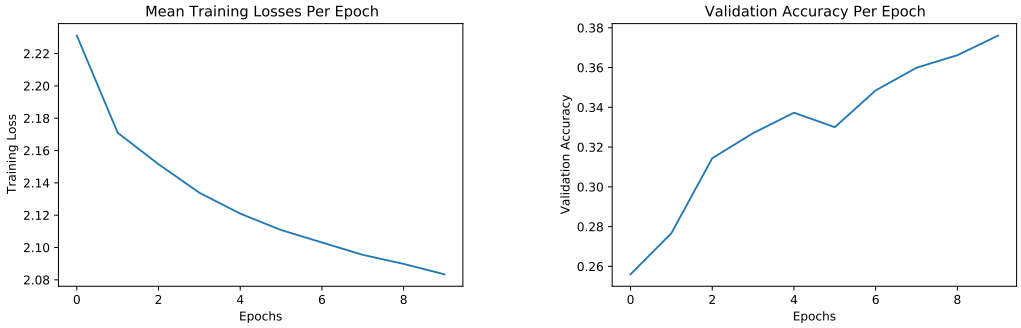


Figure 16.2: Training Loss and Validation Accuracy for a CNN on CIFAR10.

Adversarial Attacks

Just like any algorithm or software, deep learning is susceptible to attacks. For deep learning models, this vulnerability most often manifests in the model being extremely sensitive to certain types of changes in the input that really should not matter. This results in the model giving nonsensical results, which, while amusing, can cause major problems. Examples of adversarial attacks against neural networks range from adding a small amount of noise to a picture of a panda, resulting in the model classifying the image as a gibbon with 99% confidence [GSS15] to fooling facial recognition by printing a pair of eyeglasses [GKB17]. When designing machine learning models, it is important to be aware of these issues so that their impact can be mitigated.

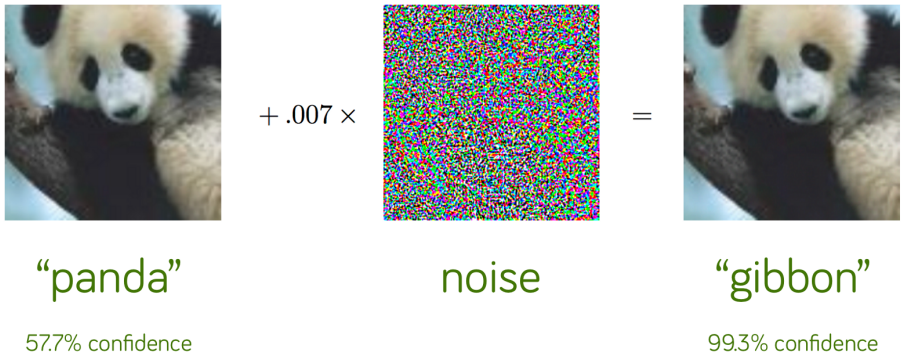


Figure 16.3: A slight modification to a correctly-classified image of a panda results in the model confidently classifying it as a gibbon, despite the image not having changed in any substantial way.

The example of modifying the image of a panda is an attack called the *Fast Gradient Sign Method (FGSM)*. FGSM is a *white-box attack*, meaning that the attacker has access to the model; this is in contrast with a *black-box attack* where the attacker only has access to the model's inputs and outputs.

During model training, gradients are used to adjust the model weights so that loss is minimized. In FGSM, the gradient is instead used to perturb the input image in a direction that *maximizes* the loss, using the following equation:

$$x_{\text{perturbed}} = x + \varepsilon \,\text{Sign}(\nabla_x \text{Loss}(\theta, x, y))$$

where x is the input, y is the label, and θ is the model parameters.

We can calculate this perturbation in PyTorch as follows. To calculate the gradient of the model with respect to a piece of data `x`, we first need to set `x.requires_grad = True` and call `x.retain_grad()`. Then, we zero out the optimizer's gradient, run `x` through the model, and compute the loss, similar to training. After this, the gradient of the output with respect to `x` can be obtained using the attribute `x.grad.data`.

The following function `fgsm_attack` accepts a model and an image and performs the FGSM attack, returning the perturbed image:

```
def fgsm_attack(model, optimizer, objective, x, y, eps):
    """
```

```
    Performs the FGSM attack on the given model and data point x with label y.
    Returns the perturbed data point.
    """
    # Calculate the gradient
    x.requires_grad = True
    x.retain_grad()
    optimizer.zero_grad()
    output = model(x)
    loss = objective(output, y)
    loss.backward()
    data_grad = x.grad.data
    # Perturb the images
    x_perturbed = x + eps * data_grad.sign()

    return x_perturbed
```

We will use this function to explore this type of adversarial attack on our neural network.

Problem 5. Write a function that loops through the test data using the function `fgsm_attack` to perturb the images and using your trained model from Problem 4.

Run your function for each epsilon in `[0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]`, and plot epsilon against the model's accuracy.

Display the perturbed version of the first image in the test data for each epsilon, using the following code. Be sure to show the old and new labels for each perturbed image. Make sure the original image is classified correctly. Your figure should look similar to Figure 16.4.

```
# Move the image to cpu and convert to numpy array
>>> ex = perturbed_data.squeeze().detach().cpu().numpy()

# Plot the image
>>> img = ex / 2 + 0.5     # unnormalize
>>> plt.imshow(np.transpose(img, (1, 2, 0)))
```

