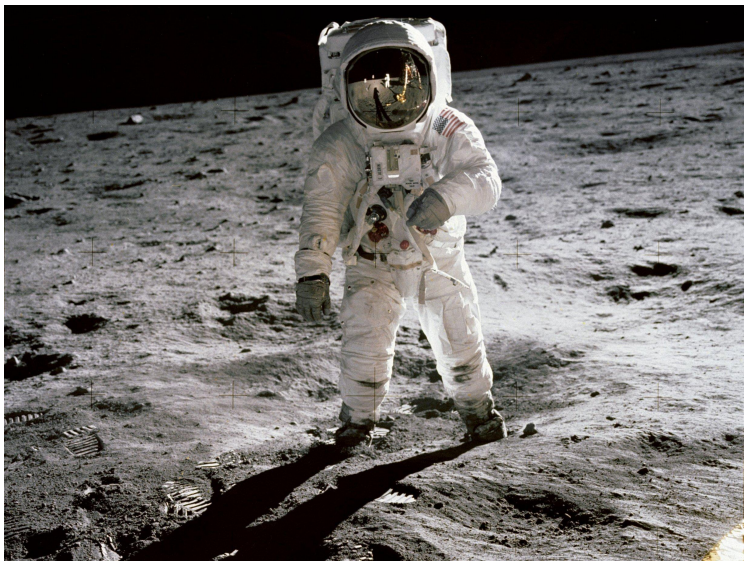
Labs for Foundations of Applied Mathematics

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

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

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

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

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

J. Larsen
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
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

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

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

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

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

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 K-Means Clustering

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

Jupyter Notebooks

Unlike previous labs where the python file submitted was a normal `.py` file, this lab among others will be done in a Jupyter Notebook (`.ipynb` or an iPython Notebook). Jupyter Notebooks is a powerful tool in visualizing data. If you have used Google Colab, this works in a similar manner but it is run on your personal machine.

Once Jupyter Notebook is installed, there are several ways of starting a Jupyter Notebook. The easiest way is to open a new terminal window and navigate to the directory with your .ipynb file, once in the desired directory, type Jupyter notebook. This should automatically open a web browser to the Jupyter Notebook dashboard, from there you can select the .ipynb file and open and edit it.

The Python kernel will keep running in the background until told to stop. So when you are done, to close the Jupyter Notebook, you need to go to file-> Close and Halt, or in the terminal window press ctrl+c (cmd+c for Mac).

> ACHTUNG!
>
> Before you push this file to Bitbucket to be graded, be sure to run each cell. When you push a .ipynb file, the current state of the file is pushed. This means what you see is exactly what the graders will see.

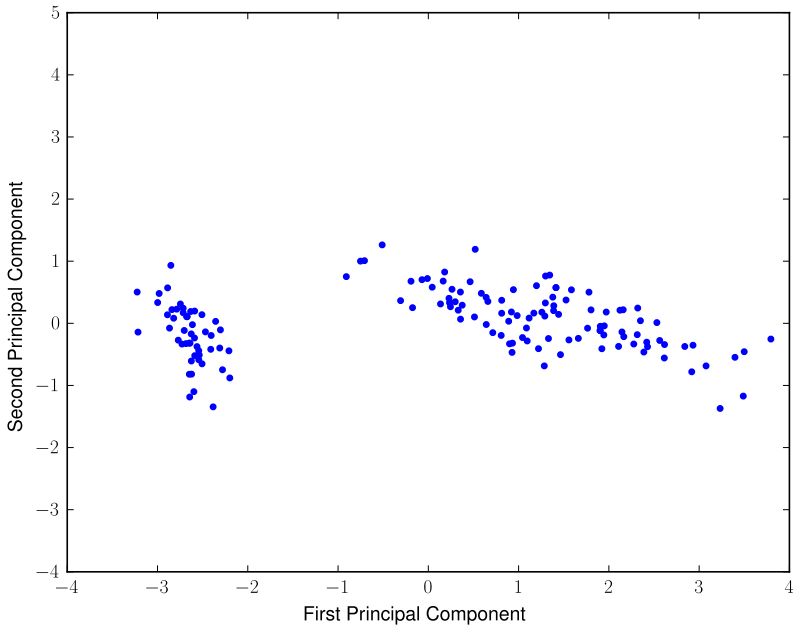


Figure 1.1: The first two principal components of the iris dataset.

Clustering

In this lab, we will analyze a few different datasets from Scikit-Learn's library and use the K-means algorithm. Figure 1.1 is a graph of the iris dataset. As a human, it is easy to identify the two distinct groups of data. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*. The K-means algorithm is a simple way of helping computers see the group distinctions.

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.

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

The K-Means Algorithm

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

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

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

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

Figure 1.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

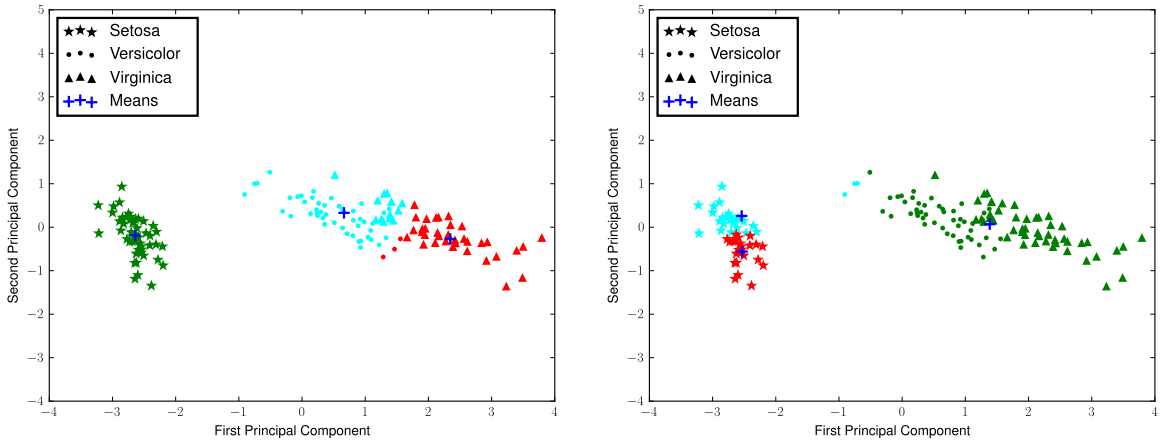


Figure 1.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

The algorithm can be summarized as follows.

1. From the data points, choose k initial cluster centers.
2. For $i = 0, \ \ldots, \texttt{max\_iter}$,

(a) Assign each data point to the cluster center that is closest, forming k clusters.

(b) Recompute the cluster centers as the means of the new clusters.

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

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

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

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

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

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

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

Fire Station Placement

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

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

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

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

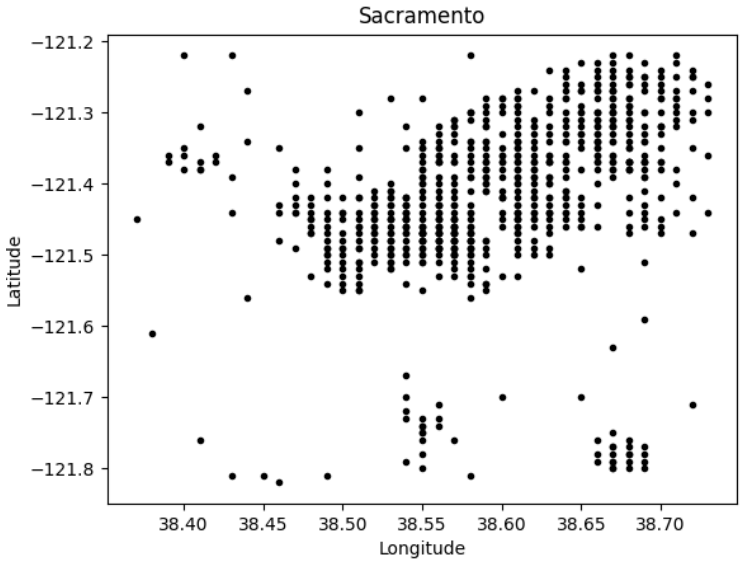


Figure 1.3: Sacramento Housing Data (1990 US Census).

Detecting Active Earthquake Regions

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

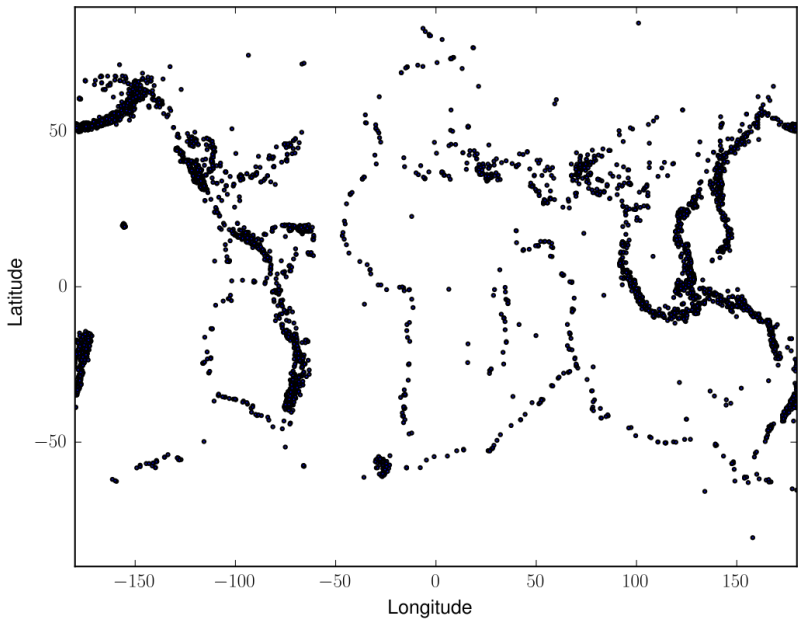


Figure 1.4: Earthquake epicenters over a 6 month period.

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

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

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

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

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

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

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

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

With 15 clusters, your plot should resemble the Figure 1.5.

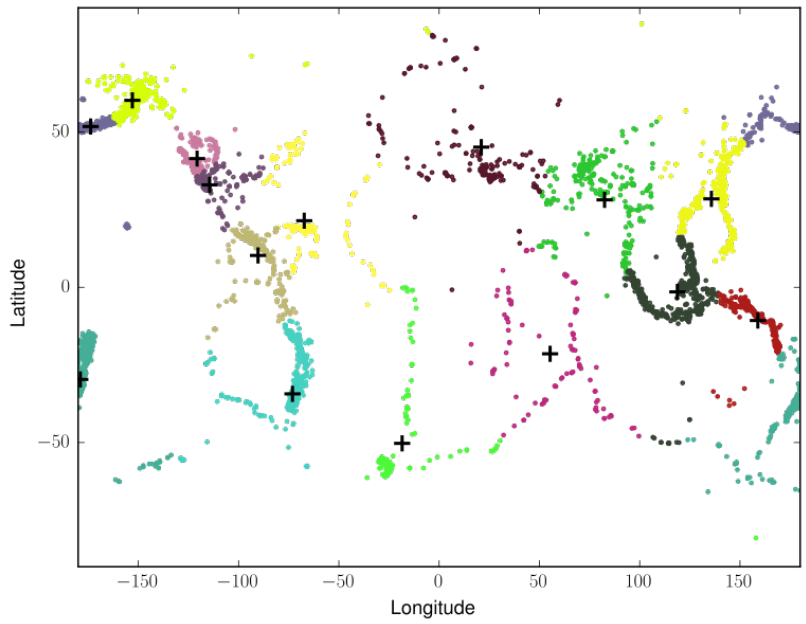


Figure 1.5: Earthquake epicenter clusters with $k = 15$.

Color Quantization

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

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

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

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

Test your function on some of the provided NASA images.

Additional Material

Spectral Clustering

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

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

for some set value σ.

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

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

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

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

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

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

Problem 5. Implement the Spectral Clustering Algorithm by calling your `kmeans` function, using the following function declaration:

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

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

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

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

Problem 6. Create a function that will return the accuracy of your spectral clustering implementation, as follows:

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

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

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

Fire Station Placement II

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

Problem 7. Import and parse the data from the `bayboundary.npy` and the `sanfrancisco.npy` files. Using either the algorithm that you wrote in problem 1 or the k-means algorithm in the SK Learn library, find the optimal locations for the 16 fire stations.

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

2 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 popular 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 **??**.

Figure 2.1: An example game of Wordle.

The secret word is chosen at random from a fixed list of 2309 words. While it is possible to only select guesses from these words, it is not necessarily the best strategy; in many situations, there is a word that can be guessed that gives more information about what the secret word is than guessing any possible secret word would. 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.

[1]Specifically, it went viral on the internet in 2022.

There are a few technicalities with how the guess is evaluated when there are duplicates of a letter. If the secret word were "speak" and a guess of "bevel" was made, the first `e` would be colored yellow and the second gray. Letters that are marked green take priority over this; if "bevel" is guessed and the secret word were "ashes" instead, the first `e` in the guess will be gray and the second green. With the same guess, if the secret word were "steel", then the first `e` would be yellow and the second green. So, if there are more of a given letter in the guess than in the secret word, only as many will be marked yellow or green in the guess as there are in the secret word.

Problem 1. Write a function that accepts the secret word and a guess, and returns the colors of the guess as an array. Label correct letters with the number 2, letters in the wrong location with 1, and incorrect letters with 0. For instance, with the secret word `"pages"` and the guess `"green"`, your function should return `array([1,0,0,2,0])`.

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 in this case to cast the guess and secret word into arrays if you need to modify them.

Problem 2. In order to efficiently implement our strategy for Wordle, we need to know what the result of each guess is for every possible secret word. We only need to make this computation once for each pair, so we will do them all at once and store them in an array.

Load the lists of possible secret words and allowed guesses from `possible_words.txt` and `allowed_words.txt`. Write a function `get_all_guess_results()` that finds the result of making a guess for each pair of secret word and allowed guess. Store the results in a 3-dimensional numpy array, where the first axis corresponds to the guess, the second to the secret word, and the third to the letter.

This computation on the whole set of words will take several minutes at the very least, so test your function on a smaller subset of the word lists. Using the first three secret words and the first two possible guesses, your function should output the following:

```
>>> get_all_guess_results(possible_words[:3], allowed_words[:2])
array([[[2, 1, 0, 0, 0],
        [2, 1, 0, 1, 0],
        [2, 1, 0, 1, 0]],

       [[2, 1, 0, 0, 0],
        [2, 1, 0, 0, 0],
        [2, 1, 0, 0, 0]]])
```

Compute the array for the full word lists. Use `np.save` to save the array to a file, to avoid needing to recompute it.

NOTE

Three-dimensional numpy arrays behave similarly to two-dimensional ones, and can be accessed and sliced in the same way. The only difference is that indexing only a single axis will give a two-dimensional array. We illustrate this by showing how to access some useful subsets of the final array. Here, `all_guess_results` is the output of Problem 2, `i=1388` is the index of a given guess ("boxes"), and `j=1914` is the index of a given secret word ("steel").

```
# This 2-D array is the result of the i-th guess for every secret word
>>> all_guess_results[i,:]
array([[1, 0, 0, 0, 0],
       [1, 0, 0, 1, 1],
       [1, 0, 0, 1, 0],
       ...,
       [1, 0, 0, 1, 0],
       [0, 0, 0, 1, 1],
       [0, 2, 0, 0, 0]])
# This is equivalent to all_guess_results[i] and all_guess_results[i,:,:]

# This 2-D array is the result of every guess for the j-th secret word
>>> all_guess_results[:,j]
array([[0, 0, 0, 2, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       ...,
       [0, 0, 0, 0, 0],
       [0, 0, 0, 2, 1],
       [0, 0, 0, 0, 0]])

# This 1-D array is the result of the i-th guess on the j-th secret word
>>> all_guess_results[i,j]
array([0, 0, 0, 2, 1])
```

Achtung!

We will use this array frequently in the next few problems and modify it in several ways; however, be sure to keep the original array, as it will still be needed. Remember that arrays are mutable, so do not do modifications in-place on the original array. If you accidentally modify the original array, reload it from your file with `np.load`.

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 completely 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 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

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

where the sum version comes from the Law of the Unconscious Statistician. 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 in fact 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.

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 can be used 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 $R(g)$; since we don't know the secret word, this is also a random variable. There are two approaches we can take to make a strategy out of this. First, we can think of trying 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 think of trying 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 result of all of this is that minimizing the entropy of $W|R(g)$ is equivalent to maximizing the entropy of $R(g)$, which we will use to select a good guess. Since the entropy of $R(g)$ is more straightforward to calculate, 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

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

[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, it can be shown that such a function is the negative logarithm of the probability of an event, for some logarithm base; refer to the Volume 3 textbook for more details. The base-2 logarithm is commonly used because it can be thought of as representing the number of bits needed to encode the information.

Since we assumed a uniform distribution over the set of possible secret words, the probability $P(R(g, W) = r)$ can be calculated simply as the proportion of secret words that yield the same result r given the same guess g. We already computed the result of making each acceptable guess with each possible secret word in Problem 2. So, to find the entropy of a guess, we 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:

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 3. Write a function that accepts the multidimensional array created in Problem 2 and calculates the entropy of each guess. Return the guess with the highest entropy. Also return the index of this guess in the list of allowed guesses, to avoid needing to find it later.

In order to simplify determining the numbers of each unique guess result, we can first condense the result of each guess into a single number. A simple way to do this is interpreting the five numbers of the result as a ternary (base 3) number. For instance, we can convert the array `[1,0,2,2,1]` to the number $1 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 2 \cdot 3^3 + 1 \cdot 3^4 = 154$. This step is also simple to vectorize, which makes it a relatively quick operation. Applying this to the whole array from Problem 2 will result in a 2-dimenional array, whose axes correspond to the possible secret words and the allowed guesses. Be sure not to overwrite the original array.

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 the allowed guesses, you can use this function to compute the entropy quickly. Applying this function directly to multidimensional arrays results in different behavior, however.

After we make a guess, we would like to compute the effect of knowing the result on the probability distribution of the secret word. 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", the only value of r for which the probability is not zero is $r = [0, 0, 0, 2, 1]$. This is precisely the result of making that guess for that word.

Now, also note that $P(W = w)$ 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 words that give 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 4. Create a function that filters the list of possible words after making a guess. Since we already computed the result of all guesses for all possible words in Problem 2, we will use this array instead of recomputing the results. Accept this array, the list of possible words, a guessed word's index in the list of allowed guesses, and the guess's result. 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 to compute the entropies for the following guess.

Hint: 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, while we filter down the list of possible secret words, we do not do anything similar for the list of allowed guesses. The reason for this is that, 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. In general, it is beneficial to allow ourselves to guess words that cannot be the secret word, because in some cases we will get more information that way.

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 5. The file `wordle.py` contains a class called `WordleGame` object 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:

- If we know the word, guess it; otherwise, choose a guess at random from the list of allowed guesses.
- Filter the list of possible words to only those that are still possible; this allows us to determine if we know what the secret word is
- Repeat until the secret word has been guessed

Return the number of guesses needed to guess the secret word. 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. This package is included with the Anaconda distribution of python, but can easily be installed with `pip` if needed.

Problem 6. 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, guess that word
- Otherwise, compute the entropies, and make the guess that has the highest entropy
- Filter the possible words to only those that are possible after the guess
- Repeat until the secret word has been guessed

Return the number of guesses needed to guess the secret word.

Problem 7. 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.

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

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

Principal Component Analysis

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

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

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

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

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.

You can load this data using the following code.

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

Write a function that performs PCA on the breast cancer dataset. Graph the first two principal components, with the first along the x-axis. Your graph should resemble Figure 3.1 below. Include in the graph title the amount of variance captured by the first two principal components, calculated with Equation 3.1.

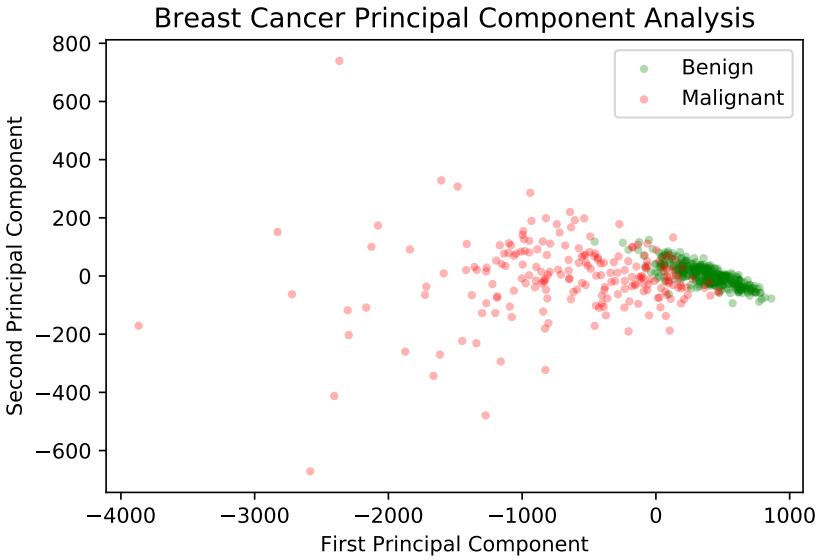


Figure 3.1: First two principal components of the transformed breast cancer data

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 sparse matrix `Xhat` and an index `i` and returns the indices of the most similar and the least similar documents.

Application: State of the Union

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

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

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

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

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

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

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

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

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

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

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

```
...             if word in vocab:
...                 word_index.append(vocab[word])
...                 counts.append(count)
...                 doc_index.append(doc)

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

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

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

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

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

We then let

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

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

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

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

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

Scikit-Learn

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

Transformers

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

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

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

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

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

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

Validation Tools

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

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

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

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 Data Augmentation

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

Data Augmentation

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

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

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

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

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

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

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

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

Figure 4.1: Different image transformations

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

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

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

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

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

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

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

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

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

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

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

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

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

Your augmented score should be better than your original score.

