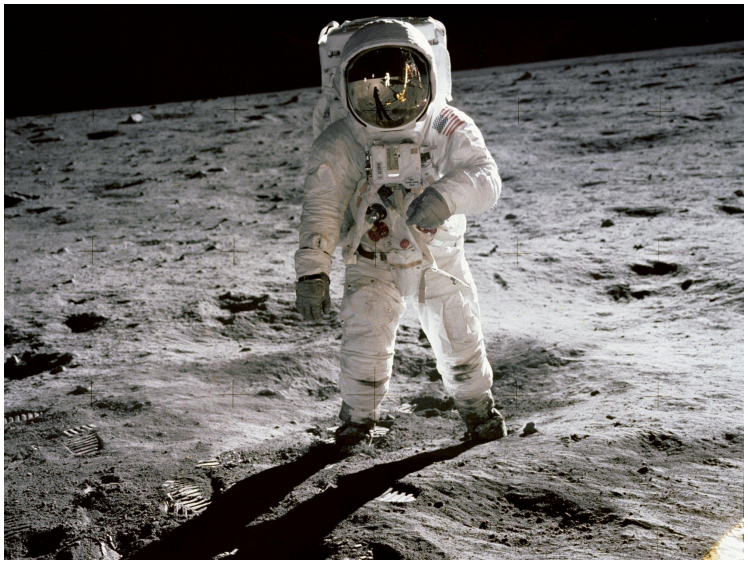
Labs for Foundations of Applied Mathematics

Volume 3
Modeling with Uncertainty and Data

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Fuhriman
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

K. Jacobson
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
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

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

M. Stauffer
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

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

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

Contents

Part I

Labs

1 Kalman Filter

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

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

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

Linear Dynamical Systems

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

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + B_k\mathbf{u}_k + \boldsymbol{\varepsilon}_k \tag{1.1}$$

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

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

$$\mathbf{z}_k = H_k\mathbf{x}_k + \boldsymbol{\delta}_k \tag{1.2}$$

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

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

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

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

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

        Parameters
        ----------
        F : ndarray of shape (n,n)
            The state transition model.
        Q : ndarray of shape (n,n)
            The covariance matrix for the state noise.
        H : ndarray of shape (m,n)
            The observation model.
        R : ndarray of shape (m,m)
            The covariance 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

$$z_x = s_x + \delta_x$$
$$z_y = s_y + \delta_y,$$

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

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

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

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 1.1 and 1.2.

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):
    """
```

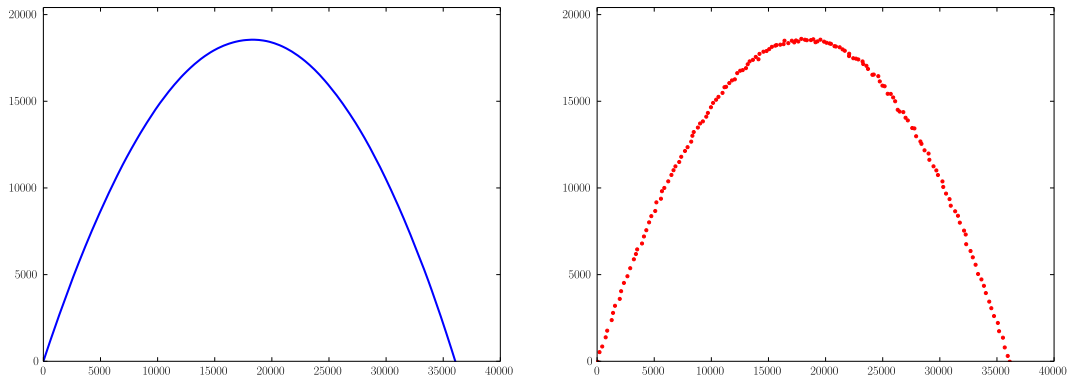


Figure 1.1: State sequence (left) and sampling of observation sequence (right).

```
    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 1.1.

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).
```

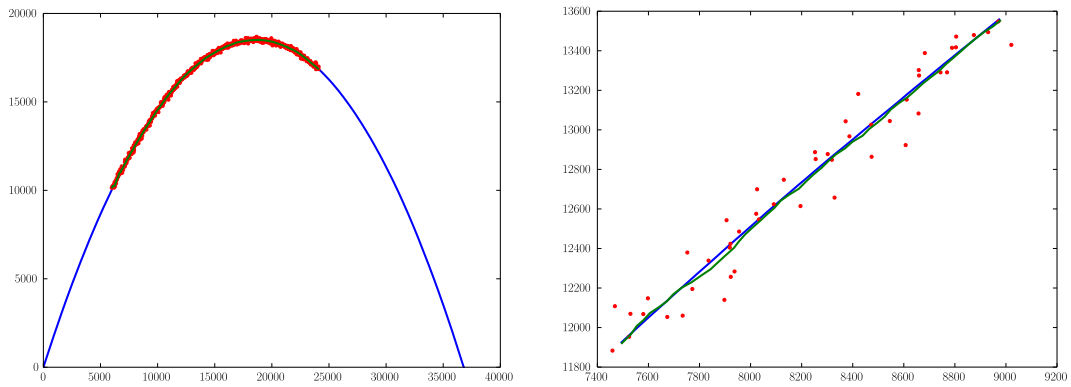


Figure 1.2: State estimates together with observations and true state sequence (detailed view on the right).