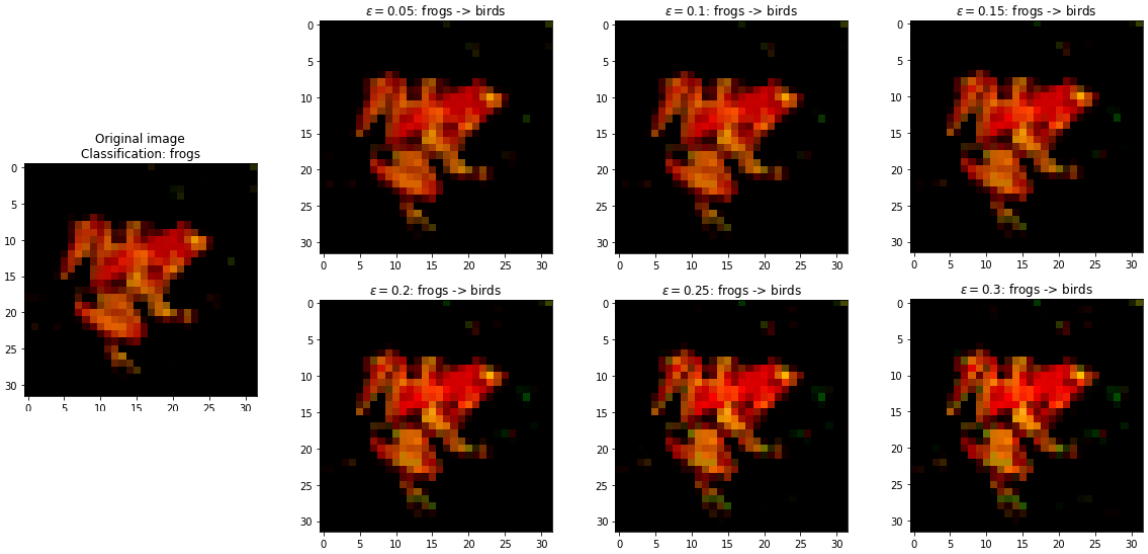


Figure 16.4: The first modified image for different values of epsilon.

Additional Materials

TensorBoard

TensorBoard is a visualization toolkit for neural networks. It was originally built for Tensorflow, but also can be used with PyTorch. The main features of TensorBoard include model visualization, dimenionality reduction, tracking and visualizing metrics, and displaying data.

To create a tensorboard, run the following code:

```
>>> %load_ext tensorboard
>>> logs_base_dir = "runs"
>>> os.makedirs(logs_base_dir, exist_ok=True)
>>> %tensorboard --logdir {logs_base_dir}
```

The TensorBoard homepage will show up inline:

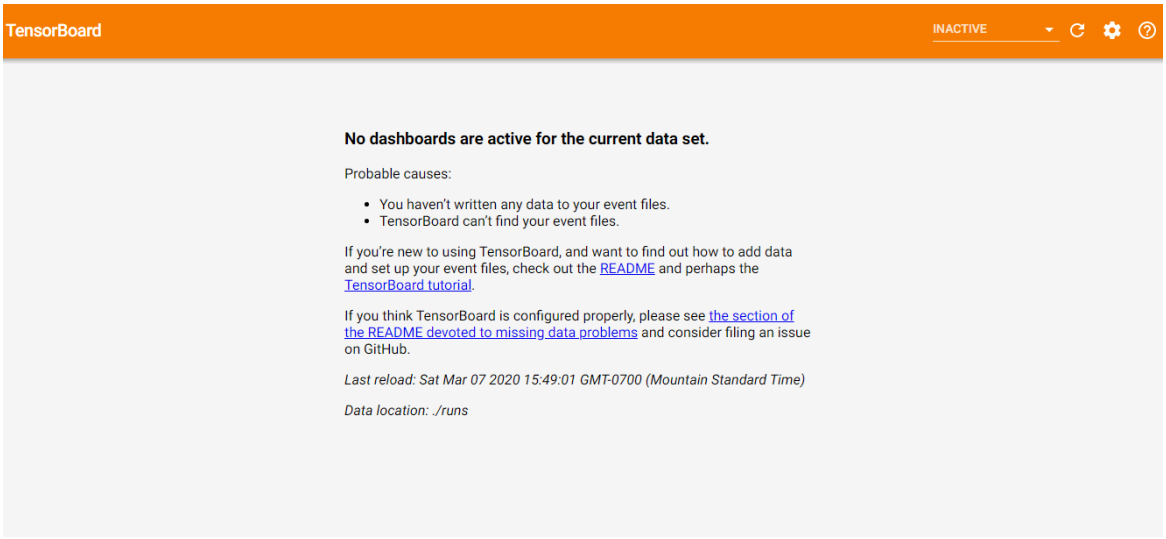


Figure 16.5: The home page of an empty TensorBoard.

We write to TensorBoard using `SummaryWriter`. It writes to files in the `logs_base_dir` that are used by TensorBoard to display information. You can view the `logs_base_dir` directory by selecting the file icon on the far left of the page. For example, we can create an interactive graph of our model.

```
>>> tb = SummaryWriter()
>>> tb.add_images("Image", images)
>>> tb.add_graph(model, images)
>>> tb.close()
```

This updates our TensorBoard with a `GRAPHS` tab, which describes the model. If it doesn't show up automatically, press the refresh button in the top right corner of the TensorBoard. You can explore the model by clicking on the components.

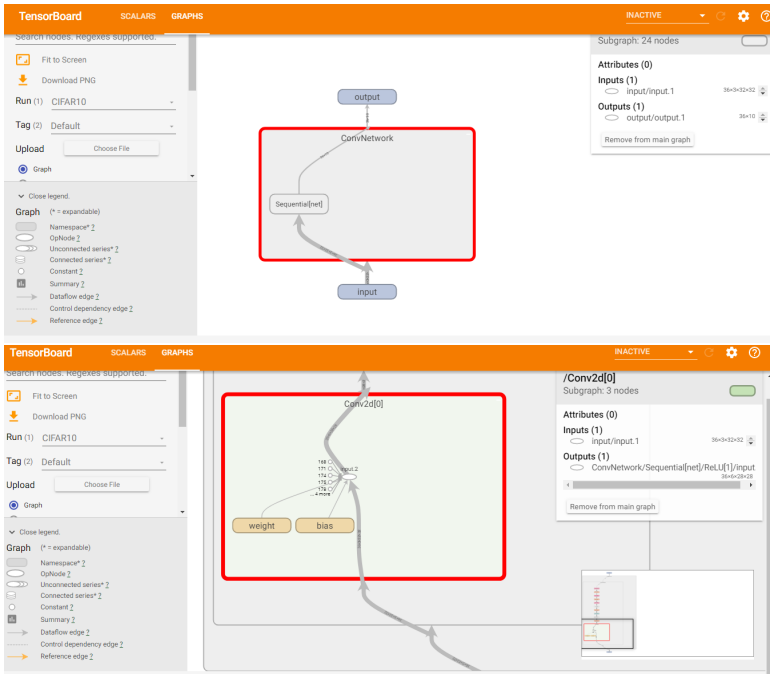


Figure 16.6: Examples of TensorBoard Graph Tab.