```
from sklearn import datasets

digits = datasets.load_digits()
X, y = digits.data, digits.target
```

Audio Augmentation

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

When adding noise, it may be useful to consider the types of noise most likely to be encountered when the method is in use. For example, if the method will be used to identify voice commands in an outdoor environment, then adding in typical outdoor noises would probably be more useful than adding white noise, which may not occur much in everyday life. Be thoughtful in choosing how to augment your data, as some types of manipulations may change or obscure the data to the point where it is no longer recognizable.

Audio Packages

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

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

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

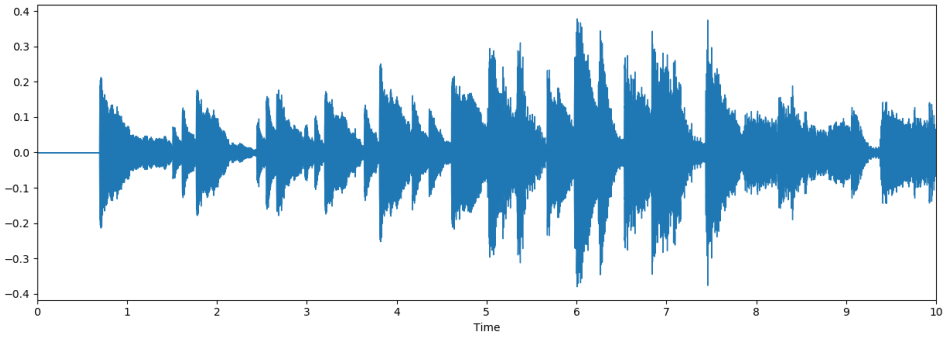


Figure 4.2: Visualization of an audio file

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

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

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

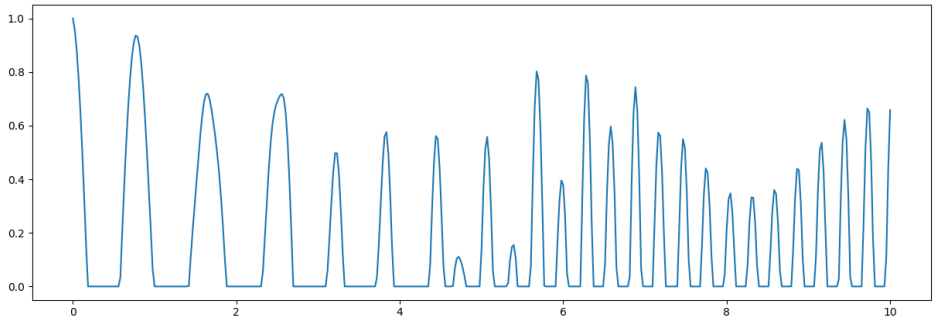


Figure 4.3: Predominant local pulse estimation of audio from figure 4.2

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

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

Table 4.1: These descriptions were taken directly from `librosa.org/librosa/effects.html`

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

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

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

Problem 4. Do the following steps 5 times:

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

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

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

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

Synthetic Minority Oversampling

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

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

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

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

The Algorithm

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

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

We will demonstrate this using data with two features, represented by x and y coordinates. Consider the points $(0, 0)$, $(1, 3)$, $(2, 1)$, and $(3, 2)$, as shown in figure 4.4.

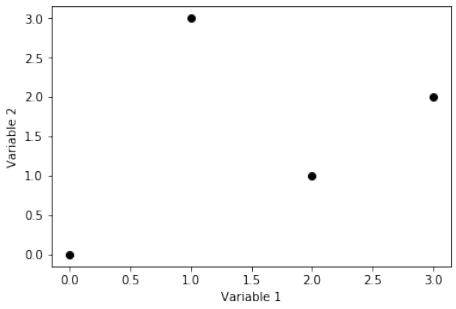


Figure 4.4: Data before SMOTE

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

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

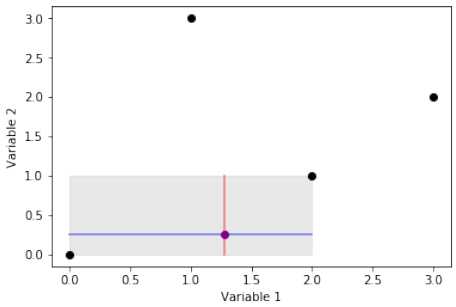
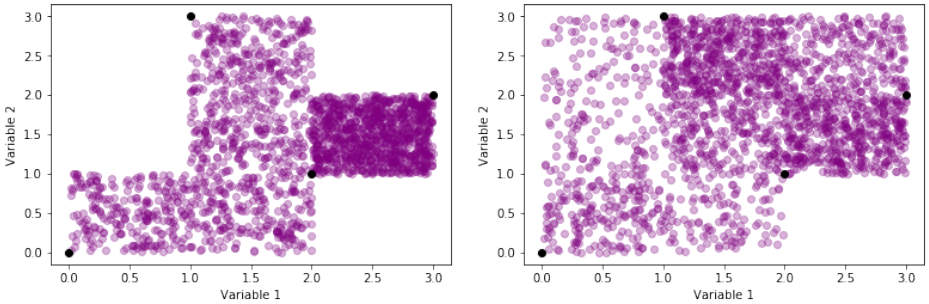


Figure 4.5: SMOTE process

Running this algorithm 500 times per original point, with $k = 1$, returns a graph like figure 4.6a. Running the algorithm 500 times per original point and increasing k to 2, returns the graph like figure 4.6b.



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

Figure 4.6: After SMOTE

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

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

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

Do the following steps 10 times:

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

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

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

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: Use functions from `numpy` or `scipy` to calculate a matrix inverse

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.

Hint: Since `ols` takes `X` without a column of ones, slice `X` when you call `ols`.

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`.

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. 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, and a legend.

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.

Find the combination of variables that builds the model with the best R^2 value when predicting `house price of unit area`. 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.

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 `distance to the nearest MRT station`, `number of convenience stores`, `latitude`, and `longitude`.

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. Then use the grid search result to fit the 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. How do these models compare to the models in problem 6?

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

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

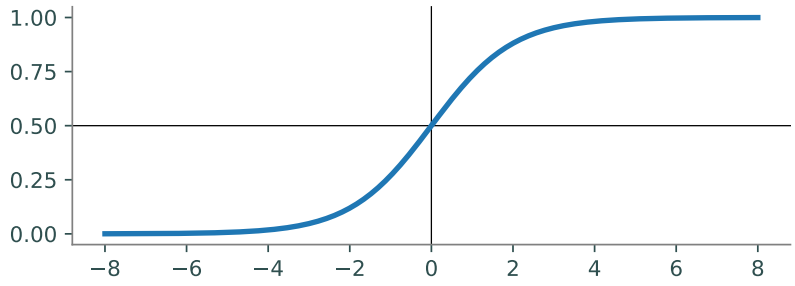


Figure 6.1: Sigmoid Function

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

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

so that

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

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

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

which is equivalent to maximizing

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

where

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

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

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

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

Problem 1. Create a Python classifier called `LogiReg` 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. Write a `fit()` method that uses equation 6.2 to find and save the optimal $\widehat{\boldsymbol{\beta}}$.

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

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

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

Problem 3. To test your classifier, create training arrays X and y as well as 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.

```
>>> 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)
```

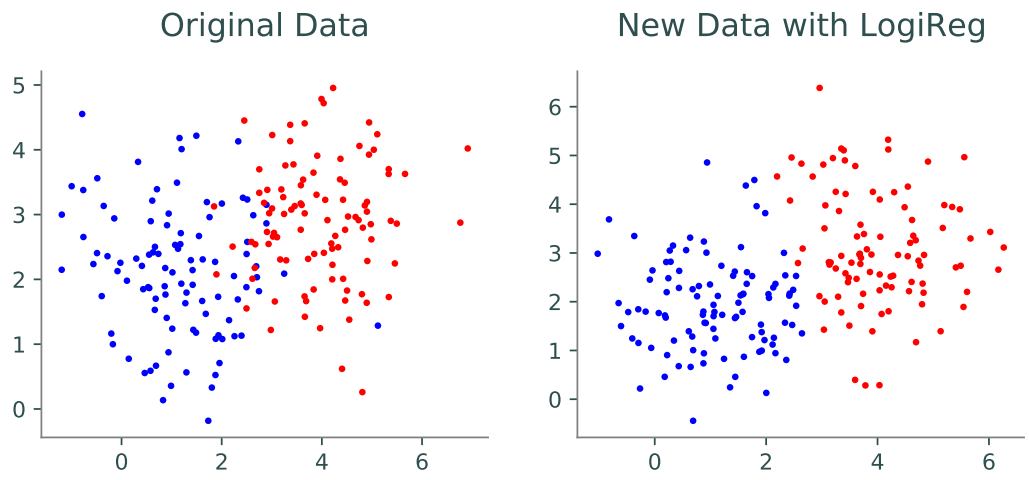


Figure 6.2: In reality, these two distributions overlap a little, but the logistic regression model makes a clean divide between the two.

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
```

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 $\mathbf{0}$, and the other centered at $\mathbf{2}$.

Using each of `LogiReg`, `statsmodels`, and `sklearn`, 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 three 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])] )
```

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 $Y \,|\, X$ is categorically distributed as

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

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

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

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

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

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

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

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

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.

Your plot should reflect Figure 6.3

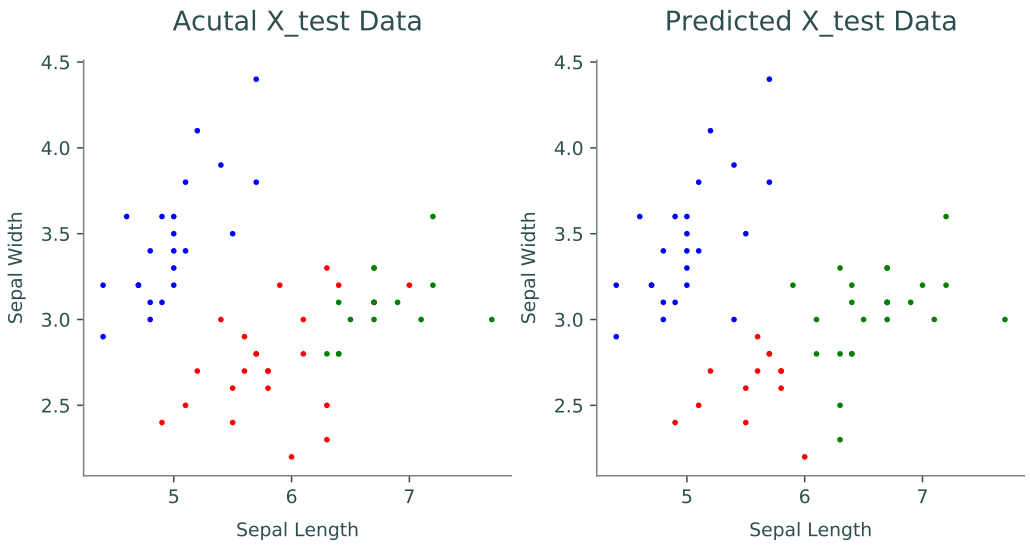


Figure 6.3: Multinomial logistic regression attempt to categorize the Iris Dataset.

7 Naive Bayes

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

About Naïve Bayes

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

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

The Classifier

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

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

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

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. We find that for any $k \in K$ we have

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

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

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

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

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

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

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

Spam Filters

A spam filter is just a special case of a classifier with two classes: spam and not spam (or ham). We can now describe in more detail how we are to calculate Equation 7.2 given that we know what the features are. We can use a labeled training set to determine $P(C=\text{spam})$ the probability of spam and $P(C=\text{ham})$ the probability of ham. To do this we will assume that the training set is a representative sample and define

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

and

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

Using a bag of words model, we can create a simple representation of $P(x_i\,|\,C=k)$ where x_i is the i^{th} word in a message, and therefore $\mathbf{x}$ is the entire message. This results in the simple definition of

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

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. In the case we have some word x_u that is not found in the training set, we can may choose $P(x_u\,|\,C=k)$ so that the computation is not effected, i.e. letting $P(x_u\,|\,C=k)=1$ for unique words.

A First Model

When building a naïve Bayes classifier we need to choose what probability distribution we believe our features to have. For this first model, we will assume that the words are a categorically distributed random variable. This means the random variable may take on say N different values, each value has a certain probability of occurring. This distribution can be thought of as a Bernoulli trial with N outcomes instead of 2.

In our situation we may have N different words which we expect may occur in a spam or ham message, so we need to use the training data to find each word and its associated probability. In order to do this we will make a DataFrame that will allow us to calculate the probability of the occurrence of a certain word x_i based on what percentage of words in the training set were that word x_i. This DataFrame that will allow us to more easily compute Equation 7.5, assuming the words are categorically distributed. While we are creating this DataFrame, it will also be a good opportunity to compute Equations 7.3 and 7.4.

Throughout the lab we will use an SMS spam dataset contained in `sms_spam_collection.csv`. The following code makes full test and train sets, but we will also provide you with code to check against specific subsets.

```
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split

>>> # load in the sms dataset
>>> df = pd.read_csv('sms_spam_collection.csv')

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

>>> # split the data into test and train sets
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7)
```

Training The Model

Problem 1. Create a class `NaiveBayesFilter`, with an `__init__()` method that is empty. Add a `fit()` method which takes as arguments `X`, the training data, and `y` the training labels. In this case `X` is a `pandas.Series` containing strings that are SMS messages. Create a new `DataFrame` with two rows and a column for each vocabulary word with `'spam'` and `'ham'` being the index. Each entry will be the number of times a word appears in spam or ham messages.

For example, `self.data.loc['ham','in']` is the number of times the word "in" appears in ham messages. Save this DataFrame as `self.data`.

Hint: be sure you count the number of occurrences of a word and not a string. For example, when searching the string `'find it in there'` for the word `'in'`, make sure you get 1 and not 2 (because of the `'in'` in `'find'`). The methods `pd.Series.str.split()` and `count()` may be helpful.

```
>>> # checkout what the DataFrame looks like
```

```
>>> NB = NaiveBayesFilter()
>>> NB.fit(X[:300], y[:300])
>>> NB.data.loc['ham','in']
47
>>> NB.data.loc['spam','in']
4
```

Predictions

Now that we have implemented the `fit()` method, we can begin to classify new data. We will do this with two methods, the first will be a method that calculates $P(S \,|\, \mathbf{x})$ and $P(H \,|\, \mathbf{x})$, and the other will determine the more likely of the two and assign a label. While it may seem like we should have $P(C = S \,|\, \mathbf{x}) = 1 - P(C = H \,|\, \mathbf{x})$, we do not. This would only be true if we assume the S and H are independent of $\mathbf{x}$. It is clear that we shouldn't make this assumption, because we are trying to determine the likelihood of S and H based on what $\mathbf{x}$ tells us. Therefore we must compute both $P(C = S \,|\, \mathbf{x})$ and $P(C = H \,|\, \mathbf{x})$.

Problem 2. Implement the `predict_proba()` method in your naïve Bayes classifier. It should take as an argument `X`, the data that needs to be classified, and it will compute the product portion of equation 7.2.

Notice that $P(x_i \,|\, C)$ is the same for every repeated instance of word x_i in message `x`. To save time, we only want to calculate this probability once. To do this, find

$$\prod_{i=1}^{l} P(x_i \,|\, C)^{n_i}$$

for each message `x` in `X` where l is the number of unique words in the message and n_i is the number of times the i^{th} unique word (x_i) occurs.

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

Problem 3. Implement the `predict()` method in your naïve Bayes classifier. This should take as an argument `X`, the data that needs to be classified. Using `predict_proba()`, finish implementing equation 7.2 and return an array that classifies each message `x` in `X`.

```
>>> # create the filter
>>> NB = NaiveBayesFilter()

>>> # fit the filter to the first 300 data points
>>> NB.fit(X[:300], y[:300])

>>> # test the predict function
```

```
>>> NB.predict(X[530:535])
array(['ham', 'spam', 'ham', 'ham', 'ham'], dtype=object)
```

Underflow

There are some issues that we encounter given this implementation. Notice that in the following example, the likelihoods for both spam and ham are 0 for each message.

```
>>> # find the likelihoods for messages 1085 and 2010
>>> NB.predict_proba(X[[1085,2010]])
array([[0., 0.],
       [0., 0.]])
```

This is because the messages are long, and thus involve the product of many numbers that are between 0 and 1. Because of this, we have encountered what is called underflow, where a number becomes so small it is not machine representable. Therefore, we should work in logspace, as to avoid inevitable underflow caused by long messages. If we take the log of equation 7.2 have

$$\text{argmax}_{k\in K} \ln\left(P(C=k)\right) + \sum_{i=1}^{n} \ln\left(P(x_i \,|\, C=k)\right), \tag{7.6}$$

and this problem is still valid since logarithms are monotonically increasing. However, if any of the $P(x_i \,|\, C=k)$ are close to 0, we risk getting an overall value of $-\infty$. To prevent this from happening, we can perform *Laplace add-one smoothing* by adding 1 to the numerator of $P(x_i \,|\, C=k)$ and 2 to its denominator. This method is equivalent to using a Bayesian method for training. Thus, equation 7.5 becomes

$$P(x_i \,|\, C=k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k} + 1}{N_{\text{words in class } k} + 2}. \tag{7.7}$$

Problem 4. Implement `predict_log_proba()` and `predict_log()` using equations 7.6 and 7.7. These methods will take the same arguments and return the same object types as the methods `predict_proba()` and `predict()`, respectively.

Notice how `X[[1085,2010]]` is now classifiable.

The Poisson Model

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

$$P(n_i = x) = \frac{(rn)^x e^{-rn}}{x!} \tag{7.8}$$

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 could easily refactor this model to use Bayesian inference to determine r, which would allow greater control over the model. This would also create a condition where the training data doesn't completely determine the model's viability. However, in this lab we will use maximum likelihood estimation to determine r.

Training the Model

Similar to the other classifier that we made, training the model amounts to using the training data to determine how $P(x_i \,|\, C = k)$ is computed, as well as compting $P(C = k)$. As stated earlier, we will attempt to find the most likely value of r for each word that appears in the training set. To do this we will use maximum likelihood estimation. The parameter we choose is the one that maximizes the likelihood function

$$\hat{r} = \text{argmax}_r L(r \,|\, \mathbf{x}) = \text{argmax}_r P(\mathbf{x} \,|\, r).$$

In this case, since we are using a Poisson distribution (7.8) for each word, we will solve the following problem for both the spam class and the ham classes

$$r_{i,k} = \text{argmax}_{r\in[0,1]} \frac{(rN_k)^{n_i} e^{-rN_k}}{n_i!}, \tag{7.9}$$

where $r_{i,k}$ is the Poisson rate for the i^{th} word in class k (either spam or ham), N_k is the total number of words in class k, and n_i 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. If we take the derivative of the right side of equation 7.9 with respect to r, set it equal to 0, and solve for the maximizing r, we find that $r_{i,k} = n_i/N_k$.

Predictions

Making predictions with this model is exactly the same as we did earlier. To clarify the calculation, lets reformulate 7.6 to fit the Poisson case better. This gives

$$\text{argmax}_{k\in K} \ln\left(P(C = k)\right) + \sum_{i=1}^{l} \ln\left(\frac{(r_{i,k}n)^{n_i} e^{-r_{i,k}n}}{n_i!}\right), \tag{7.10}$$

with l being the number of unique words in the message, 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. Notice, if $r_{i,k}$ is close to 0, we'll risk getting a total value of $-\infty$. We can fix this by using the *Laplace add-one smoothing* method as we did before, but this time on $r_{i,k}$. Thus, our new Poisson rate for the i^{th} word in class k becomes

$$r_{i,k} = \frac{n_i + 1}{N_k + 2}, \tag{7.11}$$

which has a Bayesian interpretation, as it did before.

Problem 5. Create a new class called `PoissonBayesFilter` with an `__init__()` method that may be empty. Add a `fit()` method which takes as arguments training data `X` and training labels `y`.

Implement `fit()` by finding the MLE found in equation 7.11 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.

```
>>> #create a poisson bayes object to examine it
>>> 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
```

Problem 6. Implement the `predict_log_proba()` and `predict()` methods using equation 7.10. These methods will take the same arguments and return the same object types as the methods `predict_proba()` and `predict()` in the `NaiveBayesFilter` class, respectively. You may use `scipy.stats.poisson.pmf` if you wish.

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. 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.

The results of Problem 7 can help you test the two Bayes Filters you created in this lab. Using the `accuracy_score` method of `sklearn.metrics`, you can compare your predicted labels with the ones from Problem 7. You should have very high accuracy, as demonstrated below.

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

>>> # labels returned by Problem 7
>>> actual_labels = sklearn_method(X_train, y_train, X_test)

>>> # test against NaiveBayesFilter
>>> NB = NaiveBayesFilter()
>>> NB.fit(X_train, y_train)
>>> NB_labels = NB.predict_log(X_test)
>>> accuracy_score(actual_labels, NB_labels)
0.9769289925660087

>>> # test against PoissonBayesFilter
>>> PB = PoissonBayesFilter()
>>> PB.fit(X_train, y_train)
>>> PB_labels = PB.predict(X_test)
>>> accuracy_score(acutal_labels, PB_labels)
0.9782107152012305
```

References

Rish, Irina. (2001). An Empirical Study of the Naïve Bayes Classifier. IJCAI 2001 Work Empir Methods Artif Intell. 3.

Data from: http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/

8 Random Forests

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

Classification Trees

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

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

A classification tree is composed of many *nodes*, which ask a question (i.e. "Is income >= 85?") and then split the data based on the answers. If the response is `True`, then the sample is "pushed" down the tree to the left child node. If the response is `False`, then the sample is "pushed" down the tree to the right child node. A *leaf* node is a node that has no child node. Upon arrival at a leaf, an unlabeled sample is labeled with the classification that matches the majority of labeled samples at that leaf. Table 8.1 includes information about 10 individuals and then an indicator of whether or not they made a certain purchase. To simplify construction of the tree, all data is numeric, so 1=Yes and 0=No for yes/no questions.

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

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

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

Table 8.1: Customer data with 3 features (Married, Children, Income) and a label (Purchase) indicating whether or not the customer bought the item.

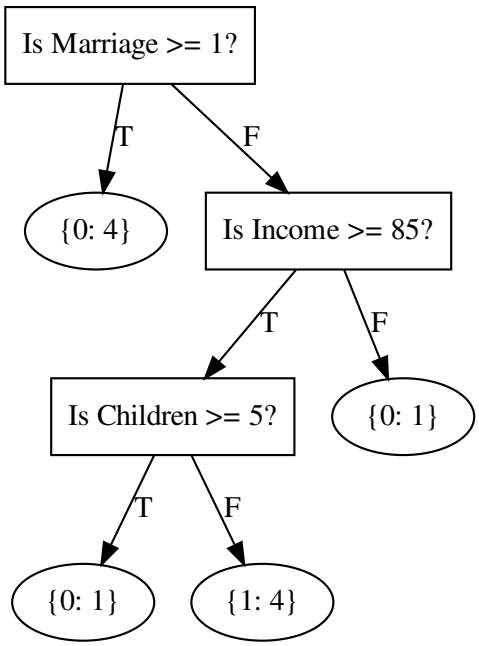


Figure 8.1: A classification tree built using Table 8.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

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

Next, write a `partition` function that partitions a data set for a given question into two `numpy` arrays: `left` and `right`. Note that `left` will contain samples where the `match` method returns `True` and `right` will contain samples where the `match` method returns `False`. Return the left and right regions of the partition in that order. If one region is empty, return it as `None`.