```
    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 1.2.

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

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

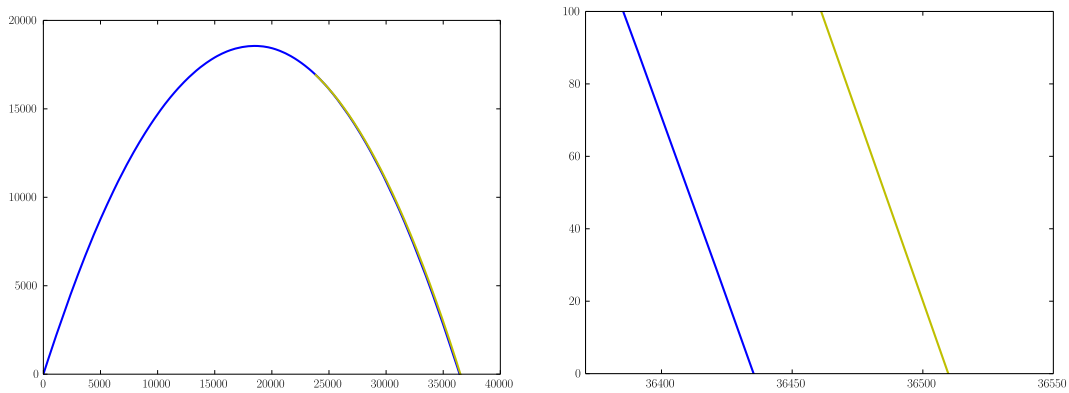


Figure 1.3: Predicted vs. actual point of impact (detailed view on right).

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 1.3.

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.

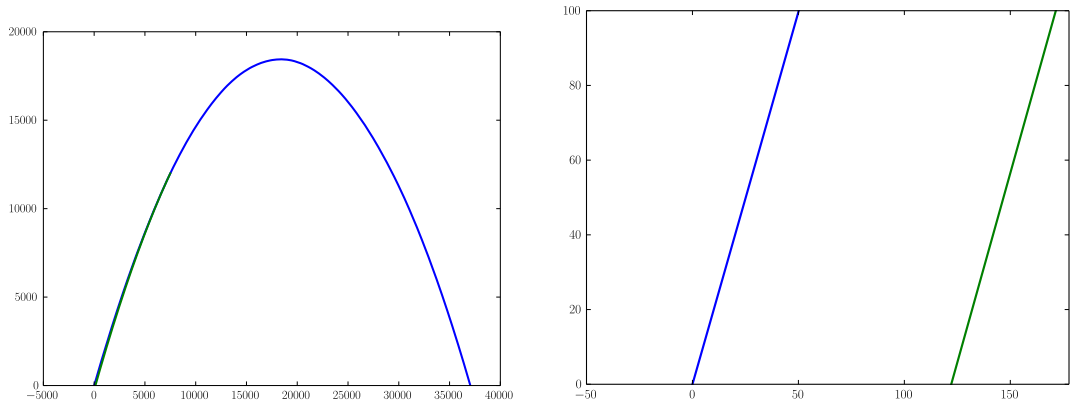


Figure 1.4: Predicted vs. actual point of origin (detailed view on right).

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. (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 1.4.

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

2 ARMA Models

Lab Objective: *Fit and forecast ARMA models.*

An ARMA(p, q) model is a covariance-stationary discrete stochastic process $\{z_t\}$ that satisfies

$$z_t - \mu = \left(\sum_{i=1}^{p} \phi_i (z_{t-i} - \mu)\right) + a_t + \left(\sum_{j=1}^{q} \theta_j a_{t-j}\right) \tag{2.1}$$

where $\mu = E[z_t]$ and a_t are identically-distributed Gaussian variables with variance σ_a^2. We note that the assumption that $\{z_t\}$ is covariance-stationary is eqivalent to the condition that the roots of the polynomial in B

$$\phi(B) = 1 - \sum_{i=1}^{p} \phi_i B^i \tag{2.2}$$

lie outside of the unit circle.

The first sum on the right hand side of 2.1 is interpreted as an "autoregression" since it is a linear combination of previously observed values of z_t. The second sum is interpreted as a "moving average" of the current and previous error terms; though formally similar to an average, note that the θ_j need not be positive nor sum to one. We say that an ARMA(p, q) model is an "autoregressive moving-average model of order p, q".