The following items can be added to TensorBoard, with more information at `https://pytorch.org/docs/stable/tensorboard.html`.

- `add_scalar/s`
- `add_image/s`
- `add_figure`
- `add_text`
- `add_graph`
- `add_hparams`

To save the training loss, write a function that returns a matplotlib figure of the training loss plot. Then use `tb.add_figure(figure_name, plot_loss())`.

```
writer.add_figure('Training Loss',plot_loss())
```

Problem 6. Create a TensorBoard for this project that includes the network, a plot of iterations versus training loss and a plot of iterations versus test accuracy from the training done in Problem 4.

17 Recurrent Neural Networks

Lab Objective: *Recurrent Neural Networks are powerful machine learning algorithms that accept sequences as inputs and can process temporal data. In this lab, we generate a Mozart-like piano sonata using the Long Short-term Memory RNN.*

Recurrent Neural Networks

Convolutional Neural Networks work well for problems like image classification where the inputs and outputs are independent and of fixed size. However, many problems do not have these constraints. For example, what if we want to predict the next word in a sentence? This is clearly not independent since the output for one iteration becomes the input for the next iteration. *Recurrent Neural Networks* (RNNs) address these issues by using sequences as the input, output, or both, allowing for temporal dynamic behavior. They perform the same task for every element of the sequence, hence their recurrent nature. Each task uses the input as well as recent previous information, called memory, from the network to create the output. Even if the input is not sequential, it is possible to process it sequentially using RNNs, resulting in powerful learning algorithms.

Data

For this lab, we will use Google Colab and its GPU capability. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a pop-up called *Notebook Settings*. Under *Hardware Settings*, select *GPU*. We recommend mounting a Google Drive to the notebook to make loading and saving data easier. This will save the data if the notebook is disconnected; if the data is saved to the Colab directory, the entire project must be rerun. To mount a Google Drive, run

```
>>> from google.colab import drive
>>> drive.mount('/content/drive')
```

Follow the instructions in the cell to authorize the account.

If you need to refresh your drive connection, you can run

```
>>> drive.mount('/content/drive', force_remount = True).
```

Download Data

We will be using a collection of Mozart piano sonatas as the data to train on. For easy download, run the following function to save the files to `filepath` in the Google Drive folder.

```
def download_data(filepath):
    if not os.path.exists(os.path.join(filepath, 'mozart_sonatas.tar.gz')):
      datasets.utils.download_url('https://github.com/Foundations-of-Applied-↩
          Mathematics/Data/raw/master/Volume3/mozart_sonatas.tar.gz', filepath,↩
           'mozart_sonatas.tar.gz', None)

      print('Extracting {}'.format('mozart_sonatas.tar.gz'))
      gzip_path = os.path.join(filepath, 'mozart_sonatas.tar.gz')
      with open(gzip_path.replace('.gz', ''), 'wb') as out_f, gzip.GzipFile(↩
          gzip_path) as zip_f:
        out_f.write(zip_f.read())

      print('Untarring {}'.format('mozart_sonatas.tar'))
      tar_path = os.path.join(filepath,'mozart_sonatas.tar')
      z = tarfile.TarFile(tar_path)
      z.extractall(tar_path.replace('.tar', ''))

>>> download_data('drive/MyDrive/Colab')
Downloading https://raw.githubusercontent.com/Foundations-of-Applied-↩
    Mathematics/Data/master/RNN/mozart_sonatas.tar.gz to drive/MyDrive/Colab/↩
    mozart_sonatas.tar.gz

Extracting mozart_sonatas.tar.gz
Untarring mozart_sonatas.tar
```

Parsing the Data

Music21 is a musical toolkit for Python developed by MIT.[1] It can read and write music files with the .mid extension, which are MIDI files, standing for Musical Instrument Digital Interface files. Midi files contain information on music, like which notes are played, how loud each note is, and for how long each note is held.

There are two important object types: Notes and Chords. A Note object is comprised of three attributes. The `pitch` and `octave` give information about the frequency of the Note. There are seven pitches: A, B, C, D, E, F, and G. These pitches repeat, doubling the frequency of the vibration of the previous matching pitch. The interval over which the frequency of a note is doubled is called an octave. A piano has seven octaves, and the middle of the keyboard is called `middle c`. In Music21, it is represented by C4, where 4 is the octave. Lastly, the `offset` is the temporal location of the Note in the file. Chord objects contain multiple Note objects that are played at the same time.

```
from music21 import converter, instrument, note, chord,  stream
```

[1] `https://web.mit.edu/music21/doc/index.html`.

```
# Read the file piano_sonota_279.mid
midi = converter.parse('piano_sonata_279.mid')
notes_to_parse = instrument.partitionByInstrument(midi).parts.stream().recurse↩
    ()

# Display the Note and Chord objects, their pitches and offsets
for element in notes_to_parse:
    if isinstance(element, note.Note):
        print(element, element.pitch, element.offset)
    elif isinstance(element, chord.Chord):
        print(element, element.pitches, element.offset)
```

```
<music21.note.Note E> E5 803.0
<music21.note.Note F> F5 803.5
<music21.chord.Chord B3 B2> (<music21.pitch.Pitch B3>, <music21.pitch.Pitch B2↩
    >) 803.5
<music21.note.Note G> G5 804.0
<music21.note.Note F> F5 804.5
<music21.chord.Chord C4 C3> (<music21.pitch.Pitch C4>, <music21.pitch.Pitch C3↩
    >) 804.5
<music21.note.Note E-> E-5 805.0
<music21.note.Note D> D5 805.5
<music21.chord.Chord E-3 E-4> (<music21.pitch.Pitch E-3>, <music21.pitch.Pitch ↩
    E-4>) 805.5
```

```
# Helper function to parse through a Chord object
def order_pitches(pitches):
    """ pitches: element.pitches object where element is a chord.Chord
        returns: sorted list of strings for each pitch in the chord
    """
    return sorted(list(set([str(n) for n in pitches])))
```

Problem 1. Download the data. Write a function that accepts the path to the .mid files, parses the files, and returns a list of the 114215 Notes and Chords as strings. There are many element types in MIDI files, so be sure to only look for Notes and Chords. For the Chords, join the pitches of the Notes in the Chords with a `.` as in (`'D3.D2'`).