Measures

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

Definition 8.1. *Let D be a data set with K different class labels and N different samples. Let N_k be the number of samples labeled class k for each $1 \leq k \leq K$, and let $f_k = \frac{N_k}{N}$. We define the* Gini impurity *to be*

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

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

Definition 8.2. *Let $s_D(p, x) = D_1, D_2$ be a partition of data D. We define the* information gain *of this partition to be*

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

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

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

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

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

```
# Load in sample names
>>> names = np.loadtxt('animal_names.csv', delimiter=',', dtype=str)

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

Optimal Split

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

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

where

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

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

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

The output for the animals data set should be

`(0.12259833679833688, Is # legs/tentacles >= 2.0?)`.

Building the Tree

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

Problem 4. Write the class `Leaf`. It should have an attribute `prediction` that is the dictionary of how many samples at the leaf belong to each label, as shown in the leaves of Figure 8.1.

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

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

Problem 5. Write a function `build_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 8.1 has depth 3. You will probably want to build this tree recursively.

Make a `Leaf` if the remaining data has too few samples, if the depth is too much, or if the information gain is 0. Otherwise, make a partition and build a new tree for each branch, returning those as `Decision_Nodes`.

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

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

ACHTUNG!

The function `draw_tree` relies on the `graphviz` package. These are are two options to aid in installing the `graphviz` package.

- You can try downloading by typing `conda install -c conda-forge python-graphviz` if you have the Anaconda distribution. If `draw_tree` returns an error about pdf being an unrecognized file type, try typing `dot -c` in your terminal.
- If you get an error related to a `PATH` problem you may need to download `graphviz` to your computer by following the instructions found at this link: Download `graphviz`.

Predicting

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

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

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

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

Random Forest

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

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

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

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

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

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

Scikit-Learn

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

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

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

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

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

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

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

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

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

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

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

9 Metropolis Algorithm

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

The Metropolis Algorithm

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

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

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

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 9.1. The Metropolis algorithm can be interpreted as follows: given our current state $\mathbf{y}$, we propose a new state according to the distribution $Q(\cdot, \mathbf{y})$. We then accept or reject it according to A. We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h. Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

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

Algorithm 9.1 Metropolis Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

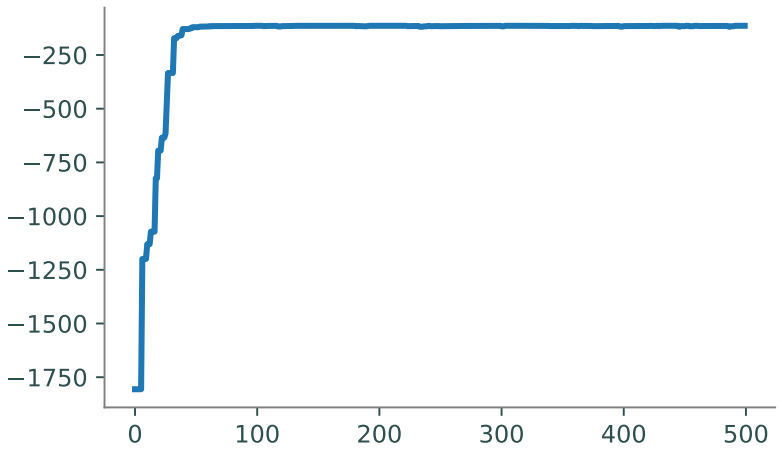


Figure 9.1: Log densities of the first 500 Metropolis samples.

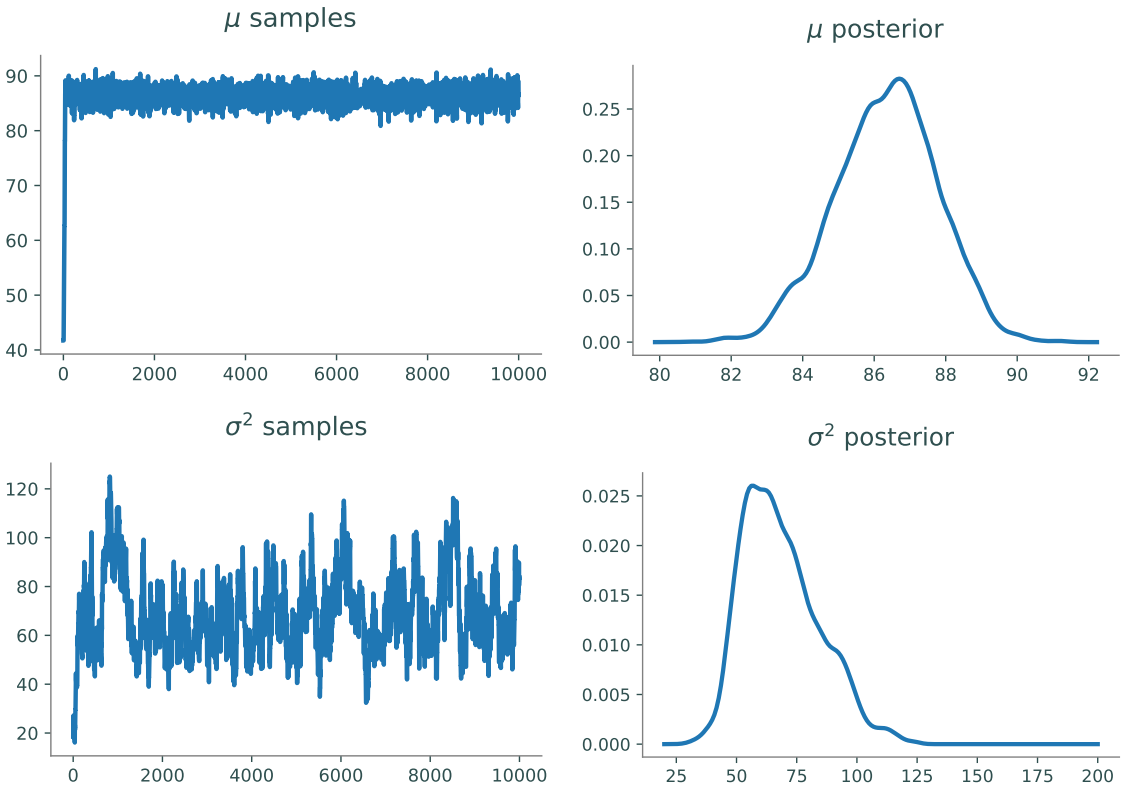


Figure 9.2: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

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 9.1 to complete the problem.

Your function should return an array of draws, an array of the log probabilities, and an acceptance rate. Use the following code to check your work. Using the seaborn.kdeplot function, plot the first 500 log probabilities, the μ samples and posterior distribution, and the σ^2 samples and posterior distribution. The results should be *similar* to Figures 9.1 and 9.2.

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

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

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

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

```
>>> s = 4

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

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
```

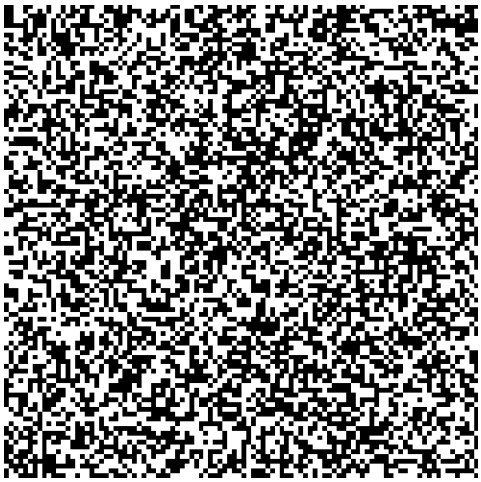


Figure 9.3: Spin configuration from random initialization.

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

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases} \tag{9.1}$$

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

$$H(\sigma^*) = H(\sigma) + 2 \sum_{j:j\sim i} \sigma_i \sigma_j. \tag{9.2}$$

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 (9.2) 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 (9.1), 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.

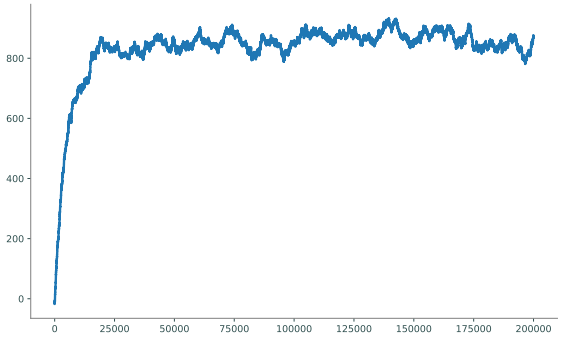
1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration σ^*.
2. Use Problem 4 to calculate the energy $H(\sigma^*)$ of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set $\sigma = \sigma^*$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

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

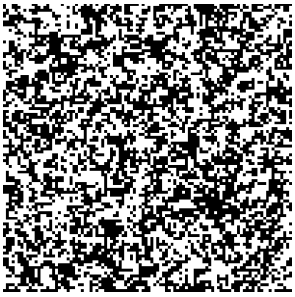
After the burn-in period, continue the iteration `n_samples` times, also recording every 100th sample (to prevent memory failure). The acceptance rate is counted after the burn-in period. Return the samples, the sequence of weighted energies $-\beta H(\sigma)$, and the acceptance rate.

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 9.4.

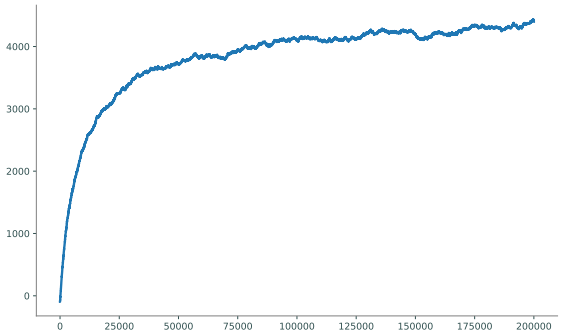
To show the spin configuration, use `plt.imshow(L,cmap='gray')`.



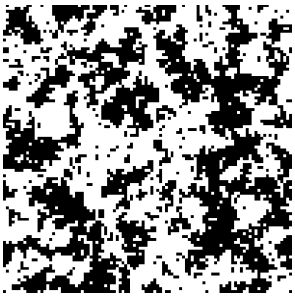
(a) Proportional log probs when $\beta = 0.2$.



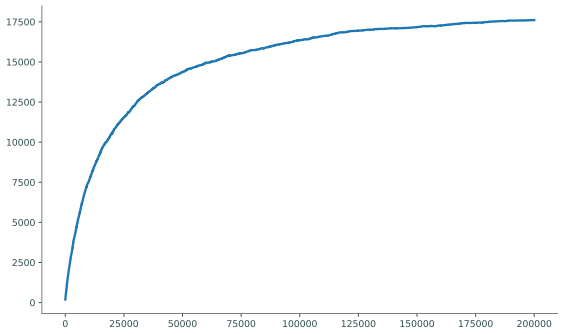
(b) Spin configuration sample when $\beta = 0.2$.



(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 9.4

10 Gibbs Sampling and LDA

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

Gibbs Sampling

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

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

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

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

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

Algorithm 10.1 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 10.1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of $\mathbf{x}^{(k)}$ after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible $\mathbf{x}$. The unique invariant distribution for the chain is the desired joint distribution

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

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

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

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

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

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

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

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

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

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

where

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

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

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

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

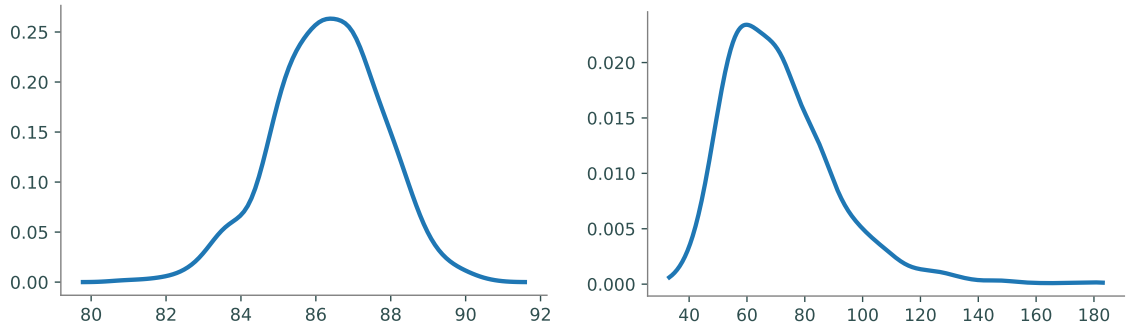
Problem 1. Write a function that accepts data $\mathbf{y}$, prior parameters ν, τ^2, α, and β, and an integer n. Use Gibbs sampling to generate n samples of μ and σ^2 for the exam scores problem.

Test your sampler with priors $\nu = 80$, $\tau^2 = 16$, $\alpha = 3$, and $\beta = 50$, collecting 1000 samples. Plot your samples of μ and your samples of σ^2. They should 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 10.1: Posterior marginal probability densities for μ and σ^2.

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

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

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

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

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

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

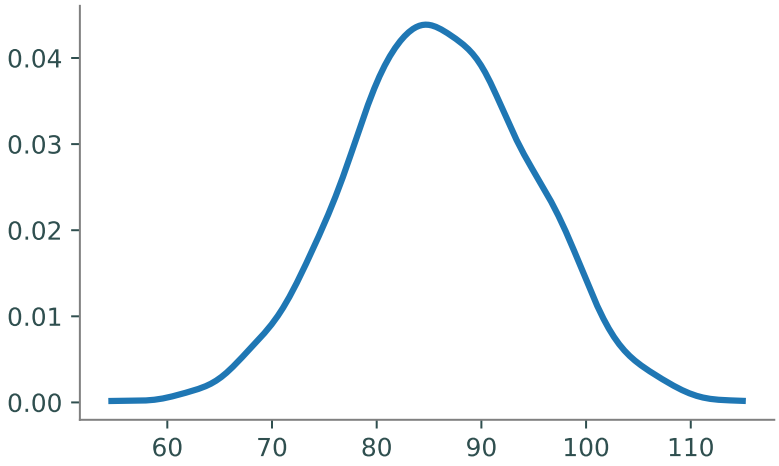


Figure 10.2: Predictive posterior distribution of exam scores.

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 10.1.

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 10.2.

Latent Dirichlet Allocation

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

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

We will now iterate through each document in the same manner. Assume we are working on document m, which you will recall contains N_m words. For word n, we first draw a topic assignment $z_{m,n}$ from the categorical distribution θ_m, and then we draw a word $w_{m,n}$ from the categorical distribution $\phi_{z_{m,n}}$. Throughout this implementation, we assume α and β are scalars[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 $w_{m,n} \sim \text{Cat}(\phi_{z_{m,n}})$ for $1 \leq n \leq N_m$.

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

This is typically depicted with graphical plate notation as in Figure 10.3.

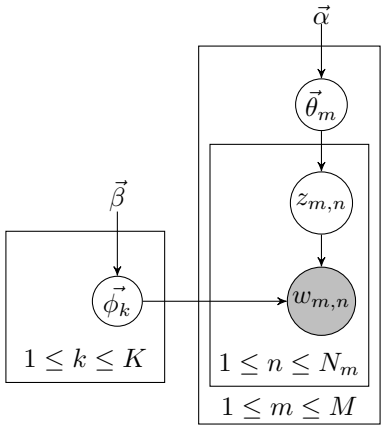


Figure 10.3: Graphical plate notation for LDA text generation.

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

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

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

[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:

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

where

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

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 $n : w$, where w 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 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 z as an integer from the correct range of topics. Then, you will increment each of the count matrices by 1, given the values for z, m, and w, where w 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 z. 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, where each line is considered a document. The corpus is built by method `buildCorpus()`, and stopwords are removed (if argument *stopwords* is provided).

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, but only for 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 one of Ronald Reagan's State of the Union addresses, found in `reagan.txt`.

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

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

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. For each topic, decide what these ten words jointly represent, and come up with a label for them.

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

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

Additional Material

LDACGS Source Code

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

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

            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:w, where w 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})

        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 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 z as an integer from the range of topics.
            Then, you will increment each of the count matrices by 1,
            given the values for z, m, and w, where w 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,i):z

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


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


    def 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, where each line is considered a document.
            The corpus is built by method buildCorpus(), and
            stopwords are removed (if argument stopwords is provided).

            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,
            but only for 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_nzw = np.zeros((self.n_topics, self.n_words))
```

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

        for i in range(burnin):
            # Sweep and store log likelihood.
            raise NotImplementedError("Problem 5 Incomplete")
        for i in range(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, w):
        """ Returns the conditional distribution given m and w.
            Called by _sweep(). """
        dist = (self.nkm[:,m] + self.alpha) * (self.nkv[:,w] + 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 z in range(self.n_topics):
            lik += np.sum(gammaln(self.nkv[z,:] + self.beta)) - gammaln(np.sum(↩
                self.nkv[z,:] + self.beta))
            lik -= self.n_words * gammaln(self.beta) - gammaln(self.n_words * ↩
                self.beta)

        for m in range(self.n_docs):
            lik += np.sum(gammaln(self.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 topterms. """
        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()
        self.theta()
        vec = np.atleast_2d(np.arange(0, self.n_words))
        topics = []
        for k in range(self.n_topics):
            probs = np.atleast_2d(self._phi[k,:])
            mat = np.append(probs, vec, 0)
            sind = np.array([mat[:,i] for i in np.argsort(mat[0])]).T
            topics.append([self.vocab[int(sind[1, self.n_words - 1 - i])] for i↩
                in range(n_terms)])
        return topics
```

11 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

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

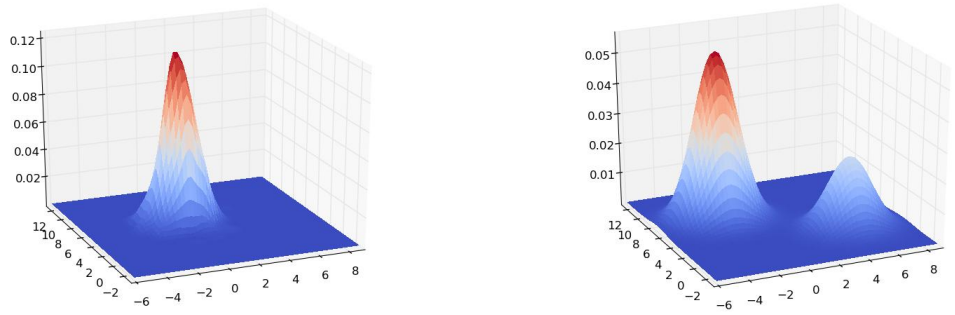
where

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

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 11.1 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 11.1

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.

To test your functions, create the following 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 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 $Z \sim \mathcal{N}(\mu_X, \sigma_X)$.

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)`.

Draw a sample of 10,000 points from the GMM defined in Problem 2. Plot the pdf of the GMM (using `plt.pcolormesh`) and a hexbin plot of the drawn points. How do the plots compare?

The following code can be used to plot the pdf:

```
## Create the grid to plot on
x = np.linspace(-8,8,100)
y = np.linspace(-8,8,100)
X, Y = np.meshgrid(x, y)
## Calculate the pdf at each point
# If your pdf function uses array broadcasting, you can do the following:
Z = gmm.pdf(np.dstack((X,Y)))
# Otherwise, you need to iterate over each point:
Z = np.array([[
        gmm.pdf([X[i,j], Y[i,j]]) for j in range(100)
        ] for i in range(100)
    ])
## Create the plot
plt.pcolormesh(X, Y, Z, shading='auto')
```

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 $\theta^t = (w_1^t, \dots, w_K^t, \mu_1^t, \dots, \mu_K^t, \Sigma_1^t, \dots, \Sigma_K^t)$. Note that t is an index, not an exponent. For each data point $z_i, 1 \leq i \leq n$ and each component $1 \leq k \leq K$, the E-step consists of computing

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

In order to accurately compute this quantity, however, we need to be more careful. It is possible that due to floating point underflow[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.

[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.

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.

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

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

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

Problem 5. Write a method `_compute_m_step` for your GMM class that performs a single iteration of the EM algorithm. Return the updated parameters, as given by the M-step. Be sure to use array broadcasting when possible.

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)))
```

Problem 7. The file `problem7.npy` contains a collection of data drawn from a two-dimensional GMM. Train a GMM on this data with `n_components=3`. Plot the pdf of your trained GMM (in the same way as in Problem 3), as well as a hexbin plot of the data. Your class should take less than 15 seconds to train on this dataset.

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 8. 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.

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. We now compare the effectiveness of GMMs for classification on this dataset with K-means, as well as comparing to sklearn's implementation.

Problem 9. Again using `classification.npz`, compare your class, sklearn's GMM implementation, and sklearn's K-means implementation for speed of training and for accuracy of the resulting clusters. Print your results. Be sure to check for permuted labels.

You should find that sklearn's GMM is actually faster on this dataset than K-means. 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.

12 Discrete Hidden Markov Models

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

In this lab, we explore Hidden Markov Models (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 $|\mathscr{Z}| = m$. In addition, a discrete state-space HMM has parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$ and an observation sequence $\boldsymbol{z}$. We would like to answer three questions about an HMM:

1. What is the likelihood that our model generated the observation sequence? In other words, what is $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$?
2. What is the most likely state sequence $\boldsymbol{x}$ to have generated $\boldsymbol{z}$, given $\boldsymbol{\theta}$?
3. How can we choose the parameters $\boldsymbol{\theta}$ that maximize $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$?

The first question is answered using the *forward pass* algorithm. For the second question, the approach taken in this lab will be to find the state sequence maximizing the expected number of correct states. The third question is an example of *unsupervised learning*, since we are attempting to learn (or fit) model parameters using data (the observation sequence $\boldsymbol{z}$) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

In this context $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$, where $\boldsymbol{\pi}$ is a stochastic vector of length n (the initial state distribution), A is a $n \times n$ column-stochastic matrix (the state transition model), and B is a $m \times n$ column-stochastic matrix (the state observation model). Further, $\boldsymbol{z}$ is a vector of length T with values in the set $\mathscr{Z} = \{0, 1, 2, \ldots, m-1\}$.

Throughout this lab, we will be using the following toy HMM to verify your code.

```
>>> # toy HMM example to be used to check answers
>>> pi = np.array([.6, .4])
>>> A = np.array([[.7, .4],[.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> z = np.array([0, 1, 0, 2])
```

Problem 1. To start off your implementation of the HMM, define a class object which you should call `HMM`. Then add the initialization method, storing the `self` aspects `pi`, `A`, and `B` as `None` objects. You will be adding methods throughout the remainder of the lab.