Likelihood via Kalman Filter

In a general ARMA(p, q) model, the likelihood is a function of the unobserved error terms a_t and is not trivial to compute. Simple approximations can be made, but these may be inaccurate under certain circumstances. Explicit derivations of the likelihood are possible, but tedious. However, when the ARMA model is placed in state-space, the Kalman filter affords a straightforward, recursive way to compute the likelihood.

We demonstrate a state-space representation of an ARMA(p, q) model. If $r = \max(p, q + 1)$, we write

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{2.3}$$

$$H = \begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_{r-1} \end{bmatrix} \tag{2.4}$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{2.5}$$

$$w_t \sim \text{MVN}(0, Q), \tag{2.6}$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Then the linear stochastic dynamical system

$$x_{t+1} = Fx_t + w_t \tag{2.7}$$

$$z_t = Hx_t + \mu \tag{2.8}$$

describes the same process as the original ARMA model. Note that the equation for z_t involves a deterministic component, namely μ. The Kalman filter theory developed in the previous lab, however, assumed no deterministic component for the observations z_t, so you should subtract off the mean μ from the time series observations z_t when using them in the predict and update steps.

Let $\Theta = \{\phi_i, \theta_j, \mu, \sigma_a^2\}$ be the set of parameters for an ARMA(p, q) model. Suppose we have a set of observations $z_1, z_2, \ldots, z_n$, denoted collectively by $\{z_t\}$. Using the chain rule, we can factorize the likelihood of the model under these data as

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} p(z_t|z_{t-1}, \ldots, z_1, \Theta) \tag{2.9}$$

Since we have assumed that the error terms are Gaussian, each conditional distribution in 2.9 is also Gaussian, and is completely characterized by its mean and variance. But these two quantities are easily found via the Kalman filter, namely

$$\text{mean} \quad H\hat{x}_{t|t-1} + \mu \tag{2.10}$$

$$\text{variance} \quad HP_{t|t-1}H^T \tag{2.11}$$

where $\hat{x}_{t|t-1}$ and $P_{t|t-1}$ are found during the Predict step. The likelihood becomes

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} N(z_t;\ H\hat{x}_{t|t-1} + \mu,\ HP_{t|t-1}H^T) \tag{2.12}$$

We begin the recursion by letting

$$\hat{x}_{1|0} = \mathbb{E}(x_1) = 0 \tag{2.13}$$

$$\text{vec}(P_{1|0}) = \mathbb{E}\left[(x_1 - \mathbb{E}x_1)(x_1 - \mathbb{E}x_1)^T\right] = \left[I_{r^2} - (F \otimes F)\right]^{-1} \cdot \text{vec}(Q) \tag{2.14}$$

where vec flattens a matrix and $\otimes$ is the Kronecker product (`numpy.kron`).

Problem 1. Write a function that computes the log-likelihood of an ARMA(p, q) model, given a time series z_t.

```
def arma_likelihood(time_series, phis=array([]), thetas=array([]), mu=0.,
        sigma=1.):
    """
    Return the log-likelihood of the ARMA model parameters, given the time
    series.

    Parameters
    ----------
    time_series : ndarray of shape (n,1)
        The time series in question, z_t
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables

    Returns
    -------
    log_likelihood : float
        The log-likelihood of the model
    """
    pass
```

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

```
>>> arma_likelihood(time_series_a, phis=array([0.9]), mu=17., sigma=0.4)
-77.6035
```

Identification and Fitting

When modeling a data set with an ARMA(p, q) model, the order of the model must be determined, as well as the other parameters. The process of choosing p and q is called *model identification*. Different methods have been used; for example, Box and Jenkins propose a methodology that involves examining the estimated autocorrelation and partial-autocorrelation functions of the data. We will choose p and q that minimize the Akaike information criterion with a correction (AICc), given by

$$2k\left(1+\frac{k+1}{n-k}\right)-2\ell(\Theta) \tag{2.15}$$

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

To compute the maximum likelihood for a model class, we need to optimize 2.12 over the space of parameters Θ. We can do so by using the function from Problem 1 along with some optimization routine, such as `scipy.optimize.fmin`.