Print the length of your list and the number of unique Notes and Chords.

```
# Example of a part of the list
['A5', 'C6', 'G3.C4', 'A5', 'B-5', 'A5', 'G5', 'D3.D2']
```

Hint: An easy way to get the list of mozart sonata file names is with the following code.

```
import glob
```

```
>>> glob.glob(filepath + "/mozart_sonatas/mozart_sonatas/*.mid")
```

Also, you'll want to wrap `element.pitch` with `str()` to convert it into a string. Furthermore, the `.join()` method may be useful when constructing the Chord strings.

For the remainder of this lab, we will refer to the notes and chords in the list created in Problem 1 simply as pitches. In order for this data to be applied to an RNN, we need to create sequences. We do this by looping through the list of pitches and slicing it into lists of a given length. The label for each sequence, or the correct pitch we want the RNN to predict, is the element immediately following the sequence. So for a sequence length of 10, given elements 1 through 9 as a sequence, element 10 would be the label.

Since RNNs only accept numbers, we need to convert the pitches to integers. Using the sample list in Problem 1 as an example, we would map `'A5'` to 0, `'C6'` to 1, `'G3.C4'` to 2, `'A5'` to 0 again, and so on. The PyTorch DataLoader accepts a list of lists, where each element is of the form `[sequence, label]`, where the sequence is a PyTorch Long tensor, and the label is an integer. So in our case, the sequence will be a tensor of integers representing pitches while the label will be the integer representing the first pitch that follows the sequence.

```
# Example Data
example_data = [169, 269, 165, 187,  24, 366, 353, 269, 260, 233, 223, 169,
        162, 366, 353, 269, 260, 233, 223, 169, 162,  24,   8, 269, 260,  91]

# Create sequences as Long Tensors
first_sequence = torch.LongTensor(example_data[0:10])
second_sequence = torch.LongTensor(example_data[1:11])
first_label = example_data[10]
second_label = example_data[11]

# Example of data points formatted for the DataLoader, [sequence, label]
>>> [first_sequence, first_label]
[tensor([169, 269, 165, 187,  24, 366, 353, 269, 260, 233]), 223]

>>> [second_sequence, second_label]
[tensor([269, 165, 187,  24, 366, 353, 269, 260, 233, 223]), 169]
```

Problem 2. Using the list returned in Problem 1, create the training, validation, and testing DataLoaders. Make sure to do all the following steps:

- Convert the pitches to integers.
- Split the data into Long tensors of length 10.
- Create the labels.
- Randomly split the data into training, validation, and test sets using a 70/15/15 split (use `torch.utils.data.random(data,lengths)` where `lengths=[0.7, 0.15, 0.15]`).

- Create the DataLoaders for these sets of data, using `batch_size=128` for the training data and `batch_size=32` for the validation and test data; also, set `shuffle=True` for the training data and `False` for the validation and test data (this is common practice in Deep Learning).

Print the length of each DataLoader (they should be 624, 536, and 536, respectively).

Hint: To keep all batches the same size, drop the last training batch in the DataLoader with the parameter `drop_last=True`.

LSTM

While RNNs have the ability to look at short-term history, like the previous word in a sentence, they lack longer term contexts. For example, predicting the last word in the "The boat is in the *water*" is relatively easy. Consider the following two sentences separated by some other text: "I grew up in France ... I speak fluent *French*." It's clear that the last word will be a language, but we need the previous information of France to correctly identify which language. RNNs can't remember this information due to exploding and vanishing gradients.

Long Short-Term Memory (LSTM) networks are a popular RNN variation capable of long-term memory that solve this problem. They are used extensively in speech recognition, machine translation, and text-to-speech programs. Every step in the LSTM has three inputs: the current input, the short-term memory (hidden state) from the previous input, and the long-term memory (cell state). There are three gates that regulate these three types of memory. The *Input Gate* decides what information will be added to the long-term memory, the *Forget Gate* chooses which information should be kept in the long-term memory, and the *Output Gate* creates the new short-term memory.

Defining the Network Layers

In PyTorch, the memory is a tuple (hidden state, cell state) and must be initialized before the LSTM layer is called. Usually, the hidden state initialization function is defined in the network class and is called during the training loop for each batch. The LSTM layer can be stacked, with the input from one layer going directly to the next layer; `num_layers` is how many stacked LSTM layers there are in the model. The `hidden_size` is the number of features in the hidden layer. This can be any size, but for this lab we will use 256.

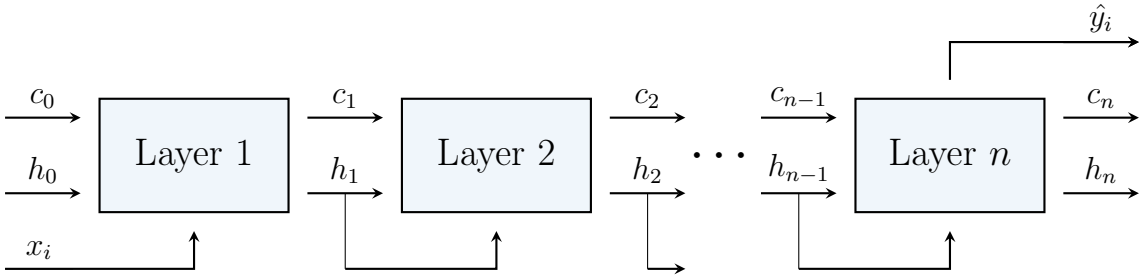


Figure 17.1: PyTorch implementation of an LSTM. The LSTM takes as input two initial memory states (hidden, cell) as well as the ith datapoint x_i from a batch, and outputs the updated memory states and predicted output $\hat{y}_i$. The input for each stacked layer is the hidden state from the previous layer, and n is equal to `num_layers`. Note that the PyTorch LSTM runs all datapoints in a batch in parallel to maximize efficiency.

```
class RNN(nn.Module):
    """ Recurrent Neural Network Class """

    def __init__(self):
        super(RNN, self).__init__()

    # Define function to initialize hidden states
      def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        h0 = weight.new(self.num_layers, batch_size, self.hidden_size).zero_().↩
            to(device)
        h1 = weight.new(self.num_layers, batch_size, self.hidden_size).zero_().↩
            to(device)
        return (h0, h1)
```

Before calling the LSTM layer, we will use an embedding layer to store the words. The embedding layer is a lookup table that takes in indices and outputs the word embeddings. This is PyTorch's method of one-hot encoding, a process in which variables are converted to binary for better predictions. The first parameter is the number of words in the dictionary; in our case, there are around 668 possible notes and chords. The second parameter is the embedding dimension. 32 and 64 are good choices for the embedding dimension.