The Forward Pass

Our first task is to efficiently compute $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$. We can do this using the *forward pass* algorithm.

First, let $\alpha_t(i) = P(z_0, \dots, z_t, x_t = i \,|\, \boldsymbol{\theta})$. Then using the law of total probability and $\alpha_t(i)$, we can efficiently compute $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$ as

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

Now we must use a rescaled version of the forward pass to prevent the $\alpha_t(i)$s from becoming too small as t gets large. Let $\odot$ denote the Hadamard (entry-wise) product of arrays. The algorithm is as below.

Algorithm 12.1 Forward Pass Algorithm

procedure Forward Pass Algorithm
  for t=0 **do**
    Set $\widehat{\alpha}_0(i) = \pi_i \cdot B_{z_0 i}, \forall i \in \mathscr{X}$
    Let $c_0 = 1/\big(\sum_{j \in \mathscr{X}} \widehat{\alpha}_0(j)\big)$
    Set $\widehat{\alpha}_0(i) = c_0 \odot \widehat{\alpha}_0(i), \forall i \in \mathscr{X}$
  for t=1, . . . , T - 1 **do**
    Compute $\tilde{\alpha}_t(i) = \sum_{j \in \mathscr{X}} \widehat{\alpha}_{t-1}(j) \cdot A_{ij} \cdot B_{z_t i}, \forall i \in \mathscr{X}$
    Compute $c_t = 1/\big(\sum_{j \in \mathscr{X}} \tilde{\alpha}_t(j)\big)$
    Rescale by setting $\widehat{\alpha}_t(i) = c_t \cdot \tilde{\alpha}_t(i), \forall i \in \mathscr{X}$

The matrix $\widehat{\alpha}$ will be of use when fitting parameters, but we can compute the desired log probability using the scaling factors c_t as follows:

$$\log P(\boldsymbol{z} \,|\, \boldsymbol{\theta}) = -\sum_{t=0}^{T-1} \log c_t.$$

Problem 2. Implement the forward pass by adding the following method to your class:

```
def _forward(self, z):
    """
    Compute the scaled forward probability matrix and scaling factors.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence

    Returns
```

```
    -------
    alpha : ndarray of shape (T, n)
        The scaled forward probability matrix
    c : ndarray of shape (T,)
        The scaling factors c = [c_0, c_1, ..., c_{T-1}]
    """
    pass
```

To verify that your code works, you should get the following output using the toy HMM:

```
>>> h = HMM()
>>> h.pi = pi
>>> h.A = A
>>> h.B = B
>>> alpha, c = h._forward(z)
>>> print(-np.log(c).sum()) # the log prob of observation
-4.6429135909
```

The Backward Pass

The backward pass produces values that can be used to calculate the most likely state sequence corresponding to an observation sequence.

We compute a scaled backward probability matrix $\widehat{\beta}$ of dimension $T \times n$ as follows:

Algorithm 12.2 Backward Pass Algorithm

1: **procedure** Backward Pass Algorithm
2: When $t = T - 1$, set $\widehat{\beta}_{T-1}(i) = c_{T-1}$, $\forall i \in \mathscr{X}$
3: **for** $t = T - 2, \ldots, 0$ **do**
4: Compute $\tilde{\beta}_t(j) = \sum_{i \in \mathscr{X}} A_{ij} \cdot \widehat{\beta}_{t+1}(i) \cdot B_{z_{t+1} i}$, $\forall j \in \mathscr{X}$
5: Rescale by setting $\widehat{\beta}_t(i) = c_t \cdot \tilde{\beta}_t(i)$, $\forall i \in \mathscr{X}$

Problem 3. Implement the backward pass by adding the following method to your class:

```
def _backward(self, z, c):
    """
    Compute the scaled backward probability matrix.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    c : ndarray of shape (T,)
        The scaling factors from the forward pass
```

```
    Returns
    -------
    beta : ndarray of shape (T, n)
        The scaled backward probability matrix
    """
    pass
```

Using the same toy example as before, your code should produce the following output:

```
>>> beta = h._backward(z, c)
>>> print(beta)
[[ 3.1361635   2.89939354]
 [ 2.86699344  4.39229044]
 [ 3.898812    2.66760821]
 [ 3.56816483  3.56816483]]
```

Computing the ξ and γ Probabilities

Having implemented both parts of the forward-backward algorithm, we are closing in on the solution to question three, namely that of fitting parameters $\boldsymbol{\theta}$ that maximize $P(\boldsymbol{z} \,\big|\, \boldsymbol{\theta})$. At this stage, we combine the information accumulated in the forward and backward algorithms to produce a three-dimensional array $\boldsymbol{\xi}$ of shape $(T-1) \times n \times n$ whose entries are related to $P(\mathbf{x}_t = i, \mathbf{x}_{t+1} = j \,\big|\, \boldsymbol{z}, \boldsymbol{\theta})$, as well as a $T \times n$ matrix $\boldsymbol{\gamma}$ whose entries are related to $P(\mathbf{x}_t = i \,\big|\, \boldsymbol{z}, \boldsymbol{\theta})$. The relevant formulae are

$$\xi_t(i,j) = \widehat{\alpha}_t(i) A_{j,i} B_{z_{t+1},j} \widehat{\beta}_{t+1}(j)$$

for $t = 0, \ldots, T-1$ and $i, j \in \mathscr{X}$,

$$\gamma_t(i) = \widehat{\alpha}_t(i) \widehat{\beta}_t(i) / c_t$$

for $t = 0, \ldots, T-1$ and $i \in \mathscr{X}$.

Problem 4. Add the following method to your class to compute the ξ and γ probabilities.

```
def _xi(self, z, alpha, beta, c):
    """
    Compute the xi and gamma probabilities.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    alpha : ndarray of shape (T, n)
        The scaled forward probability matrix from the forward pass
    beta : ndarray of shape (T, n)
        The scaled backward probability matrix from the backward pass
    c : ndarray of shape (T,)
```

```
        The scaling factors from the forward pass

    Returns
    -------
    xi : ndarray of shape (T-1, n, n)
        The xi probability array
    gamma : ndarray of shape (T, n)
        The gamma probability array
    """
    pass
```

While writing a triply-nested loop may be the simplest way to convert the formula into code, it is possible to use array broadcasting to eliminate two of the loops, which will speed up your code.

Check your code by making sure it produces the following output, using the same toy example as before.

```
>>> xi, gamma = h._xi(z, alpha, beta, c)
>>> print(xi)
[[[ 0.14166321  0.0465066 ]
  [ 0.37776855  0.43406164]]

 [[ 0.17015868  0.34927307]
  [ 0.05871895  0.4218493 ]]

 [[ 0.21080834  0.01806929]
  [ 0.59317106  0.17795132]]]
>>> print(gamma)
[[ 0.18816981  0.81183019]
 [ 0.51943175  0.48056825]
 [ 0.22887763  0.77112237]
 [ 0.8039794   0.1960206 ]]
```

Choosing Better Parameters

After running the forward-backward algorithm and computing the ξ probabilities, we are now in a position to choose new parameters $\boldsymbol{\theta}' = (\boldsymbol{\pi}', A', B')$ that increase the probability of observing our data, i.e.

$$P(\boldsymbol{z} \,|\, \boldsymbol{\theta}') \geq P(\boldsymbol{z} \,|\, \boldsymbol{\theta}).$$

The update formulae are given by

$$\begin{aligned}\boldsymbol{\pi}' &= \gamma_0(i) \\ A'_{i,j} &= \frac{\sum_{t=0}^{T-2} \xi_t(i,j)}{\sum_{t=0}^{T-2} \gamma_t(j)} \\ B'_{i,j} &= \frac{\sum_{t=0}^{T-1} \gamma_t(j)\delta_{z_t=i}}{\sum_{t=0}^{T-1} \gamma_t(j)}\end{aligned}$$

where $\delta_{z_t=i}$ equals 1 if $z_t = i$, and it equals 0 otherwise.

Problem 5. Implement the parameter update step by adding the following method to your class:

```
def _estimate(self, z, xi, gamma):
    """
    Estimate better parameter values and update self.pi, self.A, and
    self.B in place.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    xi : ndarray of shape (T-1, n, n)
        The xi probability array
    gamma : ndarray of shape (T, n)
        The gamma probability array
    """
    pass
```

Verify that your code produces the following output on the toy HMM from before:

```
h._estimate(z, xi, gamma)
>>> print(h.pi)
[ 0.18816981  0.81183019]
>>> print(h.A)
[[ 0.55807991  0.49898142]
 [ 0.44192009  0.50101858]]
>>> print(h.B)
[[ 0.23961928  0.70056364]
 [ 0.29844534  0.21268397]
 [ 0.46193538  0.08675238]]
```

Fitting the Model

We are now ready to put everything together into a learning algorithm. Given a sequence of observations, a maximum number of iterations K, and a convergence tolerance threshold ε, we fit a HMM model using the following procedure:

Algorithm 12.3 HMM Fitting Algorithm

procedure HMM Fitting Algorithm
    Randomly initialize parameters $\boldsymbol{\theta} = (\pi, A, B)$.
    Compute $\log P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$
    for $i = 0, 1, \ldots, K-1$ **do**
        Run forward pass
        Run backward pass
        Compute ξ and γ probabilities
        Update model parameters
        Compute $\log P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$ according to new parameters
        if Change in log probabilities is less than ε **then**
            break
        else
            continue

The most convenient way to randomly initialize stochastic matrices is to draw from the Dirichlet distribution, which produces vectors with nonnegative entries that sum to 1. The following Python code initializes M, $\boldsymbol{\pi}$, A, and B using this technique:

```
>>> # assume N is defined
>>> # define M to be the number of distinct observed states
>>> M = len(set(obs))
>>> pi = np.random.dirichlet(np.ones(N))
>>> A = np.random.dirichlet(np.ones(N), size=N).T
>>> B = np.random.dirichlet(np.ones(M), size=N).T
```

The learning algorithm is essentially an optimization over the parameter space (i.e. the space of tuples of stochastic arrays having the proper dimensions) with respect to the objective function $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$. The algorithm is guaranteed to increase the objective function at each iteration, so it is sure to converge. However, the objective function is riddled with local maxima, and so the outcome depends heavily on the randomly selected starting values for $\boldsymbol{\pi}$, A, and B. Figure 12.1 illustrates the issues involved. The log probability stays approximately constant for the first 100 iterations. This indicates that the algorithm is not exploring the parameter space enough, and the parameters found at the 100-th iteration are virtually the same as those found at the first or second iteration. After the first 100 iterations, however, the algorithm is finally able to explore more of the parameter space and hence make better progress toward increasing the objective function. The moral of the story is that you may need to train the HMM a few times, using different starting values, and then keep the model that has the highest log likelihood.

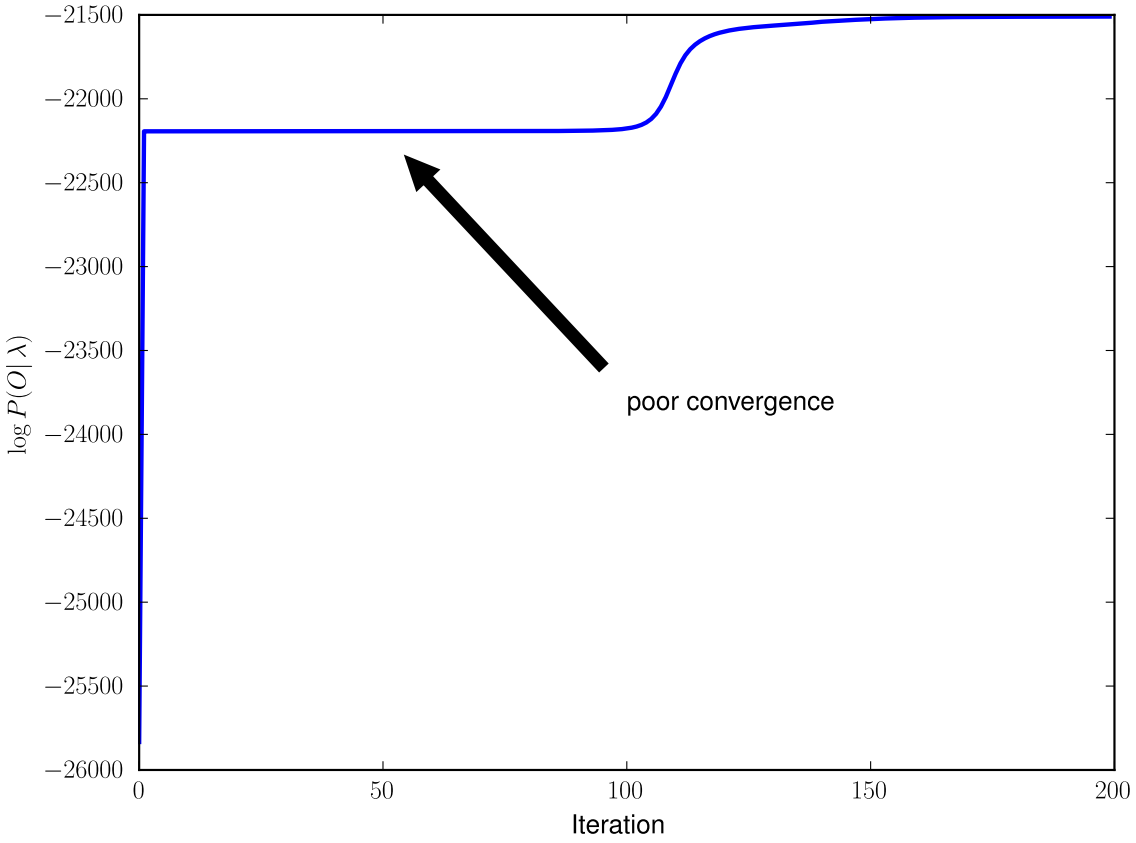


Figure 12.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

Problem 6. Implement the learning algorithm by adding the following method to your class:

```
def fit(self, z, pi, A, B, max_iter=100, tol=1e-4):
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    ----------
    z : ndarray of shape (T,)
        Observation sequence on which to train the model.
    pi : Stochastic ndarray of shape (n,)
        Initial state distribution
    A : Stochastic ndarray of shape (n, n)
        Initial state transition matrix
    B : Stochastic ndarray of shape (m, n)
        Initial state observation matrix
```

```
    max_iter : int
        The maximum number of iterations to take
    tol : float
        The convergence threshold for change in log-probability
    """
    # initialize self.pi, self.A, self.B
    # run the iteration
    pass
```

We now turn to the data found in the file `declaration.txt`. This file contains the text of the Declaration of Independence. We will use the sequence of characters (after stripping out punctuation and converting everything to lower-case) as our observation sequence. In order to convert the raw text into a useable data structure, we need to read in the file, process the string as necessary, and then map the characters to integer values. We provide helper code below to accomplish this task for various files in various languages:

```
>>> import numpy as np
>>> import string
>>> import codecs

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

>>> def prep_data(filename):
>>>     # Get the data as a single string
>>>     with codecs.open(filename, encoding='utf-8') as f:
>>>         data=f.read().lower()  # and convert to all lower case

>>>     # remove punctuation and newlines
>>>     remove_punct_map = {ord(char):
                            None for char in string.punctuation+"\n\r"}
>>>     data = data.translate(remove_punct_map)

>>>     # make a list of the symbols in the data
>>>     symbols = sorted(list(set(data)))

>>>     # convert the data to a NumPy array of symbols
>>>     a = np.array(list(data))

>>>     # make a conversion dictionary from symbols to state numbers
>>>     symbols_to_obsstates = {x:i for i,x in enumerate(symbols)}

>>>     # convert the symbols in a to state numbers
>>>     obs_sequence = vec_translate(a,symbols_to_obsstates)

>>>     return symbols, obs_sequence
```

Now apply this helper code to `declaration.txt`.

```
>>> symbols, obs = prep_data('declaration.txt')
```

Problem 7. You are now ready to train a HMM using the Declaration of Independence data. Use $N = 2$ states and $M =$`len(set(obs))`$= 27$ observation values (26 lower case characters and 1 whitespace character), and run for 150 iterations with the default value for `tol`. Generally speaking, if you converge to a log probability greater than -21550, then you have reached an acceptable set of parameters for this dataset.

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

```
>>> for i in range(len(h.B)):
>>>     print(u"{0}, {1:0.4f}, {2:0.4f}"
                .format(symbols[i], h.B[i,0], h.B[i,1]))
 , 0.0051, 0.3324
a, 0.0000, 0.1247
c, 0.0460, 0.0000
b, 0.0237, 0.0000
e, 0.0000, 0.2245
d, 0.0630, 0.0000
g, 0.0325, 0.0000
f, 0.0450, 0.0000
i, 0.0000, 0.1174
h, 0.0806, 0.0070
k, 0.0031, 0.0005
j, 0.0040, 0.0000
m, 0.0360, 0.0000
l, 0.0569, 0.0001
o, 0.0009, 0.1331
n, 0.1207, 0.0000
q, 0.0015, 0.0000
p, 0.0345, 0.0000
s, 0.1195, 0.0000
r, 0.1062, 0.0000
u, 0.0000, 0.0546
t, 0.1600, 0.0000
w, 0.0242, 0.0000
v, 0.0185, 0.0000
y, 0.0147, 0.0058
x, 0.0022, 0.0000
```

```
z, 0.0010, 0.0000
```

What do you notice about the second column of B? It seems that the HMM has detected a vowel state and a consonant state, without any prior input from an English speaker. Interestingly, the whitespace character is grouped together with the vowels. A HMM can also detect the vowel/consonant distinction in other languages.

Problem 8. Repeat the previous calculation with 3 hidden states and again with 4 hidden states. Interpret/explain your results.

Hint: with 3 hidden states, your print statement will look like the following:

```
>>> print(u"{0}, {1:0.4f}, {2:0.4f}, {2:0.4f}"
            .format(symbols[i], h.B[i,0], h.B[i,1], h.B[i,2]))
```

Now we turn to the Russian file `WarAndPeace.txt`, which is a small subset of the book *War and Peace* by Tolstoy.

```
>>> symbols, obs = prep_data('WarAndPeace.txt')
```

Problem 9. Repeat the same calculation with `WarAndPeace.txt` for 2 and 3 hidden states. Interpret/explain your results. Which Cyrillic characters appear to be vowels?

13 Speech Recognition using CDHMMs

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

13.0.1 Continuous Density Hidden Markov Models

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

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

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

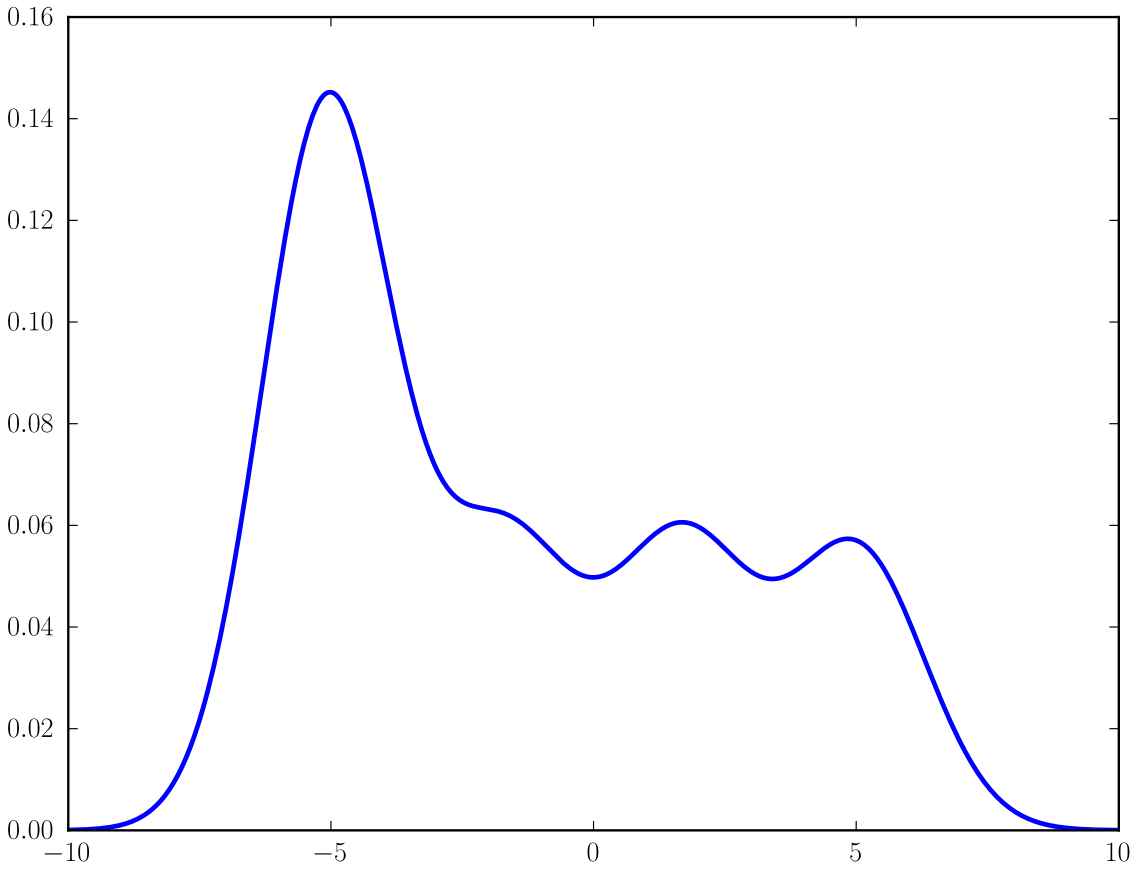


Figure 13.1: The probability density function of a mixture of Gaussians with four components.

where $N(\ \cdot\ ; \mu_k, \Sigma_k)$ denotes the probability density function for the normal distribution $\mathcal{N}(\mu_k, \Sigma_k)$. See Figure 13.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

Similar to discrete HMMs, GMMHMMs seek to model a hidden state sequence $\{X_1, \ldots, X_T\}$ and a corresponding observation sequence $\{Z_1, \ldots, Z_T\}$ where T is the number of time steps or number of observations. The major difference is that each observation $\mathbf{z}_t$ is a real-valued vector of length M distributed according to a mixture of Gaussians with K components. The parameters for such a model include the initial state distribution π and the state transition matrix A (just as with discrete HMMs). Additionally, for each state $i = 1, \ldots, N$, we have component weights $\{c_{i,1}, \ldots, c_{i,K}\}$, component means $\{\mu_{i,1}, \ldots, \mu_{i,K}\}$, and component covariance matrices $\{\Sigma_{i,1}, \ldots, \Sigma_{i,K}\}$.