Problem 2. Write a function that accepts a time series $\{z_t\}$ and returns the parameters of the model that minimize the AICc, given the constraint that $p \leq 3$, $q \leq 3$.

```
def arma_fit(time_series):
    """
    Return the ARMA model that minimizes AICc for the given time series,
    subject to p,q <= 3.

    Parameters
    ----------
    time_series : ndarray of shape (n,1)
        The time series in question, z_t

    Returns
    -------
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables
    """
    pass
```

Here's a hint for performing the optimization at each step, using `scipy.optimize.fmin`.

```
>>> # assume p, q, and time_series are defined
>>> def f(x): # x contains the phis, thetas, mu, and sigma
>>>     return -1*arma_likelihood(time_series, phis=x[:p], thetas=x[p:p+q↩
    ], mu=x[-2],sigma=x[-1])
>>> # create initial point
>>> x0 = np.zeros(p+q+2)
>>> x0[-2] = time_series.mean()
>>> x0[-1] = time_series.std()
>>> sol = op.fmin(f,x0,maxiter=10000, maxfun=10000)
```

The variable `sol` is a flat array of length $p+q+2$, whose first p entries give the optimal values for the ϕ polynomial, the next q entries give the optimal values for the θ polynomial, and the last two entries give the optimal values for μ and σ_a, respectively. Notice that we defined a wrapper function f to feed into the `scipy.optimize.fmin` routine. This wrapper function returns the *negative* of the log likelihood, since the optimization routine we are calling finds the minimum of a function, and we are interested in the *maximum* of the log likelihood.

Your code should produce the following output, where the input data is found in `time_series_a.txt` (it may take a minute or so to run):

```
>>> arma_fit(time_series_a)
(array([ 0.9087]), array([-0.5759]), 17.0652..., 0.3125...)
```

Problem 3. Use your solution from Problem 2 to fit models to the data found in `time_series_a.txt`, `time_series_b.txt`, `time_series_c.txt`. Report the fitted parameters p, q, Θ.

Forecasting

The Kalman filter provides a straightforward way to predict future states, by giving the mean and variance of the conditional distribution of future observations.

$$z_{t+k}|z_1, \cdots, z_t \sim N(z_{t+k};\ H\hat{x}_{t+k|t} + \mu,\ HP_{t+k|t}H^T) \tag{2.16}$$

Recall the relations

$$\hat{x}_{t+k|t} = F\hat{x}_{t+k-1|t} \tag{2.17}$$

$$P_{t+k|t} = FP_{t+k-1|t}F^T + Q \tag{2.18}$$

Problem 4. Forecast each data set ahead 20 intervals using the parameters discovered from Problem 3, and plot their expected values along with the original data set. Also plot the expected values plus and minus σ_{t+k}, and plus and minus $2\sigma_{t+k}$ to demonstrate credible intervals.

Note that we need the values of $\hat{x}_{n|n}$ and $P_{n|n}$ to get started. As usual, these estimates can be found using the Predict and Update recursions. Initialize $\hat{x}_{1|0}$ and $P_{1|0}$ as before, run the recursions until you obtain $\hat{x}_{n|n}$ and $P_{n|n}$, and then calculate the future estimates $\hat{x}_{t+k|t}$ and $P_{t+k|t}$. Use these to calculate the expected value and standard deviation for forecasted values (given by $H\hat{x}_{t+k|t} + \mu$ and $\sqrt{HP_{t+k|t}H^T}$, respectively).

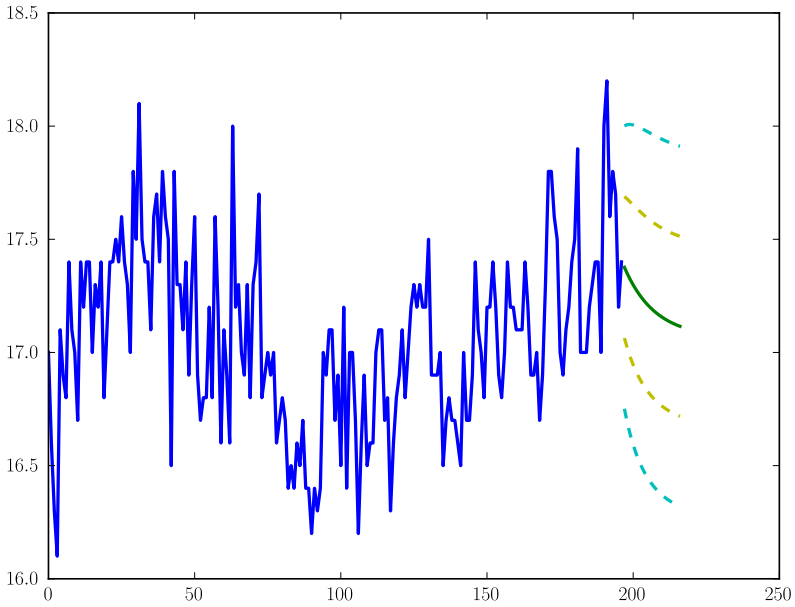
```
def arma_forecast(time_series, phis=array([]), thetas=array([]), mu=0.,
        sigma=1., future_periods=20):
    """
    Return forecasts for a time series modeled with the given ARMA model.

    Parameters
    ----------
```