The LSTM layer has 5 parameters. The first three have already been discussed. The parameter `batch_first` is a boolean that indicates if the batch size is the first or the second dimension in the input tensor. Since we are using the DataLoader, the batch size will be the first dimension and `batch_first=True`. If the last parameter, `dropout`, is defined, a Dropout layer is added after each LSTM layer, except the last. During a Dropout layer, elements of the input tensor are randomly zeroed out with probability p, and the output is scaled. This is sometimes used for regularization to improve the network. However, we will *NOT* add a Dropout layer to our model, because we will instead use a BatchNorm1d layer. BatchNorm1d layers normalize the input and have as parameters the number of features of the input. Thus, if we were to use both Dropout to BatchNorm1d layers, the input would be normalized over fewer nodes than the input actually contains, which would throw off the scaling of the model.

The last layer should be a softmax activation function. Softmax rescales a tensor to $[0, 1]$ with the sum of all elements equal to 1. Thus the output of Softmax can be thought of as a probability vector. Notice that all of the layers: Embedding, LSTM, Linear, BatchNorm1d, and LogSoftmax are initialized in the `__init__()` function.

```
  class RNN(nn.Module):
      """ Example class for LSTM model """

      def __init__(self, n_notes, embedding_dim):
          super(RNN, self).__init__()

          self.hidden_size = 256
          self.num_layers = 3      # number of layers in the LSTM
          self.n_notes = n_notes   # number of unique pitches
          self.embedding = nn.Embedding(n_notes, embedding_dim)
          self.lstm = nn.LSTM(embedding_dim, self.hidden_size,
```

```
                                self.num_layers, batch_first=True)
        self.batch1 = nn.BatchNorm1d(self.hidden_size)
        self.linear = nn.Linear(self.hidden_size, self.n_notes)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x, hidden):
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)
        out = self.batch1(lstm_out[:,-1])
        # Output from final step is passed forward
        return self.softmax(self.linear(out)), hidden
```

During training, when the model is called, the input is embedded and then passed to the LSTM layer with the hidden states. The hidden state output is saved for the next batch while the LSTM output from the final step is sent through the rest of the model. To prevent differentiating the hidden states, we must call the `detach()` method before taking a backwards step. This disables automatic differentiation on the hidden states during training. Because we don't do a backwards step during testing, we don't need to worry about detaching the hidden states during testing.

```
# Initialize the model
model = RNN()

# Hidden state training demonstration
for epoch in range(30):
    for x_truth, y_truth in train_loader:

        # Initialize the hidden states
        (h0, h1) = model.init_hidden(train_batch_size)

        # Pass data through the model to get output and new hidden states
        output, (h0, h1) = model(x_truth, (h0, h1))

        # Disable automatic differentiation on the hidden states
        h0 = h0.detach()
        h1 = h1.detach()
```

A Faster Way to Calculate Validation Accuracy

We would like to periodically check our model's validation accuracy as our model is training, but this takes time. One way to save time is to only calculate validation accuracy every n epochs. Another way is to increase the batch size of the validation DataLoader; this is the method we use in this lab, which is why we set the validation and test DataLoader batch sizes to 32 in Problem 2. The problem is, if we compare the predicted labels of a large batch to their true values at the same time, the probability that every data point is correct is small, so the validation accuracy will be mostly 0. The way to work around this is to compare the predicted labels of each batch with their true values seperately, not together. An example of how this might be done is demonstrated in the following:

```
validation = 0
```

```
model.eval()
for x_truth, y_truth in validation_loader:
    x_truth, y_truth = x_truth.to(device), y_truth.to(device)
    (h0, h1) = model.init_hidden(val_batch_size)
    y_hat, _ = model(x_truth, (h0, h1))

    # sum how many elements equal each other between the true and predicted
    # batches, then divide by the number of elements in the batch
    validation += sum(torch.eq(y_truth, y_hat.argmax(1))) / val_batch_size

mean_validation_accuracy = validation.item() / len(validation_loader)
```

Problem 3. Create an LSTM network class. Have a hidden layer size of 256, and include at least 3 LSTM layers. Also have at least 2 Linear layers. The last LSTM layer and each of the Linear layers should be followed by a BatchNorm1d layer, for at least 3 total BatchNorm layers. The final layer should be a Softmax activation.

Initialize the model. Define the loss as CrossEntropyLoss, and define the optimizer as RMSprop.

```
optimizer = torch.optim.RMSprop(model.parameters(),lr=.001)
```

Train the model for 30 epochs. Make sure to reinitialize the hidden states `(h0, h1)` for each training batch. After taking a backwards step during training, scale the gradients using

```
nn.utils.clip_grad_norm_(model.parameters(), 5)
```

This will ensure that the gradients are reasonably sized so that the model can learn.

At the end of every epoch, calculate the validation accuracy and mean loss on the validation data. Remember to change the model to `eval()` mode when running the validation data and then `train()` when running on the training data. The hidden states `(h0, h1)` will also need to be reinitialized for each validation batch.

Once the training is complete, plot the training and validation losses versus epochs on the same plot. Also, plot the validation accuracy versus epochs. Then, print the final test accuracy by running the finished model on the test data.

Hint: While training this model for 30 epochs on a GPU should take less than 5 minutes, you may want to test your code by only training it for 2 epochs, and then when everything works the way it should, train it for the whole 30 epochs. After 2 epochs, your model should have a validation accuracy of $10 - 20\%$.

Achtung!