Let's define a full GMMHMM with $N = 3$ states, components of dimension $M = 2$, and $K = 5$ components.

```
>>> import numpy as np

# 3x3 transition matrix
>>> A = np.array([[.6, .3, .1], [.2, .3, .5], [.7, .1, .2]])
```

```
# 3x5 collection of component weights
>>> weights = np.array([[.5, .1, .25, .09, 0.6], [0, .4, .3, .2, .1], [.1, .3, ↩
    .2, .1, .3]])

# 3x5x2 collection of component means
>>> means = np.array([np.floor(np.random.uniform(-20, 20, size = (5, 2))) for i↩
     in range(3)])

# 3x5x(2x2) collection of component covariance matrices
>>> covars = np.array([[np.floor(np.random.uniform(1, 20))*np.eye(2) for i in ↩
    range(5)] for j in range(3)])

# (3,) ndarray initial state distribution
>>> pi = np.array([.4, .1, .5])

# Save the model parameters
>>> gmm = [A, weights, means, covars, pi]
```

Once we have a GMMHMM, we can randomly choose the first state based on the initial state distribution π. Now we can iteratively sample from our GMMHMM as follows:

- Randomly select a GMM Gaussian component according to the probability weights of the current state.
- Sample from the selected GMM Gaussian component using the corresponding mean and covariance matrix.
- Obtain the next state using the transition matrix A.

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

# steps to randomly sample from GMMHMM
# randomly select a component using the probability weights of the current ↩
    state
>>> sample_component = np.argmax(np.random.multinomial(1, weights[state,:]))

# sample an observation from the selected GMM Gaussian component
>>> sample = np.random.multivariate_normal(means[state, sample_component, :], ↩
    covars[state, sample_component, :, :])

# obtain the next state using the transition matrix
>>> state = np.argmax(np.random.multinomial(1, A[:, state]))
```

Figure 13.2 shows an observation sequence generated from a GMMHMM with two mixture component and two states.

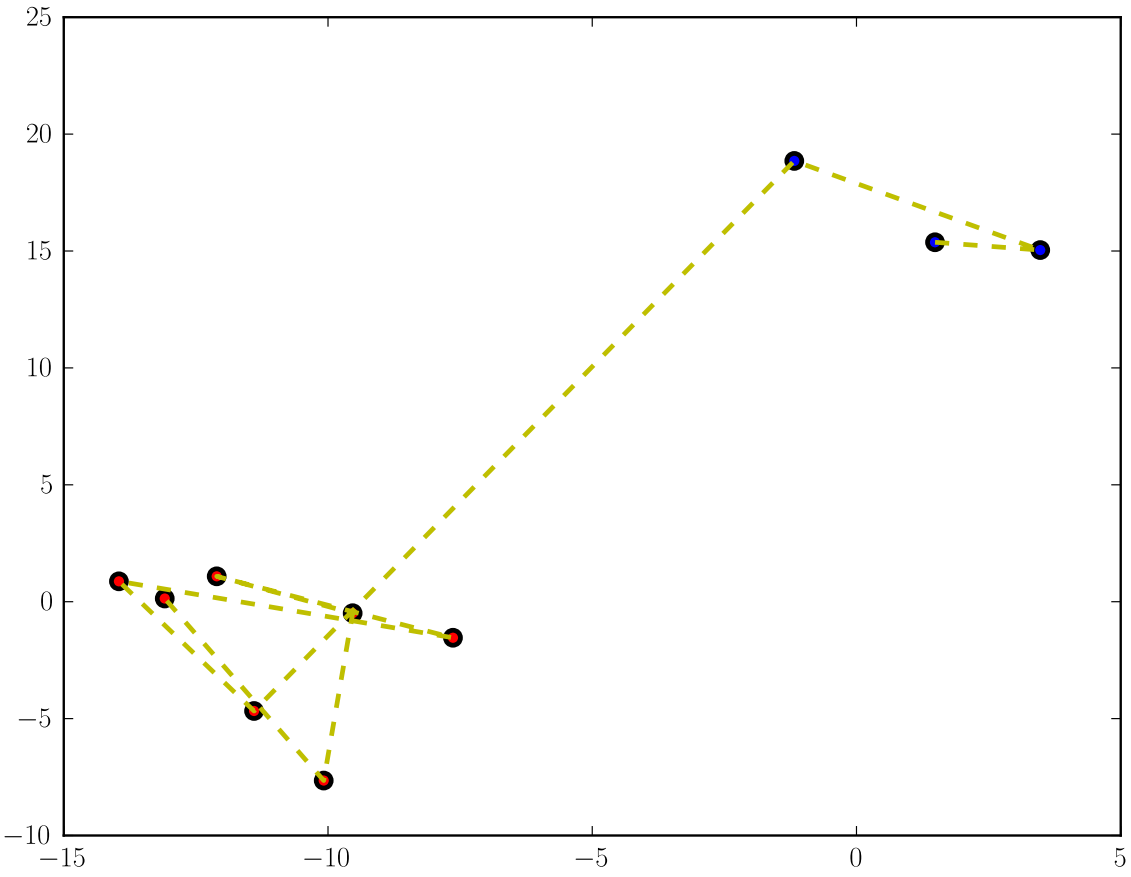


Figure 13.2: An observation sequence generated from a GMMHMM with two mixture components and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

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

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

    Returns
    -------
    states : ndarray of shape (T,)
        The sequence of states
    obs : ndarray of shape (T, M)
        The generated observations
```

```
    """
    pass
```

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

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

Speech Recognition and Hidden Markov Models

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

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

Industrial-grade speech recognition systems do not train a GMMHMM for each word in a vocabulary (that would be ludicrous for a large vocabulary), but rather on *phonemes*, or distinct sounds. The English language has 44 phonemes, yielding 44 different GMMHMMs. As you could imagine, this greatly facilitates the problem of speech recognition. Each and every word can be represented by some combination of these 44 distinct sounds. By correctly classifying a signal by its phonemes, we can determine what word was spoken. Doing so is beyond the scope of this lab, so we will simply train GMMHMMs on five words/phrases: biology, mathematics, political science, psychology, and statistics.

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

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

To load and extract, use the following code:

```
>>> samplerate, data = wavfile.read(file) # load wavfile
>>> model = MFCC.extract(data, show = False) # extract MFCC
```

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

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

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

```
>>> import numpy as np # Import packages
>>> from hmmlearn import hmm
>>> startprob, transmat = initialize(5) # Get probabilities and transition ↩
    matrices
>>> model = hmm.GMMHMM(n_components=5, covariance_type="diag", init_params = "↩
    mc") # Initialize model
>>> model.startprob_ = startprob # Set probabilities and transition matrices
>>> model.transmat_ = transmat
>>> data = train_samples[word] # Reshape data for hmmlearns fit method
>>> lengths = [data[0].shape[0]] * len(data)
>>> data_collected = np.vstack(data)
>>> model.fit(data_collected) # Fit the model
>>> model.monitor_.history[-1] # Check the score
```

Achtung!

The process for problem 3 could take up to a couple of hours. Since you will not want to run this more than once, you may want to save the best model for each word to disk using the pickle module so you can use it later.

```
>>> import pickle
>>> temp = {word: best_model}
>>> with open(word, "wb") as out:
...     pickle.dump(temp, out)
```

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

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

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

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

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

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

14 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

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + G_k\mathbf{u}_k + \mathbf{w}_k \tag{14.1}$$

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

$$\mathbf{z}_k = H_k\mathbf{x}_k + \mathbf{v}_k \tag{14.2}$$

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

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

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

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

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

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

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

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

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

How does the system evolve from one time step to the next? Assuming each time step is 0.1 seconds, it is easy enough to calculate the new position:

$$s_x' = s_x + 0.1V_x$$
$$s_y' = s_y + 0.1V_y.$$

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

$$V_x' = V_x.$$

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

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

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

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

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

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

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

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

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

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

Instantiate a `KalmanFilter` object with these values.

We now wish to simulate a sequence of states and observations from the dynamical system. In addition to the system parameters, we need an initial state $\mathbf{x}_0$ to get started. Computing the subsequent states and observations is simply a matter of following equations 14.1 and 14.2.

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

```
def evolve(self,x0,N):
    """
```

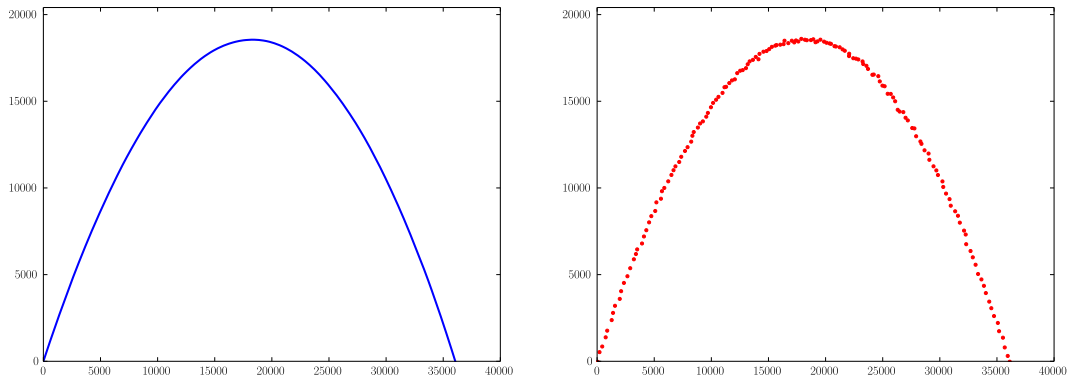


Figure 14.1: State sequence (left) and sampling of observation sequence (right).

```
    Compute the first N states and observations generated by the Kalman ↩
        system.

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

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

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

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

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the y coordinate to return to 0). Your results should qualitatively match those given in Figure 14.1.

State Estimation with the Kalman Filter

The Kalman filter is a recursive estimator that smooths out the noise in real time, estimating each current state based on the past state estimate and the current measurement. This process is done by repeatedly invoking two steps: Predict and Update. The predict step is used to estimate the current state based on the previous state. The update step then combines this prediction with the current observation, yielding a more robust estimate of the current state.

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

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

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

We evolve the filter recursively, as follows:

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

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

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

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

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

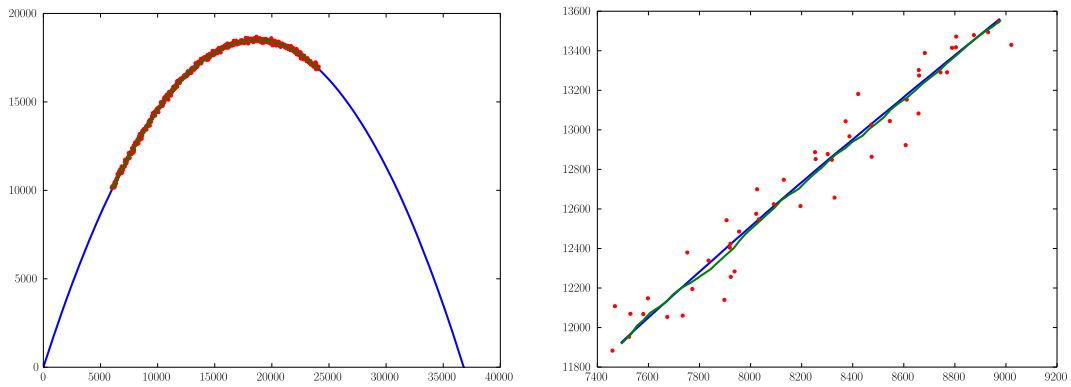


Figure 14.2: State estimates together with observations and true state sequence (detailed view on the right).

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

Returning to the projectile example, we now assume that our radar sensor has taken observations from time steps 200 through 800 (take the corresponding slice of the observations produced in Problem 3). Using these observations, we seek to estimate the corresponding true states of the projectile. We must first come up with a state estimate $\widehat{\mathbf{x}}_{200}$ 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 14.2.

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

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

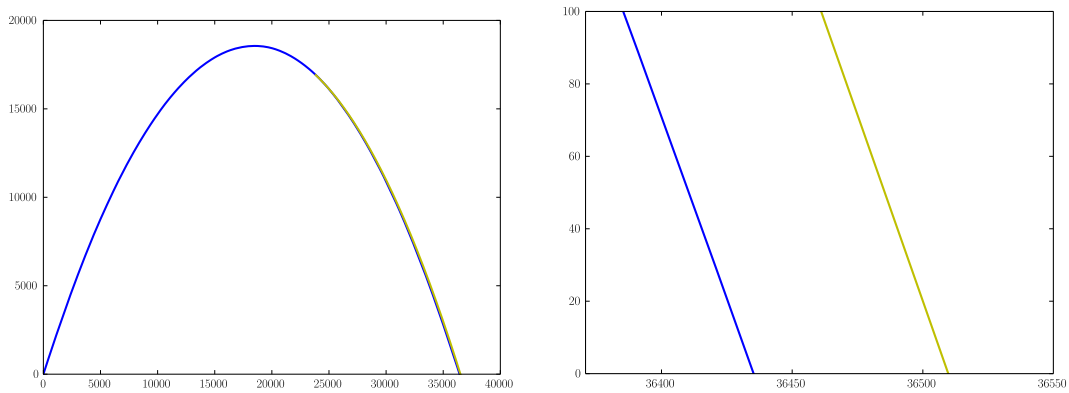


Figure 14.3: Predicted vs. actual point of impact (detailed view on right).

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

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

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

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

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

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

Plot the actual state sequence together with the predicted state sequence (as a yellow curve), and observe how near the prediction is to the actual point of impact. Your results should be similar to those shown in Figure 14.3.

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

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

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

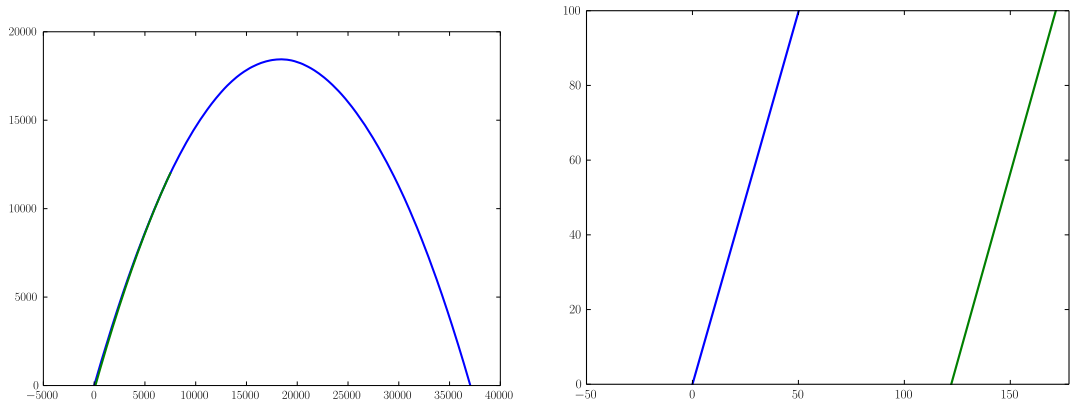


Figure 14.4: Predicted vs. actual point of origin (detailed view on right).

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

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

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

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

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

Problem 9. Using your state estimate $\widehat{\mathbf{x}}_{250}$, predict the point of origin of the projectile along with all states leading up to time step 250. Note that you may have to take a few extra time steps to predict the point of origin. (The point of origin is the first point along the trajectory where the y coordinate is 0.) Plot these predicted states (in cyan) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 14.4.

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

15 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

$$Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i Z_{t-i}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} + \boldsymbol{\varepsilon_t} \tag{15.1}$$

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:

$$Z_t = \mathbf{c} + \sum_{i=1}^{p} \Phi_i Z_{t-i} + \boldsymbol{\varepsilon_t} \tag{15.2}$$

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.

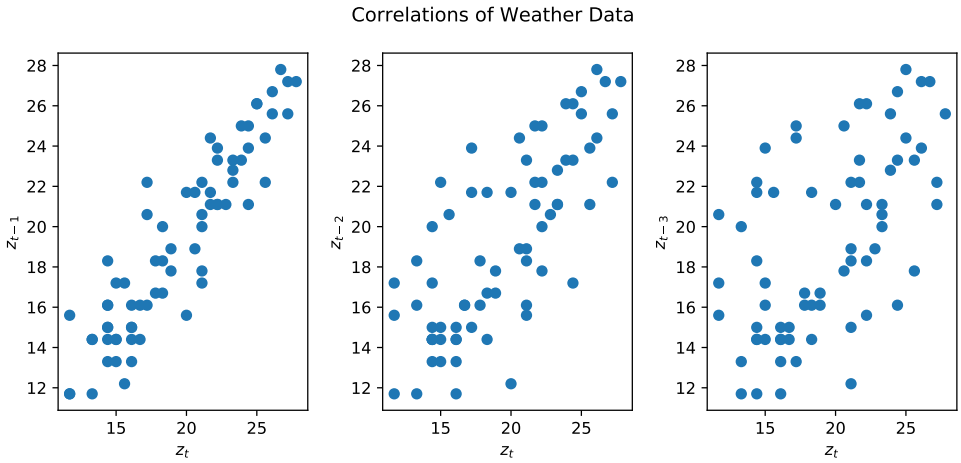


Figure 15.1: These graphs show that the weather data is correlated to its previous values. The correlation is weaker in each graph successively, showing that the further in the past the data is, the less correlated the data becomes.

MA(q) Models

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

$$Z_t = \mathbf{c} + \boldsymbol{\varepsilon_t} + \sum_{i=1}^{q} \Theta_i \boldsymbol{\varepsilon_{t-i}} \tag{15.3}$$

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

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

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

ARIMA(p, d, q) Models

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

$$\delta^d Z_t = \mathbf{c} + \underbrace{\left(\sum_{i=1}^{p} \Phi_i \mathbf{y_{t-i}}\right)}_{\text{AR(p)}} + \underbrace{\left(\sum_{j=1}^{q} \Theta_j \boldsymbol{\varepsilon_{t-j}}\right)}_{\text{MA(q)}} + \boldsymbol{\varepsilon_t} \tag{15.4}$$

Finding Parameters

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

A naive way to use an ARMA model is to choose p and q based on intuition. Figure 15.1 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 15.1, 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.

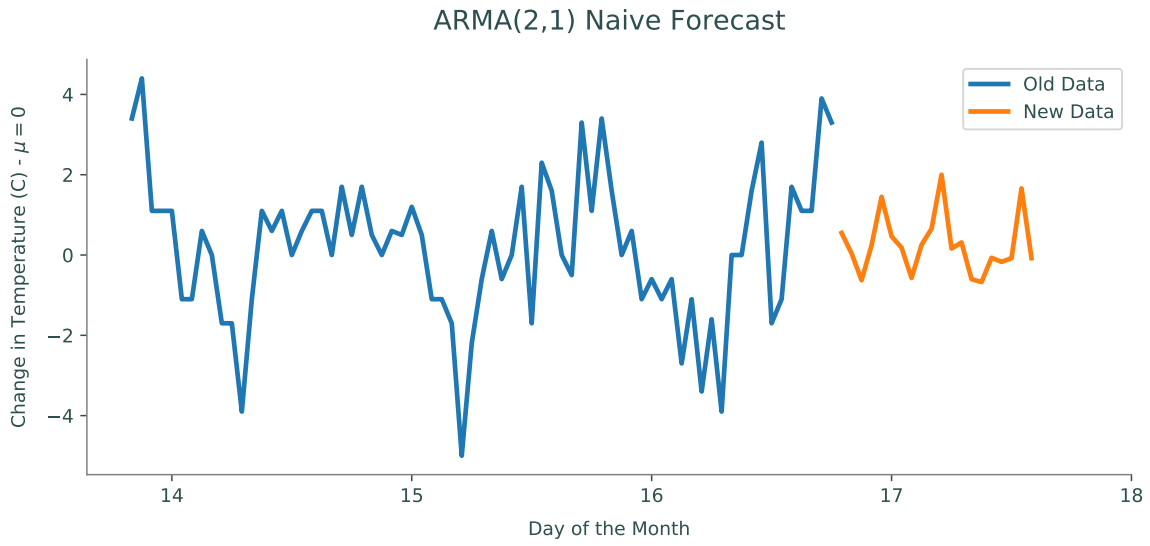


Figure 15.2: Naive forecast on `weather.npy`

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

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

Hint: you might find `np.diff()` to be usefull.

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

Use this file to test your code. For $p = 2$, $q = 1$, and $n = 20$, your plot should look similar to Figure 15.2, however, due to the variance of the error ε_t, the plot will not look exactly like Figure 15.2. The predictions may be higher or lower on the x-axis.

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

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} p(z_t|z_{t-1}, \dots, z_1, \Theta) \tag{15.5}$$

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

$$\hat{\mathbf{x}}_{t|t-1} = \begin{bmatrix} x_{t-1} & x_{t-2} & \cdots & x_{t-r} \end{bmatrix}^T \tag{15.6}$$

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{15.7}$$

$$H = \begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_{r-1} \end{bmatrix} \tag{15.8}$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{15.9}$$

$$w_t \sim \text{MVN}(0, Q), \tag{15.10}$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Note that Equation 15.2 gives

$$F\hat{\mathbf{x}}_{t-1|t-2} + w_t = \begin{bmatrix} \sum_{i=1}^{r} \phi_i x_{t-i} \\ x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-(r-1)} \end{bmatrix} + \begin{bmatrix} \varepsilon_t \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{15.11}$$

$$= \begin{bmatrix} x_t & x_{t-1} & \cdots & x_{t-(r-1)} \end{bmatrix}^T \tag{15.12}$$

$$= \hat{\mathbf{x}}_{t|t-1} \tag{15.13}$$

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

Then the linear stochastic dynamical system

$$\hat{\mathbf{x}}_{t+1|t} = F\hat{\mathbf{x}}_{t|t-1} + w_t \tag{15.14}$$

$$z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{15.15}$$

describes the same process as the original ARMA model.

NOTE

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

Equation 15.15 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 15.10 that the error terms of the model are Gaussian. This means that each conditional distribution in 15.5 is also Gaussian, and is completely characterized by its mean and variance. These two quantities are easily found via the Kalman filter:

$$\text{mean} \quad H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{15.16}$$

$$\text{variance} \quad HP_{t|t-1}H^T \tag{15.17}$$

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

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} N(z_t \mid H\hat{\mathbf{x}}_{t|t-1} + \mu,\ HP_{t|t-1}H^T) \tag{15.18}$$

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 the first problem. Return the log-likelihood of the ARMA(p, q) model as a `float`.

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

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

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

```
>>> arma_likelihood(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

$$2k\left(1 + \frac{k+1}{n-k}\right) - 2\ell(\Theta) \tag{15.19}$$

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 15.18 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. Return each parameter in Θ that minimizes the AIC of an ARMA(i, j) model, given that $1 \leq i \leq$ `p_max` and $1 \leq j \leq$ `q_max`.