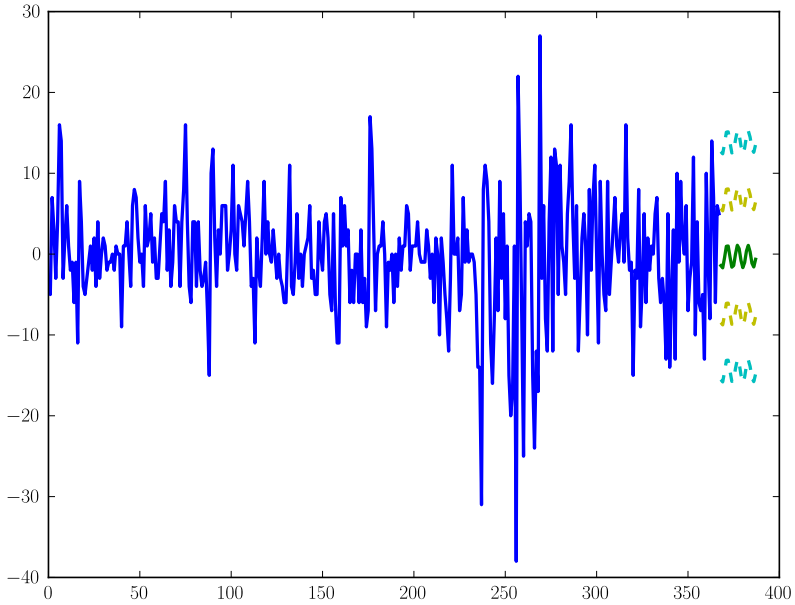
```
    time_series : ndarray of shape (n,1)
        The time series in question, z_t
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables
    future_periods : int
        The number of future periods to return

    Returns
    -------
    e_vals : ndarray of shape (future_periods,)
        The expected values of z for times n+1, ..., n+future_periods
    sigs : ndarray of shape (future_periods,)
        The standard deviations of z for times n+1, ..., n+future_periods
    """
    pass
```

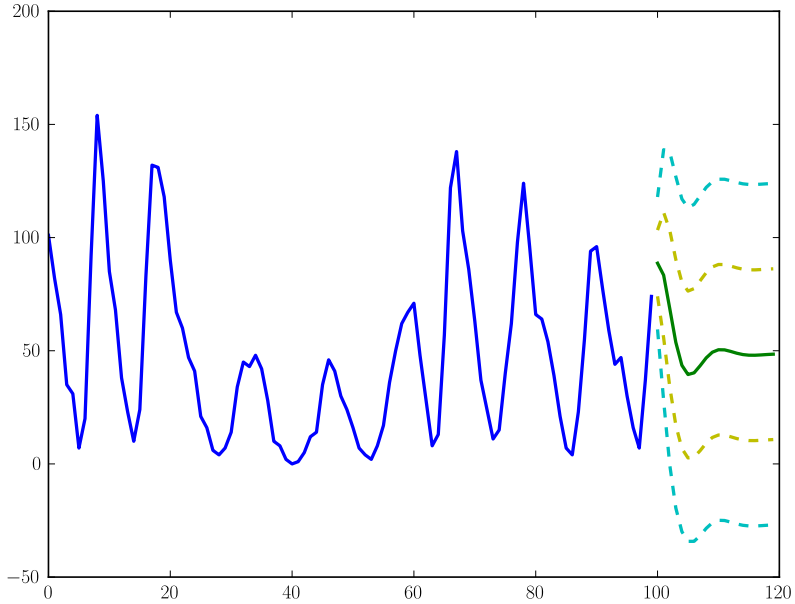
You should get the following result:

```
>>> arma_forecast(time_series_a, phis, thetas, mu, sigma, 4)
(array([ 17.3762,  17.3478,  17.322 ,  17.2986]),
 array([ 0.3125,  0.3294,  0.3427,  0.3533]))
```

Your results (when using twenty future periods) should match those in Figure 2.1.


18.5
18.0
17.5
17.0
16.5
16.0
0
50
100
150
200
250


30
20
10
0
−10
−20
−30
−40
0
50
100
150
200
250
300
350
400


200
150
100
50
0
−50
0
20
40
60
80
100
120

3 Discrete Hidden Markov Models

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

Given a discrete state-space Hidden Markov Model (HMM) with parameters λ and an observation sequence O, we would like to answer three questions:

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

2. What is the most likely state sequence to have generated O, given λ?

3. How can we choose the parameters λ that maximize $\mathbb{P}(O|\,\lambda)$?

The answers to these questions are centered around the *forward-backward* algorithm for HMMs. For the second question, the approach taken in this lab will be to find the state sequence maximizing the expected number of correct states. The third question is an example of *unsupervised learning*, since we are attempting to learn (or fit) model parameters using data (the observation sequence O) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

We assume throughout this lab that the HMM has a discrete state space of cardinality N and a discrete observation space of cardinality M. In this context $\lambda = (A, B, \pi)$, where A is a $N \times N$ column-stochastic matrix (the state transition model), B is a $M \times N$ column-stochastic matrix (the state observation model), and π is a stochastic vector of length N (the initial state distribution). Further, O is a vector of length T with values in the set $\{1, 2, \dots, M\}$.