Colab has a 12 hour limit on the amount of GPU available, and the longer one runs, the less priority it has. If you follow the instructions given in this lab correctly, training should take less than 5 minutes. Nevertheless, you may still wish to save the training progress of your model's weights after each epoch. If you wish to do so, you may include this block of code in your for loop.

```
torch.save({
  'epoch': epoch_number,
  'model_state_dict': model.state_dict(),
  'optimizer_state_dict': optimizer.state_dict(),
  'loss': loss,},filename)
```

Then, if at any point the notebook has disconnected, all you need to do is reinitialize the model, loss, and optimizer, and then run this function to load the saved model.

```
def load_model(filename):
    """ Load a saved model to continue training or evaluate """
    device = torch.device("cuda:0" if torch.cuda.is_available() ↩
        else "cpu")

    # n_notes is the number of unique pitches
    model = RNN(n_notes, embedding_dim)
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.RMSprop(model.parameters(),lr=.001)

    checkpoint = torch.load(filename,map_location=torch.device('cpu↩
        '))
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    last_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    model.eval() # Toggle evaluation mode

    return model, criterion, optimizer
```

Generating Music

Now that we have trained the model, we can create our own piano sonata excerpt by predicting a new sequence of notes. We will start with an initial sequence of notes, predict what note should follow, and then shift the sequence to include this new note in order to predict the next one, and we'll repeate this process for as long as we like.

Specifically, first select a random sequence from the test data, and initialize an empty list of predictions. Then, for each note we wish to predict, initialize the hidden states using `model.init_hidden(batch_size)`, and then get a prediction by inputting the sequence and these hidden states into the model, just as we did in the training step. The argmax of this prediction (`prediction.argmax().item()`) is an integer representing a pitch. Append this integer to our list of predictions, and then update the sequence by appending this integer to it, and then by dropping the first entry of the sequence. Repeat this process for each note we wish to predict. The list of predictions (which is a list of integers) can then be converted into pitches using the same method we used to initially convert the pitches into integers.

Problem 4. Write a function that randomly chooses a sequence in the test data (which has length 10) and predicts the next n elements, defaulting to 500. Convert the predicted elements to pitches, and return them as a list of length n. It should look similar to

```
['D4', 'C#4', 'F#5', 'G5', 'A5', 'C6', 'G3.C4', 'B-5', 'A5', 'G5', 'A5']
```

Now we need to convert our list of pitches into Music21 Notes and Chords objects. For each element in the list of pitches, we first determine if it's a note or a chord by the presence of a . (period) in the string. Music21 Note objects are created using the pitch and instrument type. If the element is a chord, we can create a Muisc21 Chord object by first creating a list of Note objects for each note in the chord.

Music21 Note and Chord objects must also have a specified *offset*. The offset indicates at what timestep each object is to be played. The first object will have an offset of 0, and the offset will increment for each following object. The simplest way to choose each offset is look at the distribution of offsets in the original dataset and choose a set amount to increment the offset each time. Since the most common offset (0.0) results in notes being played at the same time, we'll ignore it and choose to increment the offset by either 0.25 or 0.5. For a more advanced option, you could randomly generate which offset to use based on a probability distribution that reflects the following:

0.0	1999
0.25	1167
0.5	507

Table 17.1: The three most common offset distance and their frequency in the Mozart data.

In summary, while looping through our predicted pitches, if we should come accross a Chord, the code to create a Music21 Chord object would look like the following:

```
notes = [ ]

# Create Note objects for each note in the chord
for pitch in chord_pitches:
    new_note = note.Note(pitch)
    # Specify Piano as the instrument type
    new_note.storedInstrument = instrument.Piano()
    notes.append(new_note)
```

```
# Create a Chord object using list of Note objects
new_chord = chord.Chord(notes)

# Specify offset for this object
new_chord.offset = offset
```

Finally, we write the list of Music21 objects to a midi file and save it.

```
midi_stream = stream.Stream(output_notes)
midi_stream.write('midi', fp=file_location)
```

You can embed and play the file in your notebook using the following code, which first converts the `.midi` file into a `.wav` file.

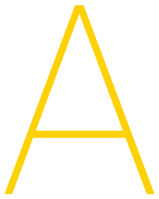
```
!apt install fluidsynth
!cp /usr/share/sounds/sf2/FluidR3_GM.sf2 ./font.sf2
!pip install midi2audio
from midi2audio import FluidSynth
from IPython.display import Audio

FluidSynth("font.sf2").midi_to_audio("file_location", "new_file_location.wav")
Audio("new_file_location.wav")
```

Problem 5. Convert the predictions from Problem 4 into Music21 Note and Chord objects and save it as `'mozart.mid'`. Embed your music file into the notebook.

Part II

Appendices

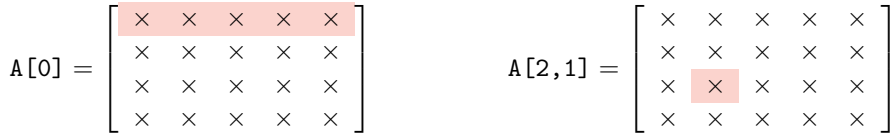


NumPy Visual Guide

Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

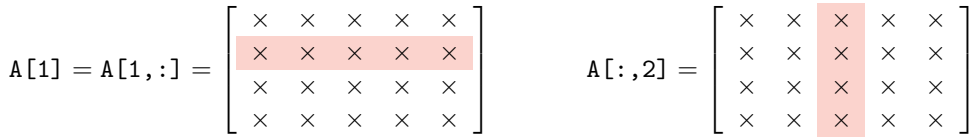
Data Access

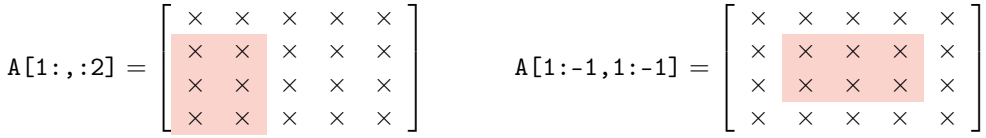
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.