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

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

$$z_{t+k}|z_1, \cdots, z_t \sim N(z_{t+k};\ H\hat{x}_{t+k|t} + \mu,\ HP_{t+k|t}H^T) \tag{15.20}$$

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

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

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

ACHTUNG!

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

Problem 4. Write a function `arma_forecast()` that accepts `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 a Kalman filter. Plot the original observations as well as the `mean` for each future observation. Plot a 95% confidence interval (2 standard deviations away from the mean) around the means of future observations. Return the means and standard deviations calculated. Hint: the standard deviation is the square root of the covariance calculated.

The following code should create a plot similar to Figure 15.3.

```
>>> # 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 plot compare to the naive ARMA model made in Problem 1?

Statsmodel ARMA

The module `statsmodels` contains a package that includes an ARMA model class. This is accessed through ARIMA model, which stands for Autoregressive Integrated Moving Average. This class also uses a Kalman Filter to calculate the MLE. When creating an ARIMA object, initialize the variables `endog` (the data) and `order` (the order of the model). The order is of the form (p, d, q) where d is the differences. To create an ARMA model, set $d = 0$. The object can then be fitted based on the MLE using a Kalman Filter.

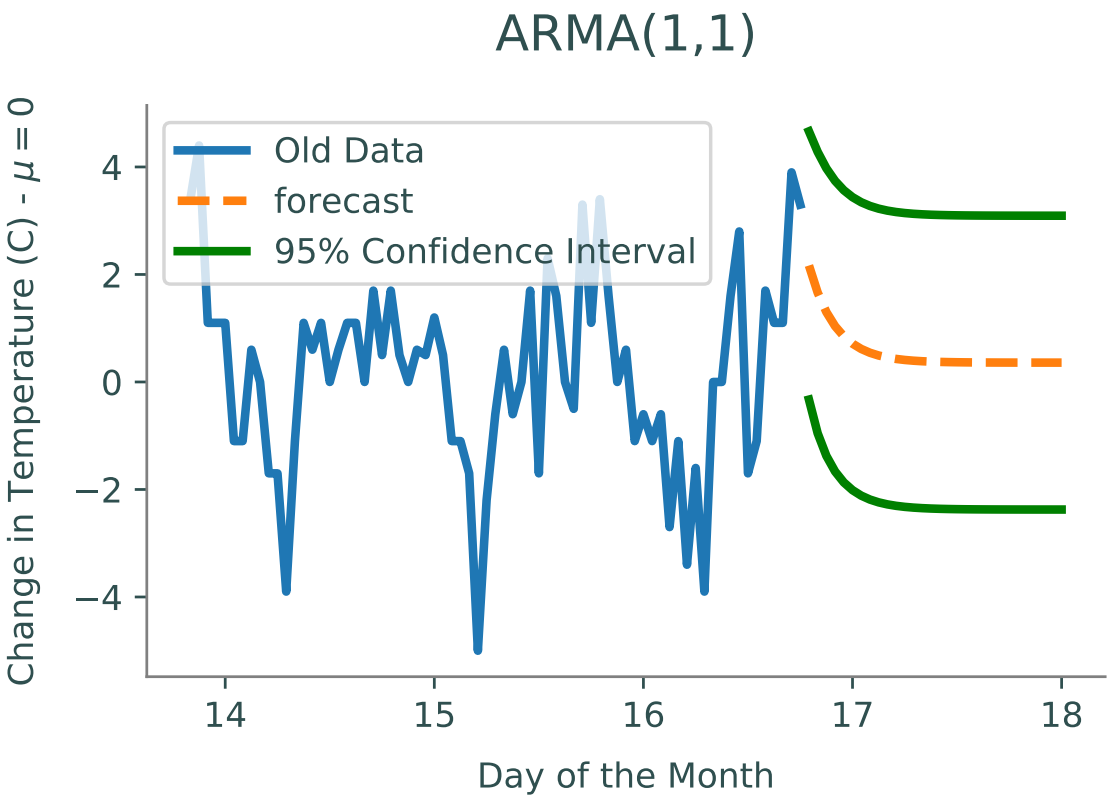


Figure 15.3: ARMA(1,1) forecast on `weather.npy`

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

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

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

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

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

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

For `p_max` $= 3$, `q_max` $= 3$, and $n = 30$, your graph should look similar to Figure 15.4. How does this graph compare to Problem 1? Problem 4?

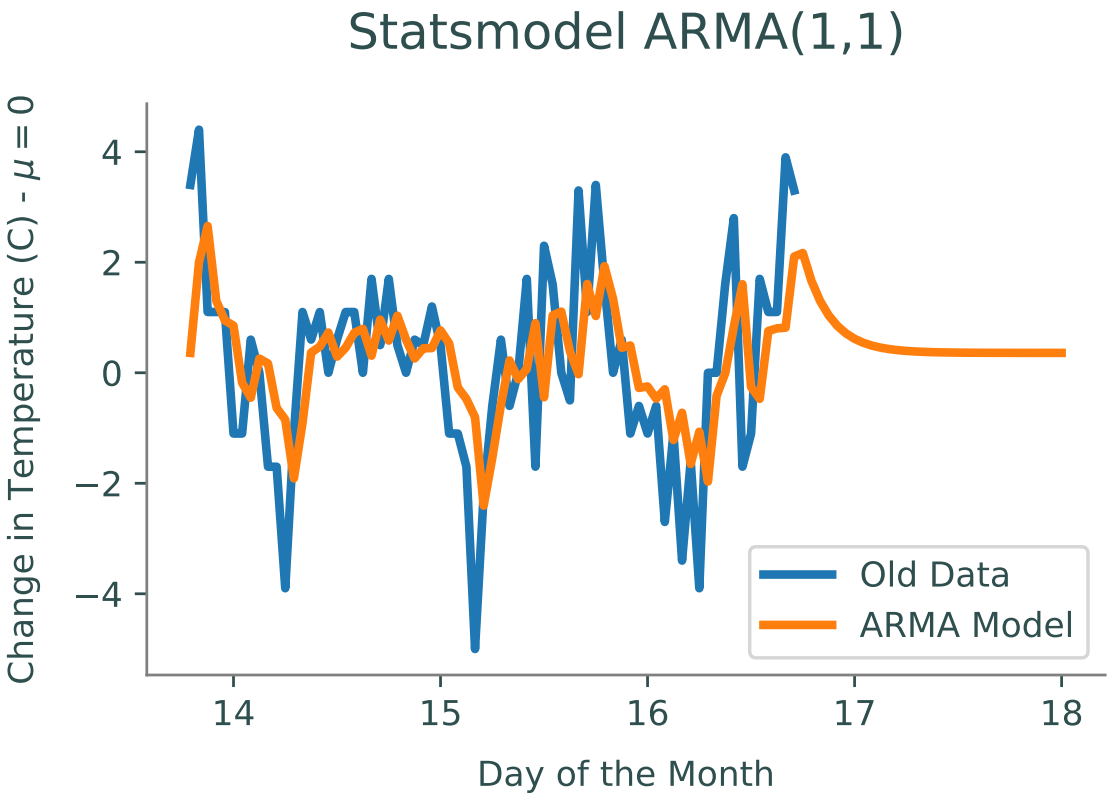


Figure 15.4: Statsmodel ARMA(3,3) forecast on `weather.npy`.

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:

$$z_{t,1} = c_1 + \phi_{11} z_{t-1,1} + \phi_{12} z_{t-1,2} + \theta_{11} \varepsilon_{t-1,1} + \theta_{12} \varepsilon_{t-1,2} \tag{15.21}$$

$$z_{t,2} = c_1 + \phi_{21} z_{t-1,1} + \phi_{22} z_{t-1,2} + \theta_{21} \varepsilon_{t-1,1} + \theta_{22} \varepsilon_{t-1,2} \tag{15.22}$$

This can be written in matrix form as shown in equation 15.1. The module `statsmodels` contains a package that includes an VARMAX model class which can be used to create a multivariate ARMA model. This stands for Vector Autoregression Moving Average with Exogenous Regressors. An exogenous regressor is a time series that affects the model but is not affected by it. In the example below we have two time series corresponding to the price of copper and aluminum. Since aluminum is a substitute for copper, it is reasonable to assume the price of aluminum may help us predict the price of copper and vice versa. Note that when fitting a VARMAX model setting the parameter `ic` to `aic` selects parameters based on AIC criterion.

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

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

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

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

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

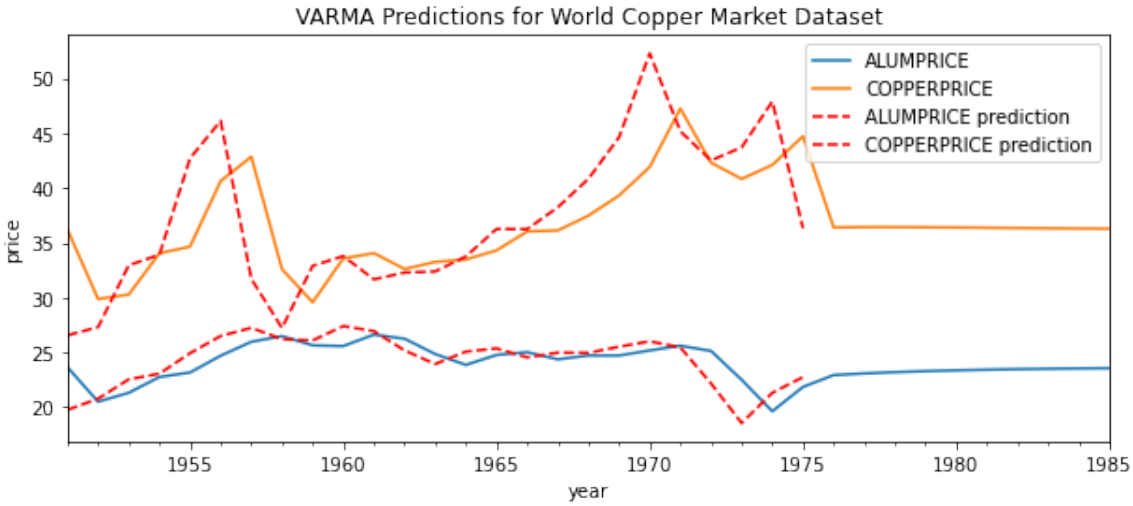


Figure 15.5: Statsmodel VAR(1) forecast.

Problem 6. Write a function `sm_varma()` that accepts start and end dates for forecasting. Use the statsmodels VARMAX class to forecast on macroeconomic data between the start and end dates. Use AIC as the criterion for model selection when fitting the model. Plot the prediction, original data and a 95% confidence interval (2 standard deviations away from the mean) around the future observations. Return the AIC of the chosen model. The plot should be similar to Figure 15.6.

The following code shows how to obtain the data.

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

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.

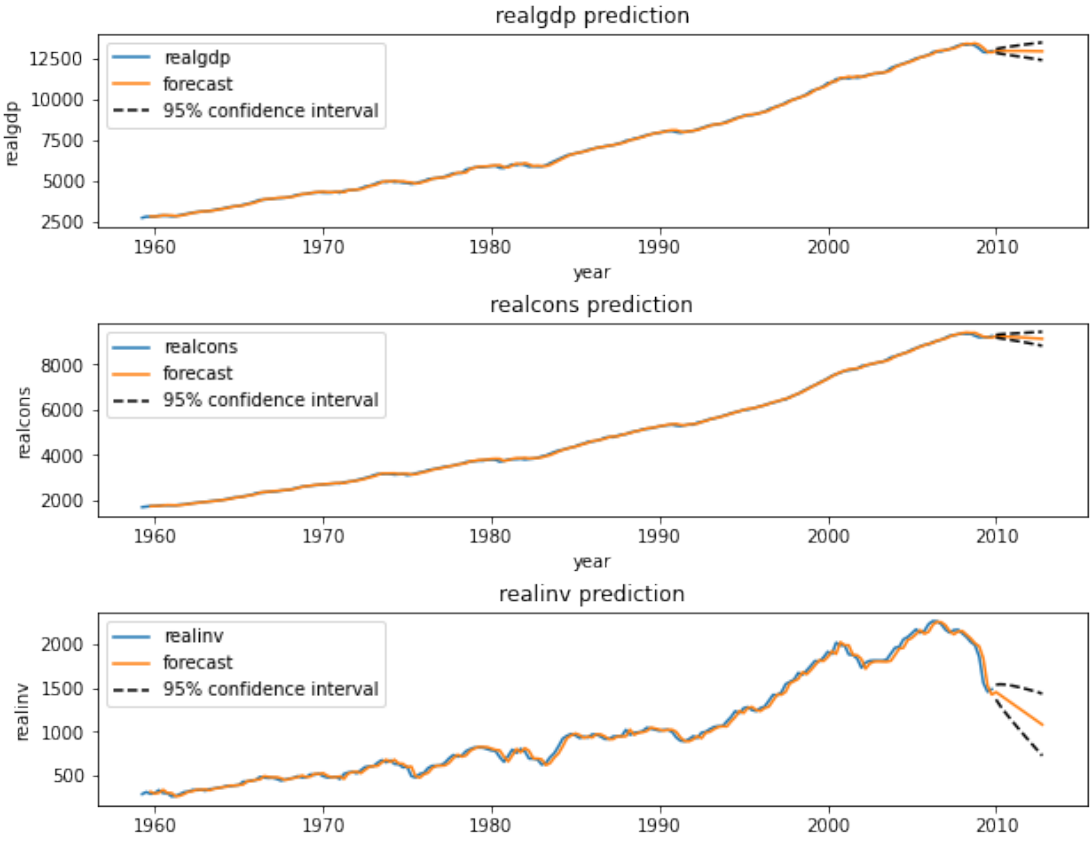


Figure 15.6: Macroeconomic data is forecasted 12 years in the future using statsmodels.

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()
```

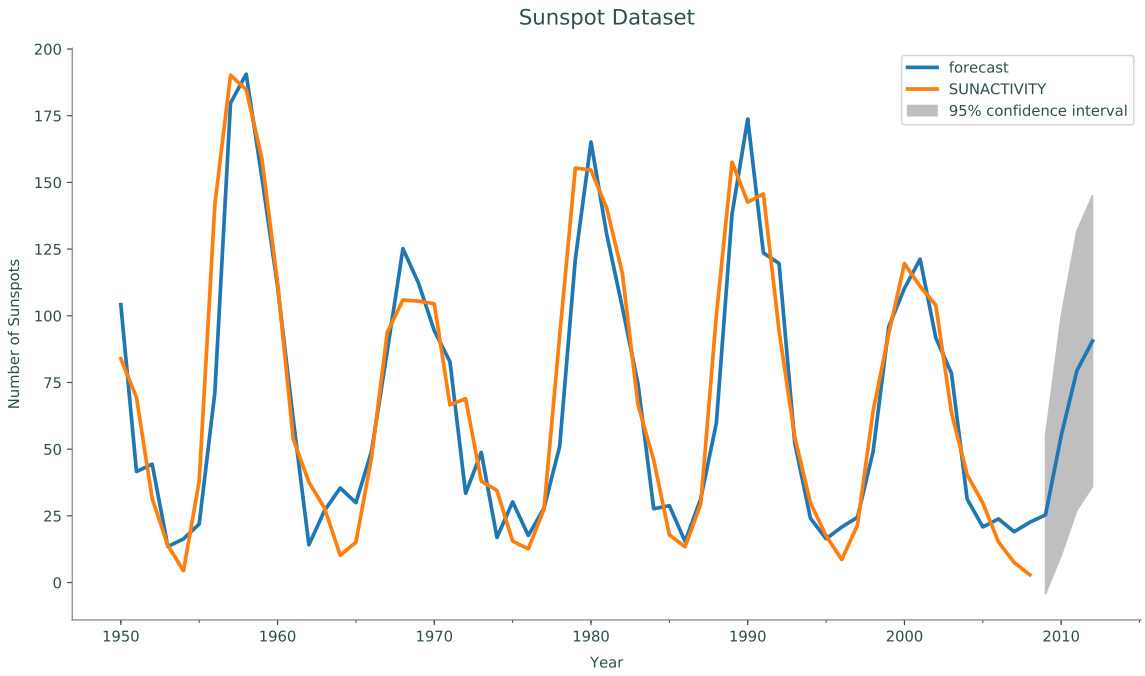


Figure 15.7: Sunspot activity data is forecasted four years in the future using `statsmodels`.

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 15.8.

(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']
```

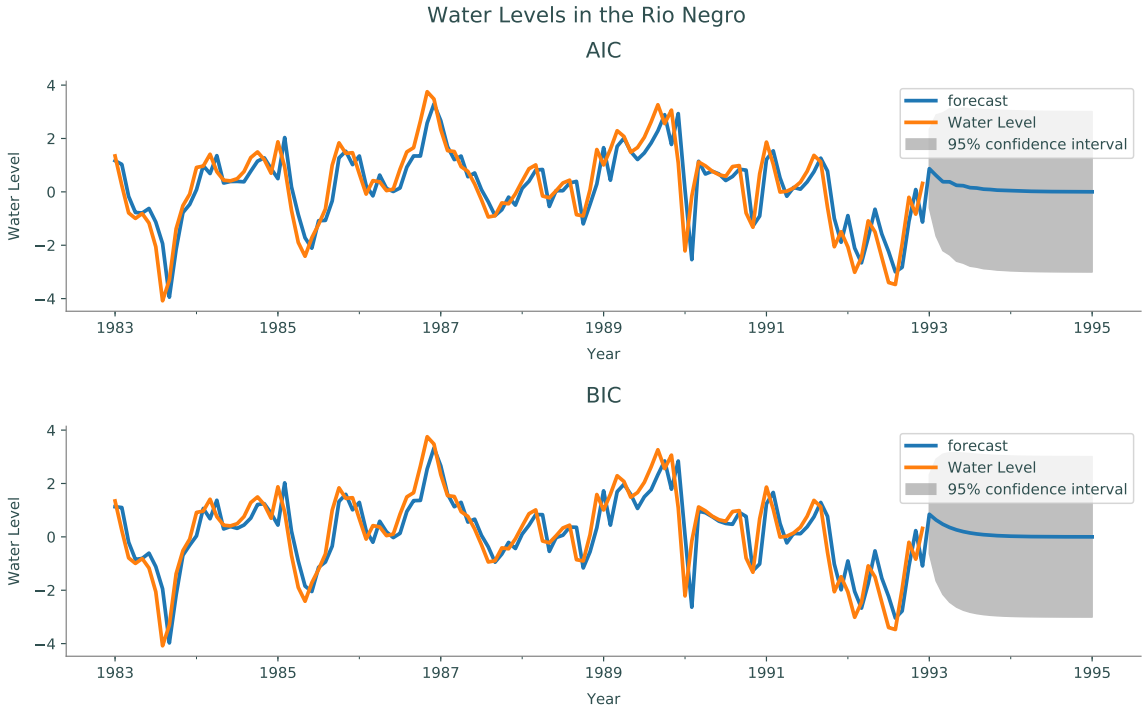


Figure 15.8: AIC and BIC based ARMA models of `manaus` dataset.

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

$$y_t = T_t + S_t + R_t, \tag{15.23}$$

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

$$T_t = \frac{1}{M} \sum_{-M/2<i<M/2} y_{i+t}. \tag{15.24}$$

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,

$$S_t = \frac{1}{K} \sum_{i \equiv t \pmod{M}} x_i \tag{15.25}$$

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 15.1.

Proof of Equation 15.15

$$\sum_{i=1}^{p} \phi_i(z_{t-i} - \mu) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} = \sum_{i=1}^{p} \phi_i(H\hat{\mathbf{x}}_{t-i}) + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} \tag{15.26}$$

$$= \sum_{i=1}^{r} \phi_i(x_{t-i} + \sum_{k=1}^{r-1} \theta_k x_{t-i-k}) + a_t + \sum_{j=1}^{r-1} \theta_j a_{t-j} \tag{15.27}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j \Big(\sum_{i=1}^{r} \phi_i x_{t-j-i} + a_{t-j} \Big) \tag{15.28}$$

$$= a_t + \sum_{i=1}^{r} \phi_i(x_{t-i}) + \sum_{j=1}^{r-1} \theta_j x_{t-k} \tag{15.29}$$

$$= x_t + \sum_{j=1}^{r-1} \theta_j x_{t-k} \theta_k x_{t-k} \tag{15.30}$$

$$= z_t. \tag{15.31}$$

16 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.

$$H_{ij}^{s+1} = H_{ij}^{s}\frac{((W^s)^T V)_{ij}}{((W^s)^T W^s H^s)_{ij}} \tag{16.1}$$

and

$$W_{ij}^{s+1} = W_{ij}^{s}\frac{(V(H^{s+1})^T)_{ij}}{(W^s H^{s+1}(H^{s+1})^T)_{ij}} \tag{16.2}$$

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 16.1 and 16.2.

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.

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.

$$V = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 2 \\ 2 & 3 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 2 & 3 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{matrix}$$

After performing NMF on V, we'll get the following W and H.

$$W = \begin{array}{c} \begin{matrix} Component1 & Component2 & Component3 \end{matrix} \\ \begin{pmatrix} 2.1 & 0.03 & 0. \\ 1.17 & 0.19 & 1.76 \\ 0.43 & 0.03 & 0.89 \\ 0.26 & 2.05 & 0.02 \\ 0.45 & 0. & 0. \end{pmatrix} \end{array} \begin{matrix} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{matrix}$$

$$H = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0.00 & 0.45 & 0.00 & 0.43 & 1.0 & 0.9 \\ 0.00 & 0.91 & 1.45 & 1.9 & 0.35 & 0.37 \\ 1.14 & 1.22 & 0.55 & 0.0 & 0.47 & 0.53 \end{pmatrix} \end{array} \begin{matrix} Component1 \\ Component2 \\ Component3 \end{matrix}$$

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.

$$WH = \begin{array}{c} \begin{matrix} John & Alice & Mary & Greg & Peter & Jennifer \end{matrix} \\ \begin{pmatrix} 0.0000 & 0.9723 & 0.0435 & 0.96 & 2.1105 & 1.9011 \\ 2.0064 & 2.8466 & 1.2435 & 0.8641 & 2.0637 & 2.0561 \\ 1.0146 & 1.3066 & 0.533 & 0.2419 & 0.8588 & 0.8698 \\ 0.0228 & 2.0069 & 2.9835 & 4.0068 & 0.9869 & 1.0031 \\ 0.0000 & 0.2025 & 0.0000 & 0.1935 & 0.45 & 0.405 \end{pmatrix} \end{array} \begin{matrix} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{matrix}$$

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 $rank = 3, 4, 5, \ldots$. For each iteration, run NMF using `n_components`=rank and reconstruct the matrix V. Calculate the root mean square error using `sklearn.metrics.mean_squared_error` of the original dataframe and the reconstructed matrix V. When the RMSE is less than the benchmark value, stop. Return the rank and the reconstructed matrix of this rank.

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']

A Getting Started

The labs in this curriculum aim to introduce computational and mathematical concepts, walk through implementations of those concepts in Python, and use industrial-grade code to solve interesting, relevant problems. Lab assignments are usually about 5–10 pages long and include code examples (yellow boxes), important notes (green boxes), warnings about common errors (red boxes), and about 3–7 exercises (blue boxes). Get started by downloading the lab manual(s) for your course from `http://foundations-of-applied-mathematics.github.io/`.

Submitting Assignments

Labs

Every lab has a corresponding specifications file with some code to get you started and to make your submission compatible with automated test drivers. Like the lab manuals, these materials are hosted at `http://foundations-of-applied-mathematics.github.io/`.

Download the `.zip` file for your course, unzip the folder, and move it somewhere where it won't get lost. This folder has some setup scripts and a collection of folders, one per lab, each of which contains the specifications file(s) for that lab. See `Student-Materials/wiki/Lab-Index` for the complete list of labs, their specifications and data files, and the manual that each lab belongs to.

> ACHTUNG!
>
> Do **not** move or rename the lab folders or the enclosed specifications files; if you do, the test drivers will not be able to find your assignment. Make sure your folder and file names match `Student-Materials/wiki/Lab-Index`.

To submit a lab, modify the provided specifications file and use the file-sharing program specified by your instructor (discussed in the next section). The instructor will drop feedback files in the lab folder after grading the assignment. For example, the Introduction to Python lab has the specifications file `PythonIntro/python_intro.py`. To complete that assignment, modify `PythonIntro/python_intro.py` and submit it via your instructor's file-sharing system. After grading, the instructor will create a file called `PythonIntro/PythonIntro_feedback.txt` with your score and some feedback.

Homework

Non-lab coding homework should be placed in the `_Homework/` folder and submitted like a lab assignment. Be careful to name your assignment correctly so the instructor (and test driver) can find it. The instructor may drop specifications files and/or feedback files in this folder as well.

Setup

> **ACHTUNG!**
>
> We strongly recommend using a Unix-based operating system (Mac or Linux) for the labs. Unix has a true bash terminal, works well with git and python, and is the preferred platform for computational and data scientists. It is possible to do this curriculum with Windows, but expect some road bumps along the way.

There are two ways to submit code to the instructor: with git (`http://git-scm.com/`), or with a file-syncing service like Google Drive. Your instructor will indicate which system to use.

Setup With Git

Git is a program that manages updates between an online code repository and the copies of the repository, called *clones*, stored locally on computers. If git is not already installed on your computer, download it at `http://git-scm.com/downloads`. If you have never used git, you might want to read a few of the following resources.

- Official git tutorial: `https://git-scm.com/docs/gittutorial`
- Bitbucket git tutorials: `https://www.atlassian.com/git/tutorials`
- GitHub git cheat sheet: `services.github.com/.../github-git-cheat-sheet.pdf`
- GitLab git tutorial: `https://docs.gitlab.com/ce/gitlab-basics/start-using-git.html`
- Codecademy git lesson: `https://www.codecademy.com/learn/learn-git`
- Training video series by GitHub: `https://www.youtube.com/playlist?list=PLg7.../`

There are many websites for hosting online git repositories. Your instructor will indicate which web service to use, but we only include instructions here for setup with Bitbucket.

1. *Sign up.* Create a Bitbucket account at `https://bitbucket.org`. If you use an academic email address (ending in `.edu`, etc.), you will get free unlimited public and private repositories.
2. *Make a new repository.* On the Bitbucket page, click the + button from the menu on the left and, under **CREATE**, select **Repository**. Provide a name for the repository, mark the repository as **private**, and make sure the repository type is **Git**. For **Include a README?**, select **No** (if you accidentally include a `README`, delete the repository and start over). Under **Advanced settings**, enter a short description for your repository, select **No forks** under forking, and select **Python** as the language. Finally, click the blue **Create repository** button. Take note of the URL of the webpage that is created; it should be something like `https://bitbucket.org/<name>/<repo>`.

3. *Give the instructor access to your repository.* On your newly created Bitbucket repository page (`https://bitbucket.org/<name>/<repo>` or similar), go to **Settings** in the menu to the left and select **User and group access**, the second option from the top. Enter your instructor's Bitbucket username under **Users** and click **Add**. Select the blue **Write** button so your instructor can read from and write feedback to your repository.

4. *Create an SSH key.* This step needs to be done only once on each computer that you want to be able to use to access your repository. If you have multiple repositories on the same computer, you do *not* need to repeat this step for each one. To create an SSH key, in a shell application (Terminal on Linux or Mac, or Git Bash (`https://gitforwindows.org/`) on Windows), enter the following command:

```
$ ssh-keygen
```

   Press the Enter or Return key to accept the default file location. It will then prompt to enter a passphrase; this acts as a password to use the SSH key. If you do not want a passphrase, leave it blank and press Enter again. The key will then be created. The file for the key will be placed in in the `/home/<username>/.ssh` directory on Linux; in `/Users/<username>/.ssh` on macOS; and in `/c/users/<username>/.ssh` on Windows.

   Now that the key is created, you need to add it to your Bitbucket account. From Bitbucket, choose **Personal settings** and then **SSH keys**. Click **Add key** and enter a label (what it is doesn't matter). Now, using the file explorer, navigate to the SSH key you created, and open the *public key* file. The file will be called something like `id_rsa.pub`; do *NOT* use `id_rsa` (without the `.pub` extension). Copy the contents of this file, paste it into the Key field on Bitbucket, and press Save.

   For more options and some troubleshooting information, refer to `https://support.atlassian.com/bitbucket-cloud/docs/set-up-an-ssh-key/`.

5. *Connect your folder to the new repository.* In a shell application (Terminal on Linux or Mac, or Git Bash (`https://gitforwindows.org/`) on Windows), enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder  # cd means 'change directory'.

# Make sure you are in the right place.
$ pwd                 # pwd means 'print working directory'.
/path/to/folder
$ ls *.md             # ls means 'list files'.
README.md             # This means README.md is in the working directory.

# Connect this folder to the online repository.
$ git init
$ git remote add origin git@bitbucket.org:<name>/<repo>.git

# Record your credentials.
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Add the contents of this folder to git and update the repository.
```

```
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

For example, if your Bitbucket username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is on the desktop, enter the following commands.

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

# Connect this folder to the online repository.
$ git init
$ git remote add origin git@bitbucket.org:greek314/acmev1.git

# Record credentials.
$ git config --local user.name "archimedes"
$ git config --local user.email "greek314@example.com"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

At this point you should be able to see the files on your repository page from a web browser. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line:

```
$ git remote set-url origin git@bitbucket.org:<name>/<repo>.git
```

Note

You may get the an error like the following when you run `git push`:

```
remote: Bitbucket Cloud recently stopped supporting account passwords↩
     for Git authentication.
...
fatal: Authentication failed for 'https://bitbucket.org/<name>/<repo↩
    >.git/'
```

If this error occurs, your repository URL is in the wrong format; most likely, you used the `https` version instead of what is shown above. You can use the `git remote set-url origin` command to fix this issue as well.

6. *Download data files.* Many labs have accompanying data files. To download these files, navigate to your clone and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folder for you. You can also find individual data files through `Student-Materials/wiki/Lab-Index`.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash download_data.sh
```

7. *Install Python package dependencies.* The labs require several third-party Python packages that don't come bundled with Anaconda. Run the following command to install the necessary packages.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash install_dependencies.sh
```

8. (Optional) *Clone your repository.* If you want your repository on another computer after completing steps 1–5, use the following commands.

```
# Navigate to where you want to put the folder.
$ cd ~/Desktop/or/something/

# Clone the folder from the online repository.
$ git clone git@bitbucket.org:<name>/<repo>.git <foldername>

# Record your credentials in the new folder.
$ cd <foldername>
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Download data files to the new folder.
$ bash download_data.sh
```

Setup Without Git

Even if you aren't using git to submit files, you must install it (`http://git-scm.com/downloads`) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 6 and 7 of the previous section to download the data files and install package dependencies.

Using Git

Git manages the history of a file system through *commits*, or checkpoints. Use `git status` to see the files that have been changed since the last commit. These changes are then moved to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. Save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

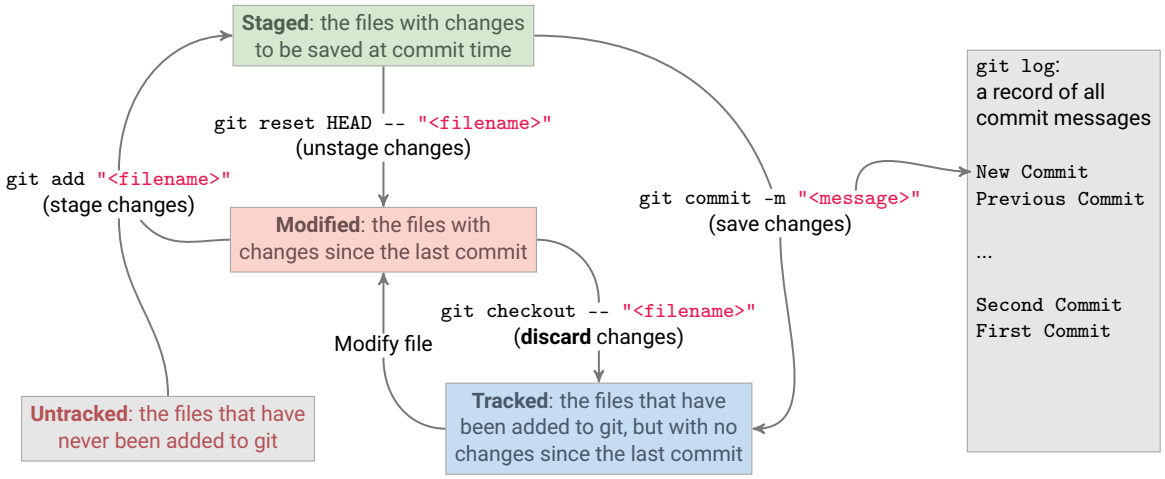


Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

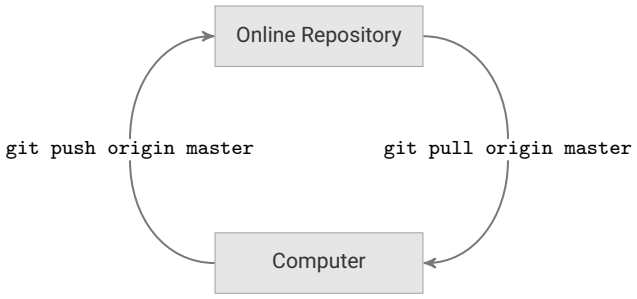


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Table A.1: Common git commands.

NOTE

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of git@bitbucket.org:<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in *vim* (`https://en.wikipedia.org/wiki/Vim_(text_editor)`), is asking you to enter a message (or use the default message) to create a *merge commit* that will reconcile both changes. To close this screen and create the merge commit, type `:wq` and press `enter`.

Example Work Sessions

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master                          # Pull updates.
### Make changes to a file.
$ git add -u                                      # Track changes.
$ git commit -m "Made some changes."              # Commit changes.
$ git push origin master                          # Push updates.
```

```
# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ git pull origin master
From bitbucket.org:username/repo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

### Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
 1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@bitbucket.org:username/repo.git
   5742a1b..fed9b34  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

B Installing and Managing Python

Lab Objective: *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to `https://www.anaconda.com/download/`.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see `https://docs.anaconda.com/anaconda/install/`. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

> ACHTUNG!
>
> This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see `https://xkcd.com/349/`).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See `https://docs.anaconda.com/anaconda/packages/pkg-docs` for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
`conda install <package-name>`	Install the specified package.
`conda update <package-name>`	Update the specified package.
`conda update conda`	Update `conda` itself.
`conda update anaconda`	Update **all** packages included in Anaconda.
`conda --help`	Display the documentation for `conda`.

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda                # Make sure that conda is up to date.
$ conda install matplotlib          # Attempt to install matplotlib.
$ conda update matplotlib           # Attempt to update matplotlib.
```

See `https://conda.io/docs/user-guide/tasks/manage-pkgs.html` for more examples.

Note

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

Achtung!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
`pip install package-name`	Install the specified package.
`pip install --upgrade package-name`	Update the specified package.
`pip freeze`	Display the version number on all installed packages.
`pip --help`	Display the documentation for `pip`.

See `https://pip.pypa.io/en/stable/user_guide/` for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: `https://atom.io/`
- Sublime Text: `https://www.sublimetext.com/`
- Notepad++ (Windows): `https://notepad-plus-plus.org/`
- Geany: `https://www.geany.org/`
- Vim: `https://www.vim.org/`
- Emacs: `https://www.gnu.org/software/emacs/`

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                        # List the files in the current directory.
hello_world.py
$ cat hello_world.py        # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py     # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

> **Note**
>
> While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:
>
> - Windows subsystem for linux: `docs.microsoft.com/en-us/windows/wsl/`.
> - Git bash: `https://gitforwindows.org/`.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See `https://github.com/jupyter/jupyter/wiki/` for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Juptyer Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and LaTeX, and can embedded images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: `http://jupyterlab.readthedocs.io/en/stable/`
- PyCharm: `https://www.jetbrains.com/pycharm/`
- Spyder: `http://code.google.com/p/spyderlib/`
- Eclipse with PyDev: `http://www.eclipse.org/`, `https://www.pydev.org/`

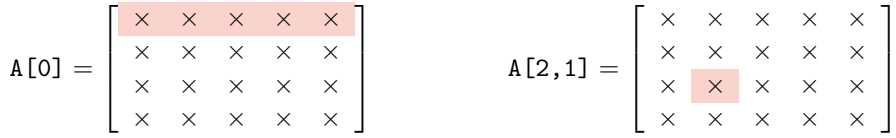
See `https://realpython.com/python-ides-code-editors-guide/` for a good overview of these (and other) workflow tools.

C 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.*

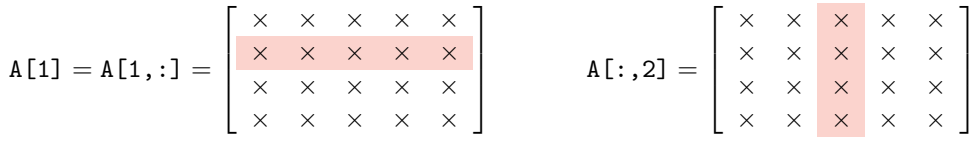
Data Access

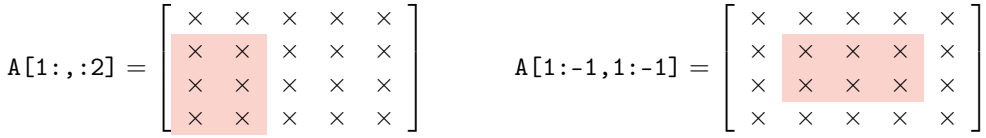
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.



Slicing

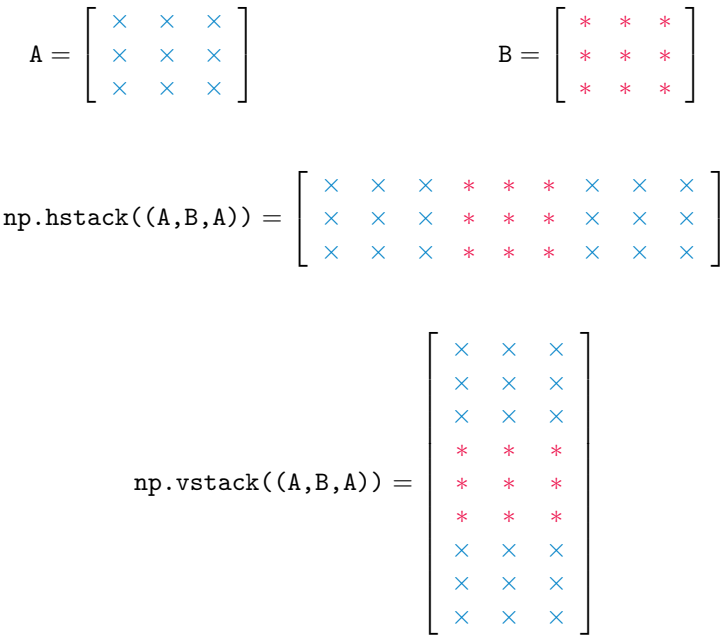
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



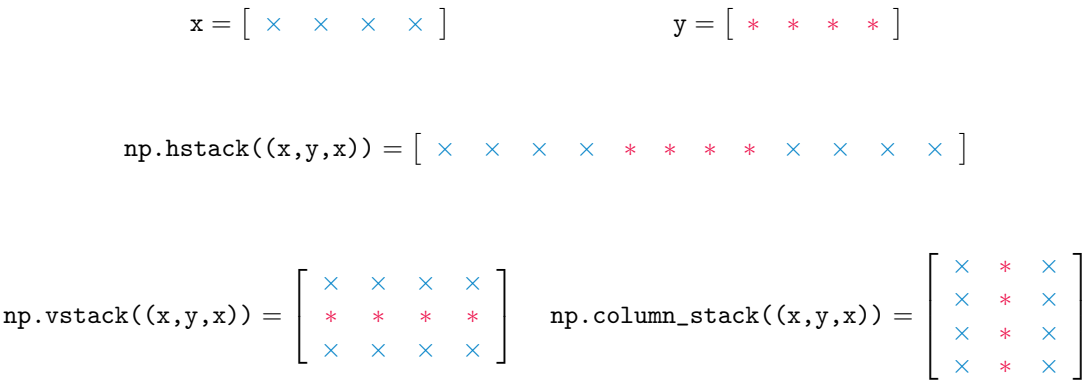


Stacking

`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`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

D Introduction to Scikit-Learn

Lab Objective: *Scikit-learn is the one of the fundamental tools in Python for machine learning. In this appendix we highlight and give examples of some popular scikit-learn tools for classification and regression, training and testing, data normalization, and constructing complex models.*

> **Note**
>
> This guide corresponds to scikit-learn version 0.20, which has a few significant differences from previous releases. See `http://scikit-learn.org/stable/whats_new.html` for current release notes. Install scikit-learn (the `sklearn` module) with `conda install scikit-learn`.

Base Classes and API

Many machine learning problems center on constructing a function $f : X \to Y$, called a *model* or *estimator*, that accurately represents properties of given data. The domain X is usually $\mathbb{R}^D$, and the range Y is typically either $\mathbb{R}$ (regression) or a subset of $\mathbb{Z}$ (classification). The model is trained on N *samples* $(\mathbf{x}_i)_{i=1}^N \subset X$ that usually (but not always) have N accompanying *labels* $(y_i)_{i=1}^N \subset Y$.

Scikit-learn [PVG+11, BLB+13] takes a highly object-oriented approach to machine learning models. Every major scikit-learn class inherits from `sklearn.base.BaseEstimator` and conforms to the following conventions:

1. The constructor `__init__()` receives *hyperparameters* for the classifier, which are parameters for the model f that are **not dependent on data**. Each hyperparameter must have a default value (i.e., every argument of `__init__()` is a keyword argument), and each argument must be saved as an instance variable of the **same name** as the parameter.

2. The `fit()` method constructs the model f. It receives an $N \times D$ matrix X and, optionally, a vector $\mathbf{y}$ with N entries. Each row $\mathbf{x}_i$ of X is one sample with corresponding label y_i. By convention, `fit()` always returns `self`.

Along with the `BaseEstimator` class, there are several other "mix in" base classes in `sklearn.base` that define specific kinds of models. The three listed below are the most common.[1]

[1] See `http://scikit-learn.org/stable/modules/classes.html#base-classes` for the complete list.

- `ClassifierMixin`: for *classifiers*, estimators that take on discrete values.
- `RegressorMixin`: for *regressors*, estimators that take on continuous values.
- `TransformerMixin`: for preprocessing data before estimation.

Classifiers and Regressors

The `ClassifierMixin` and `RegressorMixin` both require a `predict()` method that acts as the actual model f. That is, `predict()` receives an $N \times D$ matrix X and returns N predicted labels $(y_i)_{i=1}^N$, where y_i is the label corresponding to the ith row of X. Both of these base class have a predefined `score()` method that uses `predict()` to test the accuracy of the model. It accepts $N \times D$ test data and a vector of N corresponding labels, then reports either the classification accuracy (for classifiers) or the R^2 value of the regression (for regressors).

For example, a `KNeighborsClassifier` from `sklearn.neighbors` inherits from `BaseEstimator` and `ClassifierMixin`. This classifier uses a simple strategy: to classify a new piece of data $\mathbf{z}$, find the k training samples that are "nearest" to $\mathbf{z}$, then take the most common label corresponding to those nearest neighbors to be the label for $\mathbf{z}$. Its constructor accepts hyperparameters such as `n_neighbors`, for determining the number of neighbors k to search for, `algorithm`, which speficies the strategy to find the neighbors, and `n_jobs`, the number of parallel jobs to run during the neighbors search. Again, these hyperparameters are independent of any data, which is why they are set in the constructor (before fitting the model). Calling `fit()` organizes the data X into a data structure for efficient nearest neighbor searches (determined by `algorithm`). Calling `predict()` executes the search, determines the most common label of the neighbors, and returns that label.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.model_selection import train_test_split

# Load the breast cancer dataset and split it into training and testing groups.
>>> cancer = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(cancer.data,
...                                                     cancer.target)
>>> print(X_train.shape, y_train.shape)
(426, 30) (426,)        # There are 426 training points, each with 30 features.

# Train a KNeighborsClassifier object on the training data.
# fit() returns the object, so we can instantiate and train in a single line.
>>> knn = KNeighborsClassifier(n_neighbors=2).fit(X_train, y_train)
# The hyperparameter 'n_neighbors' is saved as an attribute of the same name.
>>> knn.n_neighbors
2

# Test the classifier on the testing data.
>>> knn.predict(X_test[:6])
array([0, 1, 0, 1, 1, 0])       # Predicted labels for the first 6 test points.
>>> knn.score(X_test, y_test)
0.8951048951048951              # predict() chooses 89.51% of the labels right.
```

The `KNeighborsClassifier` object could easily be replaced with a different classifier, such as a `GaussianNB` object from `sklearn.naive_bayes`. Since `GaussianNB` also inherits from `BaseEstimator` and `ClassifierMixin`, it has `fit()`, `predict()`, and `score()` methods that take in the same kinds of inputs as the corresponding methods for the `KNeighborsClassifier`. The only difference, from an external perspective, is the hyperparameters that the constructor accepts.