> **Achtung!**
>
> The mathematical exposition in the lab assumes the standard 1-based indexing of vectors and matrices. Be sure to carefully translate the various formulae into 0-based indexing when implementing these methods for Python coding. This means that, in Python, your array containing the observation sequence O will actually have values in the set $\{0, 1, \dots, M-1\}$ so that they may be used to index the matrix B correctly.

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

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

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

The Forward Pass

Our first task is to efficiently compute $\log \mathbb{P}(O|\lambda)$. We can do this using the *forward pass* of the forward-backward algorithm. We must take care to compute all values in a numerically stable way; we do this by properly scaling values as necessary.

We compute a scaled forward probability matrix $\widehat{\alpha}$ of dimension $T \times N$ as follows: Let $\widehat{\alpha}_{i,:}, B_{i,:}$ denote the i-th rows of $\widehat{\alpha}$ and B, respectively, let $\odot$ denote the Hadamard (or entry-wise) product of arrays, and let $\langle \cdot, \cdot \rangle$ denote the standard dot product. (Note that here, using 0-based indexing and the toy HMM example, $B_{O_3,:}$ would refer to $[.5, .1]$.) Then

- $c_1 = \langle \pi, B_{O_1,:} \rangle^{-1}$
- $\widehat{\alpha}_{1,:} = c_1(\pi \odot B_{O_1,:})$
- For $t = 2, \ldots, T$:

  $c_t = \langle A\widehat{\alpha}_{t-1,:}, B_{O_t,:} \rangle^{-1}$
  $\widehat{\alpha}_{t,:} = c_t((A\widehat{\alpha}_{t-1,:}) \odot B_{O_t,:})$

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:

$$\log \mathbb{P}(O|\lambda) = -\sum_{t=1}^{T} \log c_t.$$

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

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

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

```
    Returns
    -------
    alpha : ndarray of shape (T,N)
        The scaled forward probability matrix
    c : ndarray of shape (T,)
        The scaling factors c = [c_1,c_2,...,c_T]
    """
    pass
```

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

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

The Backward Pass

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

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

- $\widehat{\beta}_{T,i} = c_T$ for $i = 1, \ldots, N$
- $\widehat{\beta}_{t,:} = c_t A^T (B_{O_{t+1},:} \odot \widehat{\beta}_{t+1,:})$ for $t = T-1, \ldots, 1$

(Above, A^T is the *transpose* of A, not the T-th power of A.)

It turns out that

$$\mathbb{P}(\mathbf{x}_t = i | O, \lambda) = \frac{\widehat{\alpha}_{t,i}\widehat{\beta}_{t,i}}{\sum_{j=1}^{N} \widehat{\alpha}_{t,j}\widehat{\beta}_{t,j}}$$

and so we can easily compute the most likely state at time t by

$$\mathbf{x}_t^* = \text{argmax}_i \widehat{\alpha}_{t,i}\widehat{\beta}_{t,i}.$$

This is the solution to the second question posed at the beginning of the lab.

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

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

     Parameters
```

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

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

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

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

Computing the δ and γ Probabilities

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

$$\widehat{\delta}_{t,i,j}=\frac{\widehat{\alpha}_{t,i}A_{j,i}B_{O_{t+1},j}\widehat{\beta}_{t+1,j}}{\sum_{k,l}\widehat{\alpha}_{t,k}A_{l,k}B_{O_{t+1},l}\widehat{\beta}_{t+1,l}}$$

for $t=1,\ldots,T-1$ and $i,j=1,\ldots,N$,

$$\widehat{\gamma}_{t,i}=\sum_{j=1}^{N}\widehat{\delta}_{t,i,j}$$

for $t=1,\ldots,T-1$ and $i=1,\ldots,N$, and finally

$$\widehat{\gamma}_{T,:}=\frac{\widehat{\alpha}_{T,:}\odot\widehat{\beta}_{T,:}}{\langle\widehat{\alpha}_{T,:},\widehat{\beta}_{T,:}\rangle}.$$

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

```
def _delta(self, obs, alpha, beta):
```

```
    """
    Compute the delta probabilities.