Slicing

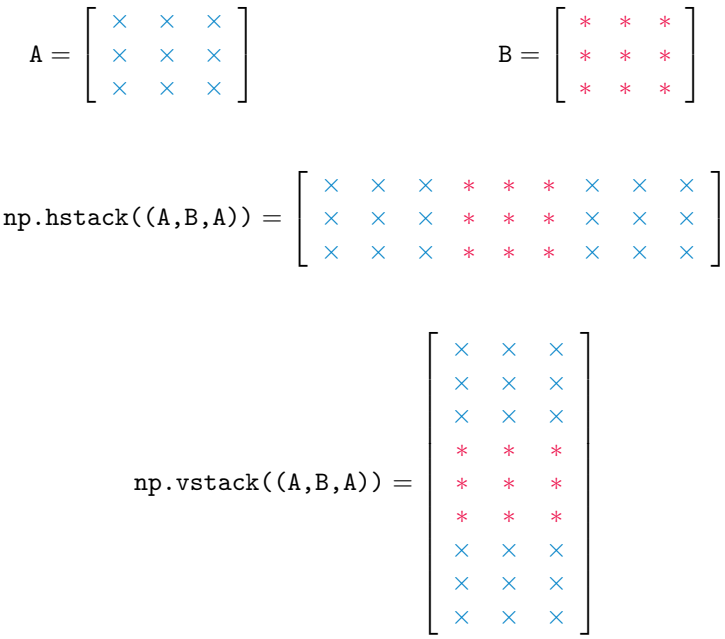
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



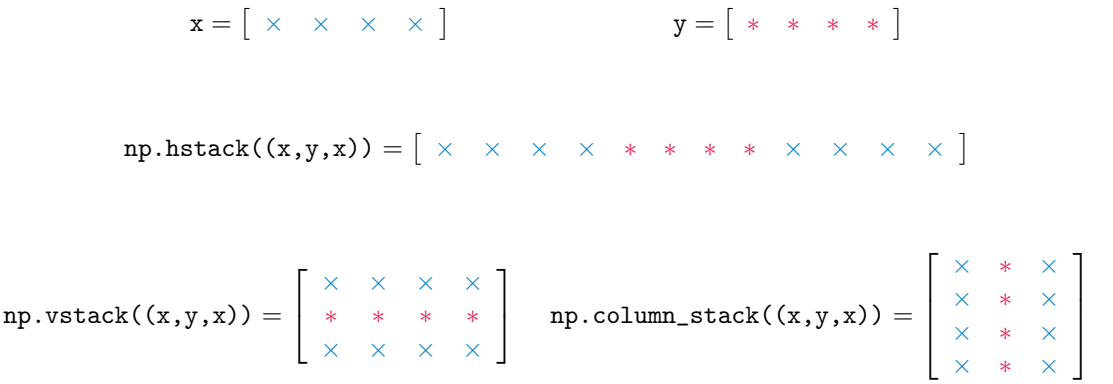


Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.



Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.



The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

B Matplotlib Syntax and Customization Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

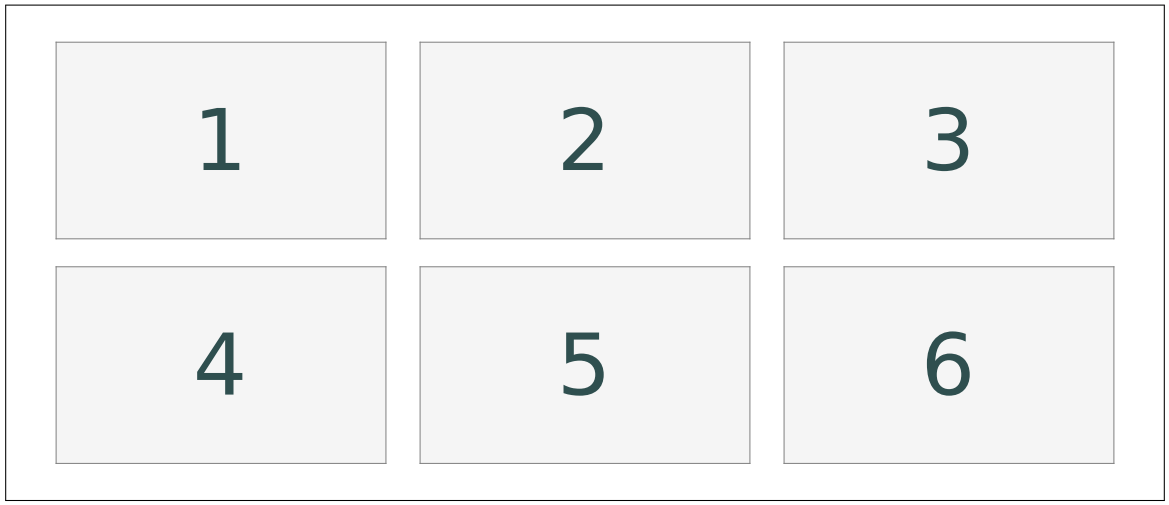


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Achtung!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

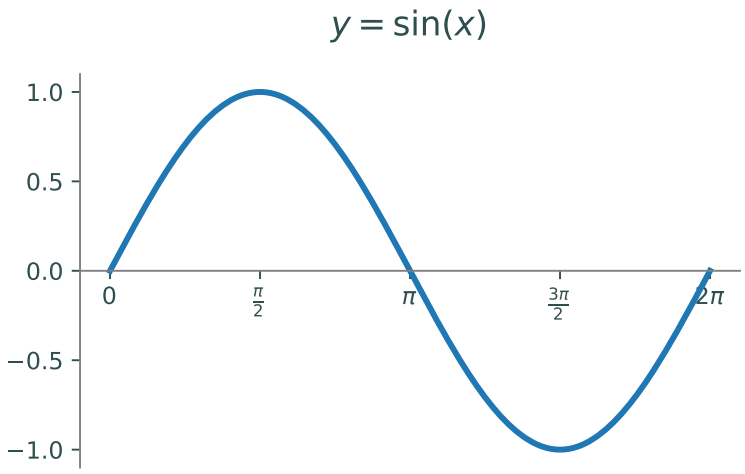


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

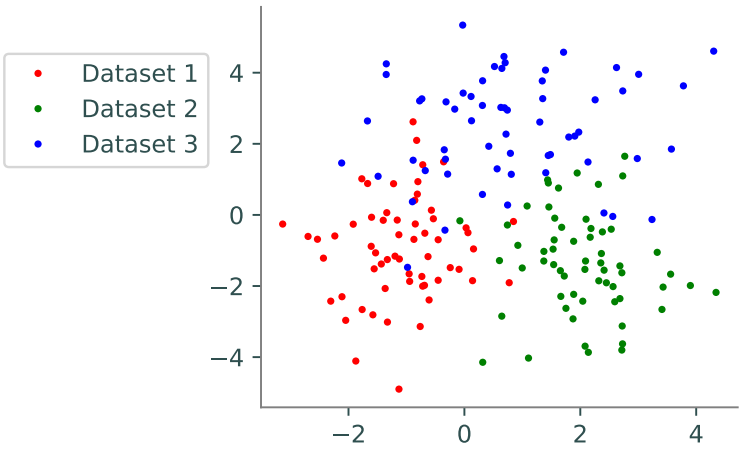


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

Code	Color
`'y'`	yellow
`'k'`	black
`'w'`	white
`'C0'` - `'C9'`	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate('text',(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc='best')
`text()`	Add text at a given position on the plot	text(x,y,'text')
`title()`	Add a title to the plot	title('text')
`suptitle()`	Add a title to the figure	suptitle('text')
`xlabel()`	Add a label to the x-axis	xlabel('text')
`ylabel()`	Add a label to the y-axis	ylabel('text')