```
>>> from sklearn.naive_bayes import GaussianNB

>>> gnb = GaussianNB().fit(X_train, y_train)
>>> gnb.predict(X_test[:6])
array([1, 1, 0, 1, 1, 0])
>>> gnb.score(X_test, y_test)
0.9440559440559441
```

Roughly speaking, the `GaussianNB` classifier assumes all features in the data are independent and normally distributed, then uses Bayes' rule to compute the likelihood of a new point belonging to a label for each of the possible labels. To do this, the `fit()` method computes the mean and variance of each feature, grouped by label. These quantities are saved as the attributes `theta_` (the means) and `sigma_` (the variances), then used in `predict()`. Parameters like these that **are dependent on data** are only defined in `fit()`, not the constructor, and they are always named with a trailing underscore. These "non-hyper" parameters are often simply called *model parameters*.

```
>>> gnb.classes_        # The collection of distinct training labels.
array([0, 1])
>>> gnb.theta_[:,0]     # The means of the first feature, grouped by label.
array([17.55785276, 12.0354981 ])
# The samples with label 0 have a mean of 17.56 in the first feature.
```

The `fit()` method should do all of the "heavy lifting" by calculating the model parameters. The `predict()` method should then use these parameters to choose a label for test data.

	Hyperparameters	Model Parameters
Data dependence	No	Yes
Initialization location	`__init__()`	`fit()`
Naming convention	Same as argument name	Ends with an underscore
Examples	`n_neighbors`, `algorithm`, `n_jobs`	`classes_`, `theta_`, `sigma_`

Table D.1: Naming and initialization conventions for scikit-learn model parameters.

Building Custom Estimators

The consistent conventions in the various scikit-learn classes makes it easy to use a wide variety of estimators with near-identical syntax. These conventions also makes it possible to write custom estimators that behave like native scikit-learn objects. This usually only involves writing `fit()` and `predict()` methods and inheriting from the appropriate base classes. As a simple (though poorly performing) example, consider an estimator that either always predicts the same user-provided label, or that always predicts the most common label in the training data. Which strategy to use is independent of the data, so we encode that behavior with hyperparameters; the most common label must be calculated from the data, so that is a model parameter.

```
>>> import numpy as np
>>> from collections import Counter
>>> from sklearn.base import BaseEstimator, ClassifierMixin

>>> class PopularClassifier(BaseEstimator, ClassifierMixin):
...     """Classifier that always guesses the most common training label."""
...     def __init__(self, strategy="most_frequent", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         """Find and store the most common label."""
...         self.popular_label_ = Counter(y).most_common(1)[0][0]
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always guess the most popular training label."""
...         M = X.shape[0]
...         if self.strategy == "most_frequent":
...             return np.full(M, self.popular_label_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train a PopularClassifier on the breast cancer training data.
>>> pc = PopularClassifier().fit(X_train, y_train)
>>> pc.popular_label_
1
# Score the model on the testing data.
>>> pc.score(X_test, y_test)
0.6573426573426573                  # 65.73% of the testing data is labeled 1.

# Change the strategy to always guess 0 by changing the hyperparameters.
>>> pc.strategy = "constant"
>>> pc.constant = 0
>>> pc.score(X_test, y_test)
0.34265734265734266                  # 34.27% of the testing data is labeled 0.
```

This is a terrible classifier, but it is actually implemented as `sklearn.dummy.DummyClassifier` because any legitimate machine learning algorithm should be able to beat it, so it is useful as a baseline comparison.

Note that `score()` was inherited from `ClassifierMixin` (it isn't defined explicitly), so it returns a classification rate. In the next example, a slight simplification of the equally unintelligent `sklearn.dummy.DummyRegressor`, the `score()` method is inherited from `RegressorMixin`, so it returns an R^2 value.

```
>>> from sklearn.base import RegressorMixin

>>> class ConstRegressor(BaseEstimator, RegressorMixin):
...     """Regressor that always predicts a mean or median of training data."""
...     def __init__(self, strategy="mean", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         self.mean_, self.median_ = np.mean(y), np.median(y)
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always predict the middle of the training data."""
...         M = X.shape[0]
...         if self.strategy == "mean":
...             return np.full(M, self.mean_)
...         elif self.strategy == "median":
...             return np.full(M, self.median_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train on the breast cancer data (treating it as a regression problem).
>>> cr = ConstRegressor(strategy="mean").fit(X_train, y_train)
>>> print("mean:", cr.mean_, " median:", cr.median_)
mean: 0.6173708920187794  median: 1.0

# Get the R^2 score of the regression on the testing data.
>>> cr.score(X_train, y_train)
0                           # Unsurprisingly, no correlation.
```

Achtung!

Both `PopularClassifier` and `ConstRegressor` wait until `predict()` to validate the `strategy` hyperparameter. The check could easily be done in the constructor, but that goes against scikit-learn conventions: in order to cooperate with automated validation tools, the constructor of any class inheriting from `BaseEstimator` must store the arguments of `__init__()` as attributes—with the same names as the arguments—and do nothing else.

Note

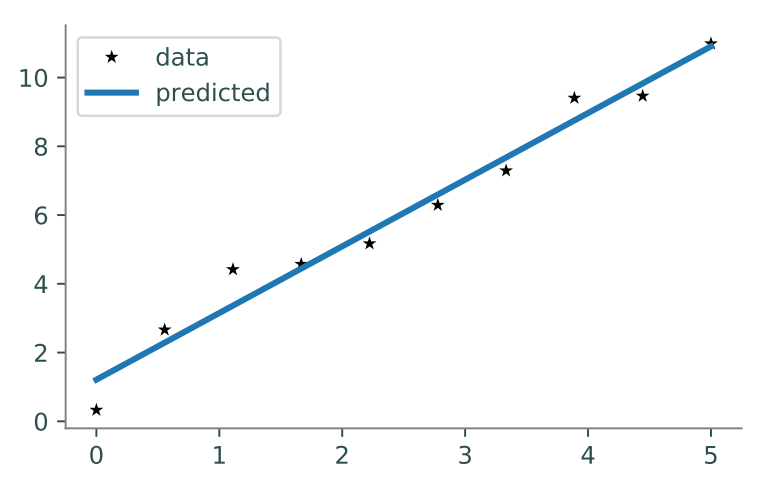
The first input to `fit()` and `predict()` are **always** two-dimensional $N \times D$ NumPy arrays, where N is the number of observations and D is the number of features. To fit or predict on one-dimensional data ($D = 1$), reshape the input array into a "column vector" before feeding it into the estimator. One-dimensional problems are somewhat rare in machine learning, but the following example shows how to do a simple one-dimensional linear regression.

```
>>> from matplotlib import pyplot as plt
>>> from sklearn.linear_model import LinearRegression

# Generate data for a 1-dimensional regression problem.
>>> X = np.linspace(0, 5, 10)
>>> Y = 2*X + 1 + np.random.normal(size=10)

# Reshape the training data into a column vector.
>>> lr = LinearRegression().fit(X.reshape((-1,1)), Y)

# Define another set of points to do predictions on.
>>> x = np.linspace(0, 5, 20)
>>> y = lr.predict(x.reshape((-1,1)))   # Reshape before predicting.
>>> plt.plot(X, Y, 'k*', label="data")
>>> plt.plot(x, y, label="predicted")
>>> plt.legend(loc="upper left")
>>> plt.show()
```



Transformers

A scikit-learn *transformer* processes data to make it better suited for estimation. This may involve shifting and scaling data, dropping columns, replacing missing values, and so on.

Classes that inherit from the `TransformerMixin` base class have a `fit()` method that accepts an $N \times D$ matrix X (like an estimator) and an optional set of labels. The labels are not needed—in fact the `fit()` method should do nothing with them—but the parameter for the labels remains as a keyword argument to be consistent with the `fit(X,y)` syntax of estimators. Instead of a `predict()` method, the `transform()` method accepts data, modifies it (usually via a copy), and returns the result. The new data may or may not have the same number of columns as the original data.

One common transformation is shifting and scaling the features (columns) so that they each have a mean of 0 and a standard deviation of 1. The following example implements a basic version of this transformer.

```
>>> from sklearn.base import TransformerMixin

>>> class NormalizingTransformer(BaseEstimator, TransformerMixin):
...     def fit(self, X, y=None):
...         """Calculate the mean and standard deviation of each column."""
...         self.mu_ = np.mean(X, axis=0)
...         self.sig_ = np.std(X, axis=0)
...         return self
...
...     def transform(self, X):
...         """Center each column at zero and normalize it."""
...         return (X - self.mu_) / self.sig_
...
# Fit the transformer and transform the cancer data (both train and test).
>>> nt = NormalizingTransformer()
>>> Z_train = nt.fit_transform(X_train) # Or nt.fit(X_train).transform(X_train)
>>> Z_test = nt.transform(X_test)       # Transform test data (without fitting)

>>> np.mean(Z_train, axis=0)[:3]        # The columns of Z_train have mean 0...
array([-8.08951237e-16, -1.72006384e-17,  1.78678147e-15])
>>> np.std(Z_train, axis=0)[:3]         # ...and have unit variance.
array([1., 1., 1.])
>>> np.mean(Z_test, axis=0)[:3]         # The columns of Z_test each have mean
array([-0.02355067,  0.11665332, -0.03996177])                  # close to 0...
>>> np.std(Z_test, axis=0)[:3]          # ...and have close to unit deviation.
array([0.9263711 , 1.18461151, 0.91548103])

# Check to see if the classification improved.
>>> knn.fit(X_train, y_train).score(X_test, y_test)             # Old score.
0.8951048951048951
>>> knn.fit(Z_train, y_train).score(Z_test, y_test)             # New score.
0.958041958041958
```

This particular transformer is implemented as `sklearn.preprocessing.StandardScaler`. A close cousin is `sklearn.preprocessing.RobustScaler`, which ignores outliers when choosing the scaling and shifting factors.

Like estimators, transformers may have both hyperparameters (provided to the constructor) and model parameters (determined by `fit()`). Thus a transformer looks and acts like an estimator, with the exception of the `predict()` and `transform()` methods.

Achtung!

The `transform()` method should only rely on model parameters derived from the training data in `fit()`, **not** on the data that is worked on in `transform()`. For example, if the `NormalizingTransformer` is fit with the input $\widehat{X}$, then `transform()` should shift and scale any input X by the mean and standard deviation of $\widehat{X}$, not by the mean and standard deviation of X. Otherwise, the transformation is different for each input X.

Scikit-learn Module	**Classifier** Name	Notable Hyperparameters
discriminant_analysis	LinearDiscriminantAnalysis	solver, shrinkage, n_components
discriminant_analysis	QuadraticDiscriminantAnalysis	reg_param
ensemble	AdaBoostClassifier	n_estimators, learning_rate
ensemble	RandomForestClassifier	n_estimators, max_depth
linear_model	LogisticRegression	penalty, C
linear_model	SGDClassifier	loss, penalty, alpha
naive_bayes	GaussianNB	priors
naive_bayes	MultinomialNB	alpha
neighbors	KNeighborsClassifier	n_neighbors, weights
neighbors	RadiusNeighborsClassifier	radius, weights
neural_network	MLPClassifier	hidden_layer_size, activation
svm	SVC	C, kernel
tree	DecisionTreeClassifier	max_depth
Scikit-learn Module	**Regressor** Name	Notable Hyperparameters
ensemble	AdaBoostRegressor	n_estimators, learning_rate
ensemble	ExtraTreesRegressor	n_estimators, max_depth
ensemble	GradientBoostingRegressor	n_estimators, max_depth
ensemble	RandomForestRegressor	n_estimators, max_depth
isotonic	IsotonicRegression	y_min, y_max
kernel_ridge	KernelRidge	alpha, kernel
linear_model	LinearRegression	fit_intercept
neural_network	MLPRegressor	hidden_layer_size, activation
svm	SVR	C, kernel
tree	DecisionTreeRegressor	max_depth
Module	**Transformer** Name	Notable Hyperparameters
decomposition	PCA	n_components
preprocessing	Imputer	missing_values, strategy
preprocessing	MinMaxScaler	feature_range
preprocessing	OneHotEncoder	categorical_features
preprocessing	QuantileTransformer	n_quantiles, output_distribution
preprocessing	RobustScaler	with_centering, with_scaling
preprocessing	StandardScaler	with_mean, with_std

Table D.2: Common scikit-learn classifiers, regressors, and transformers. For full documentation on these classes, see `http://scikit-learn.org/stable/modules/classes.html`.

Validation Tools

Knowing how to determine whether or not an estimator 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 range of the desired model is $Y = \{0, 1\}$.

Evaluation Metrics

The `score()` method of a scikit-learn estimator representing the model $f : X \to \{0, 1\}$ returns the *accuracy* of the model, which is the percent of labels that are 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 observations with actual label i but that are classified as label j. In binary classification, calling the class with label 0 the *negatives* and the class with label 1 the *positives*, this becomes the following.

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

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, 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 (for example, always predicting the same label), so the following metric is also common.

- F_β *Score*: $(1 + \beta^2)\dfrac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \dfrac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2 FN}$.

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 these metrics in `sklearn.metrics`, as well as functions for evaluating regression, non-binary classification, and clustering models. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. For the complete list and further discussion, see `http://scikit-learn.org/stable/modules/model_evaluation.html`.

```
>>> from sklearn.metrics import (confusion_matrix, classification_report,
...                              accuracy_score, precision_score,
...                              recall_score, f1_score)

# Fit the esimator to training data and predict the test labels.
>>> 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 accuracy (the "usual" score), precision, recall, and f1 scores.
>>> accuracy_score(y_test, knn_predicted)   # (CM[0,0] + CM[1,1]) / CM.sum()
0.8951048951048951
>>> precision_score(y_test, knn_predicted)  # CM[1,1] / CM[:,1].sum()
0.9438202247191011
>>> recall_score(y_test, knn_predicted)     # CM[1,1] / CM[1,:].sum()
0.8936170212765957
>>> f1_score(y_test, knn_predicted)
0.9180327868852459

# Get all of these scores at once with classification_report().
>>> 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

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

Cross Validation

The `sklearn.model_selection` module has utilities to streamline and improve model evaluation.

- `train_test_split()` randomly splits data into training and testing sets (we already used this).
- `cross_val_score()` randomly splits the data and trains and scores the model a set number of times. Each trial uses different training data and results in a different model. The function returns the score of each trial.
- `cross_validate()` does the same thing as `cross_val_score()`, but it also reports the time it took to fit, the time it took to score, and the scores for the test set as well as the training set.

Doing multiple evaluations with different testing and training sets is extremely important. If the scores on a cross validation test vary wildly, the model is likely overfitting to the training data.

```
>>> from sklearn.model_selection import cross_val_score, cross_validate

# Make (but do not train) a classifier to test.
>>> knn = KNeighborsClassifier(n_neighbors=3)

# Test the classifier on the training data 4 times.
>>> cross_val_score(knn, X_train, y_train, cv=4)
array([0.88811189, 0.92957746, 0.96478873, 0.92253521])

# Get more details on the train/test procedure.
>>> cross_validate(knn, X_train, y_train, cv=4,
...                return_train_score=False)
{'fit_time': array([0.00064683, 0.00042295, 0.00040913, 0.00040436]),
 'score_time': array([0.00115728, 0.00109601, 0.00105286, 0.00102782]),
 'test_score': array([0.88811189, 0.92957746, 0.96478873, 0.92253521])}

# Do the scoring with an alternative metric.
>>> cross_val_score(knn, X_train, y_train, scoring="f1", cv=4)
array([0.93048128, 0.95652174, 0.96629213, 0.93103448])
```

> **Note**
>
> Any estimator, even a user-defined class, can be evaluated with the scikit-learn tools presented in this section as long as that class conforms to the scikit-learn API discussed previously (i.e., inheriting from the correct base classes, having `fit()` and `predict()` methods, managing hyperparameters and parameters correctly, and so on). Any time you define a custom estimator, following the scikit-learn API gives you instant access to tools such as `cross_val_score()`.

Grid Search

Recall that the *hyperparameters* of a machine learning model are user-provided parameters that do not depend on the training data. Finding the optimal hyperparameters for a given model is a challenging and active area of research.[2] However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with an estimator, a dictionary of hyperparameters, and cross validation parameters (such as `cv` and `scoring`). When its `fit()` method is called, it does a cross validation test on the given estimator with every possible hyperparameter combination.

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

[2] Intelligent hyperparameter selection is sometimes called *metalearning*. See, for example, [SGCP+18].

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

>>> knn = KNeighborsClassifier()
# Specify the hyperparameters to vary and the possible values they should take.
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...               "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, cv=4, scoring="f1", verbose=1)
>>> knn_gs.fit(X_train, y_train)
Fitting 4 folds for each of 5 candidates, totalling 20 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent worker.
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.1s finished

# After fitting, the gridsearch object has data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_)
{'n_neighbors': 5, 'weights': 'uniform'} 0.9532526583188765
```

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*. To parallelize the grid search over n cores, set the `n_jobs` parameter to n, or set it to -1 to divide the labor between as many cores as are available.

In some circumstances, the parameter grid can be also organized in a way that eliminates redundancy. Consider an `SVC` classifier from `sklearn.svm`, an estimator that works by lifting the data into a high-dimensional space, then constructing a hyperplane to separate the classes. The `SVC` has a hyperparameter, `kernel`, that determines how the lifting into higher dimensions is done, and for each choice of kernel there are additional corresponding hyperparameters. To search the total hyperparameter space without redundancies, enter the parameter grid as a list of dictionaries, each of which defines a different section of the hyperparameter space. In the following code, doing so reduces the number of trials from $3 \times 2 \times 3 \times 4 = 72$ to only $1 + (1 \times 1 \times 3) + (1 \times 4) = 11$.

```
>>> from sklearn.svm import SVC

>>> svc = SVC(C=0.01, max_iter=100)
>>> param_grid = [
...     {"kernel": ["linear"]},
...     {"kernel": ["poly"], "degree": [2,3], "coef0": [0,1,5]},
...     {"kernel": ["rbf"], "gamma": [.01, .1, 1, 100]}]
>>> svc_gs = GridSearchCV(svc, param_grid,
                          cv=4, scoring="f1",
                          verbose=1, n_jobs=-1).fit(X_train, y_train)
Fitting 4 folds for each of 11 candidates, totalling 44 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  44 out of  44 | elapsed:    2.4s finished

>>> print(svc_gs.best_params_, svc_gs.best_score_)
{'gamma': 0.01, 'kernel': 'rbf'} 0.8909310239174055
```

See `https://scikit-learn.org/stable/modules/grid_search.html` for more details about `GridSearchCV` and its relatives.

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* (`sklearn.pipeline.Pipeline`) chains together one or more transformers and one estimator into a single object, complete with `fit()` and `predict()` methods. For example, it is often a good idea to shift and scale data before feeding it into a classifier. The `StandardScaler` transformer can be combined with a classifier with a pipeline. Calling `fit()` on the resulting object calls `fit_transform()` on each successive transformer, then `fit()` on the estimator at the end. Likewise, calling `predict()` on the `Pipeline` object calls `transform()` on each transformer, then `predict()` on the estimator.

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

# Chain together a scaler transformer and a KNN estimator.
>>> pipe = Pipeline([("scaler", StandardScaler()),      # "scaler" is a label.
                     ("knn", KNeighborsClassifier())])  # "knn" is a label.
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972                       # Already an improvement!
```

Since `Pipeline` objects behaves like estimators (following the `fit()` and `predict()` conventions), they can be used with tools like `cross_val_score()` and `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 part of the pipeline that is labeled `knn`.

```
# Specify the possible hyperparameters 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 = Pipeline([("scaler", StandardScaler()),
                     ("knn", KNeighborsClassifier())])
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=4, n_jobs=-1, verbose=1).fit(X_train, y_train)
Fitting 4 folds for each of 40 candidates, totalling 160 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 160 out of 160 | elapsed:    0.3s finished

>>> 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 could end in either a `KNeighborsClassier()` or an `SVC()`, even though they have different hyperparameters. Like before, use a list of dictionaries to specify the hyperparameter space.

```
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("classifier", KNeighborsClassifier())])
>>> 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]}]
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=5, scoring="f1",
...                        verbose = 1, n_jobs=-1).fit(X_train, y_train)
Fitting 5 folds for each of 44 candidates, totalling 220 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 220 out of 220 | elapsed:    0.6s finished

>>> 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]])
```

Additional Material

Exercises

Problem 1. Writing custom scikit-learn transformers is a convenient way to organize the data cleaning process. Consider the data in `titanic.csv`, which contains information about passengers on the maiden voyage of the *RMS Titanic* in 1912. Write a custom transformer class to clean this data, implementing the `transform()` method as follows:

1. Extract a copy of data frame with just the `"Pclass"`, `"Sex"`, and `"Age"` columns.
2. Replace `NaN` values in the `"Age"` column (of the copied data frame) with the mean age. The mean age of the training data should be calculated in `fit()` and used in `transform()` (compare this step to using `sklearn.preprocessing.Imputer`).
3. Convert the `"Pclass"` column datatype to pandas categoricals (`pd.CategoricalIndex`).
4. Use `pd.get_dummies()` to convert the categorical columns to multiple binary columns (compare this step to using `sklearn.preprocessing.OneHotEncoder`).
5. Cast the result as a NumPy array and return it.

Ensure that your transformer matches scikit-learn conventions (it inherits from the correct base classes, `fit()` returns `self`, etc.).

Problem 2. Read the data from `titanic.csv` with `pd.read_csv()`. The `"Survived"` column indicates which passengers survived, so the entries of the column are the labels that we would like to predict. Drop any rows in the raw data that have `NaN` values in the `"Survived"` column, then separate the column from the rest of the data. Split the data and labels into training and testing sets. Use the training data to fit a transformer from Problem 1, then use that transformer to clean the training set, then the testing set. Finally, train a `LogisticRegressionClassifier` and a `RandomForestClassifier` on the cleaned training data, and score them using the cleaned test set.

Problem 3. Use `classification_report()` to score your classifiers from Problem 2. Next, do a grid search for each classifier (using only the cleaned training data), varying at least two hyperparameters for each kind of model. Use `classification_report()` to score the resulting best estimators with the cleaned test data. Try changing the hyperparameter spaces or scoring metrics so that each grid search yields a better estimator.

Problem 4. Make a pipeline with at least two transformers to further process the Titanic dataset. Do a gridsearch on the pipeline and report the hyperparameters of the best estimator.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[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.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[SGCP+18] Brandon Schoenfeld, Christophe Giraud-Carrier, Mason Poggemann, Jarom Christensen, and Kevin Seppi. Preprocessor selection for machine learning pipelines. *arXiv preprint arXiv:1810.09942*, 2018.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.