    Parameters
    ----------
    obs : ndarray of shape (T,)
        The observation sequence
    alpha : ndarray of shape (T,N)
        The scaled forward probability matrix from the forward pass
    beta : ndarray of shape (T,N)
        The scaled backward probability matrix from the backward pass

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

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

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

```
>>> delta, gamma = h._delta(obs, alpha, beta)
>>> print delta
[[[ 0.14166321  0.0465066 ]
  [ 0.37776855  0.43406164]]

 [[ 0.17015868  0.34927307]
  [ 0.05871895  0.4218493 ]]

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

Choosing Better Parameters

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

$$\mathbb{P}(O|\,\lambda') \geq \mathbb{P}(O|\,\lambda).$$

The update formulas are given by

$$A'_{i,j} = \frac{\sum_{t=1}^{T-1} \widehat{\delta}_{t,j,i}}{\sum_{t=1}^{T-1} \widehat{\gamma}_{t,j}}$$
$$B'_{i,j} = \frac{\sum_{t=1}^{T} \widehat{\gamma}_{t,j} 1_{\{O_t=i\}}}{\sum_{t=1}^{T} \widehat{\gamma}_{t,j}}$$
$$\pi' = \widehat{\gamma}_{1,:}$$

where $1_{\{O_t=i\}}$ is one if $O_t = i$ and zero otherwise.

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

```
def _estimate(self, obs, delta, gamma):
    """
    Estimate better parameter values.

    Parameters
    ----------
    obs : ndarray of shape (T,)
        The observation sequence
    delta : ndarray of shape (T-1,N,N)
        The delta probability array
    gamma : ndarray of shape (T,N)
        The gamma probability array
    """
    # update self.A, self.B, self.pi in place
    pass
```

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

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

Fitting the Model

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

- Randomly initialize parameters $\lambda = (A, B, \pi)$
- Compute $\log \mathbb{P}(O|\,\lambda)$
- For $i = 1, 2, \ldots, K$:
  - Run forward pass
  - Run backward pass
  - Compute δ probabilities
  - Update model parameters
  - Compute $\log \mathbb{P}(O|\,\lambda)$ according to new parameters
  - If change in log probabilities is less than ε, break
  - Else, continue

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

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

The learning algorithm is essentially an optimization over the parameter space (i.e. the space of tuples of stochastic arrays having the proper dimensions) with respect to the objective function $\mathbb{P}(O|\,\lambda)$. The algorithm is guaranteed to increase the objective function at each iteration, so it is sure to converge. However, the objective function is riddled with local maxima, and so the outcome depends heavily on the randomly selected starting values for A, B, and π. Figure 7.2 illustrates the issues involved. The log probability stays approximately constant for the first 100 iterations. This indicates that the algorithm is not exploring the parameter space enough, and the parameters found at the 100-th iteration are virtually the same as those found at the first or second iteration. After the first 100 iterations, however, the algorithm is finally able to explore more of the parameter space and hence make better progress toward increasing the objective function. The moral of the story is that you may need to train the HMM a few times, using different starting values, and then keep the model that has the highest log likelihood.

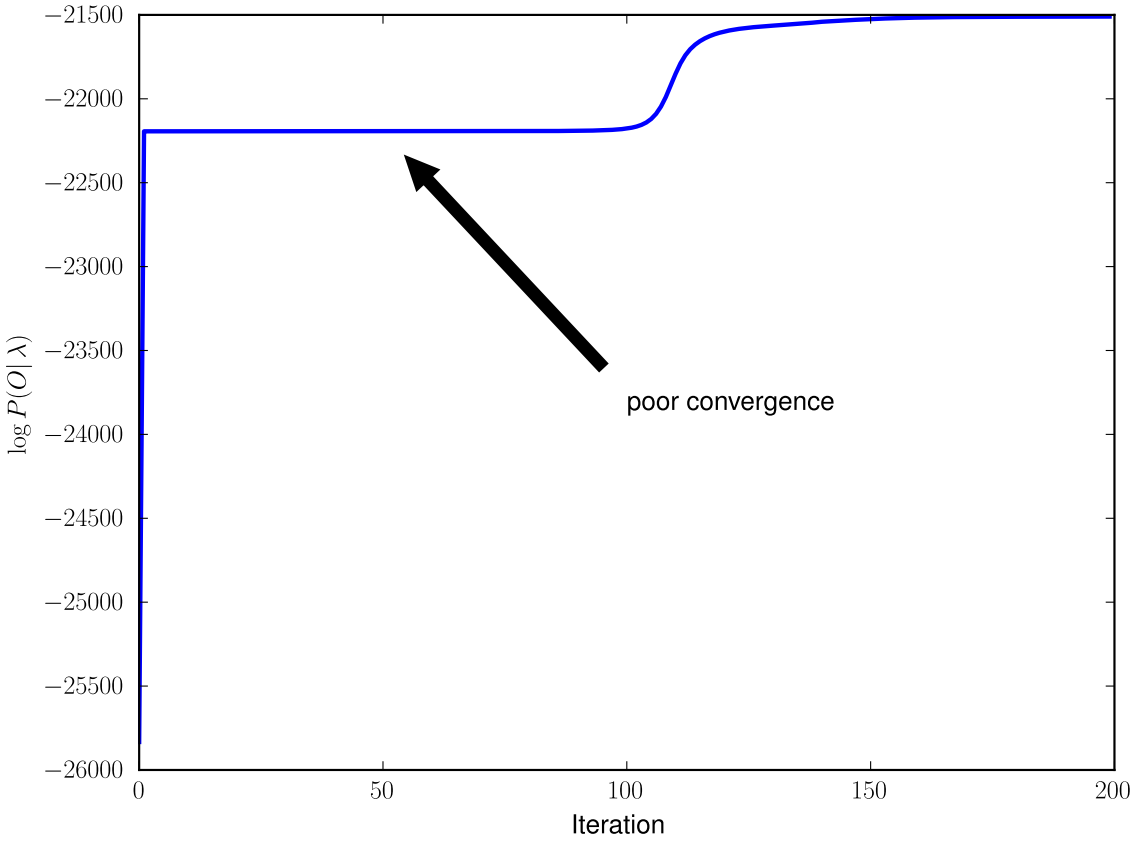


Figure 3.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

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