Table B.4: Text and annotation functions in Matplotlib

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

[1] See `http://www.latex-project.org/` for more information.

```
# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

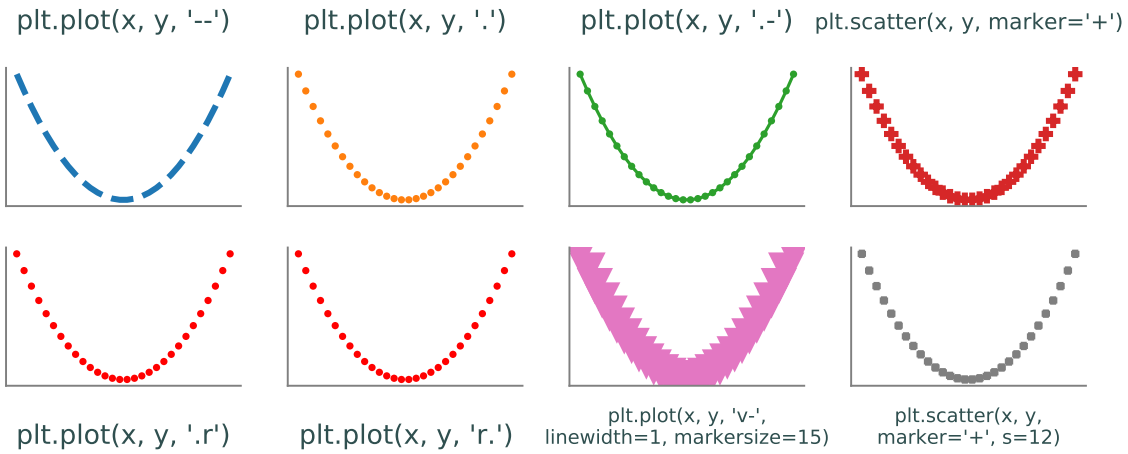


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
```

```
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

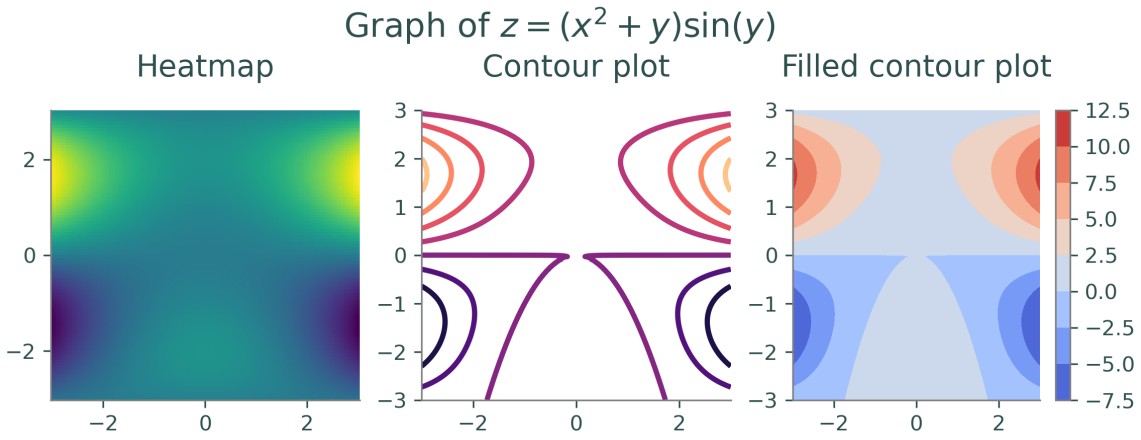


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

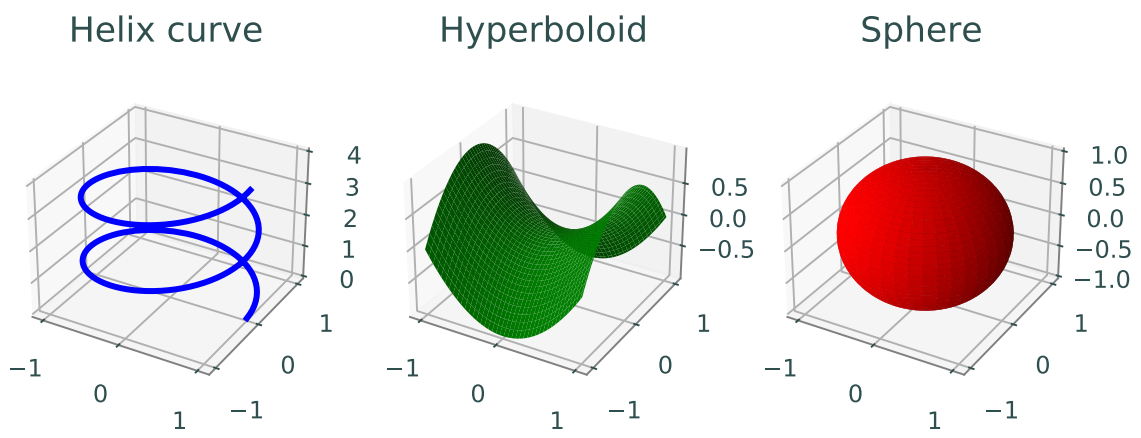


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[GKB17] Ian J. Goodfellow, Alexey Kurakin, and Samy Bengio. Adversarial examples in the physical world. 2017.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. 2015.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.