```
def fit(self, obs, A, B, pi, max_iter=100, tol=1e-3):
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    ----------
    obs : ndarray of shape (T,)
        Observation sequence on which to train the model.
    A : stochastic ndarray of shape (N,N)
        Initialization of state transition matrix
    B : stochastic ndarray of shape (M,N)
        Initialization of state observation matrix
    pi : stochastic ndarray of shape (N,)
        Initialization of initial state distribution
```

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

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

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


>>> def vec_translate(a, my_dict):

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


>>> def prep_data(filename):

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

>>>     # remove punctuation and newlines
>>>     remove_punct_map = {ord(char): None for char in string.punctuation+"\n\↩
    r"}
>>>     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 200 iterations with the default value for `tol`. Generally speaking, if you converge to a log probability greater than -21550, then you have reached an acceptable set of parameters for this dataset.

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

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

```
y, 0.0147, 0.0058
x, 0.0022, 0.0000
z, 0.0010, 0.0000
```

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

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

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

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

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

4 Gaussian Mixture Models

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

You've already seen GMMs as the observation distribution in certain continuous density HMMs. Here, we will discuss them further and learn how to estimate their parameters, given data.

The main idea behind a mixture model is contained in the name, i.e. it is a *mixture* of different models. What do we mean by a mixture? A mixture model is composed of K *components*, each component being responsible for a portion of the data. The responsibilities of these components are represented by mixture *weights* w_i, for $i = 1, \cdots, k$. As you may have guessed, these weights are nonnegative and sum to 1. Thus component j is responsible for $100 \cdot w_j$ percent of the data generated by the model.

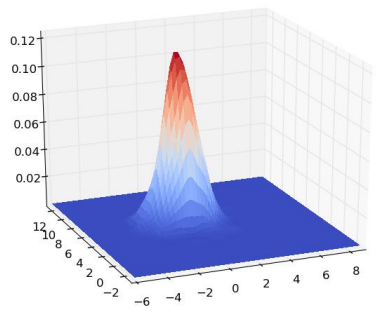
Each component is itself a probability distribution. In a GMM, each component is specifically a Gaussian (multivariate normal) distribution. Thus we additionally have parameters μ_i, Σ_i for $i = 1, \cdots, K$, i.e. a mean and covariance for each component in the GMM. It is important here to keep in mind that a GMM does not arise from adding weighted multivariate normal random variables, but rather from weighting the responsibility of each multivariate normal random variable. In the first case, we would simply have a different multivariate normal distribution, whereas in the second case we have a mixture. Refer to Figure **??** for a visualization of this.

Thus, a fully defined GMM has parameters $\lambda = (w, \mu, \mathbf{\Sigma})$. The density of a GMM is given by $\mathbb{P}(x|\lambda) = \sum_{i=1}^{K} w_i \mathcal{N}(x; \mu_i, \Sigma_i)$ where

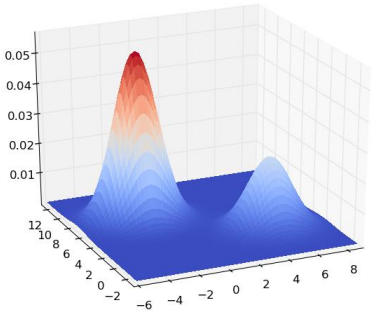
$$\mathcal{N}(x; \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{\frac{K}{2}} |\Sigma_i|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1} (x-\mu_i)}$$

Problem 1. Write a function to evaluate the density of a normal distribution at a point x, given parameters μ and Σ. Include the option to return the log of this probability, but be sure to do it intelligently! Also write a function that computes the density of a GMM at a point x, given the parameters λ, along with the log option.

Throughout this lab, we will build a GMM class with various methods. We will outline this now.



(a) Sum of weighted multivariate normal random variables.



(b) Weighted mixture of multivariate normal random variables.

Problem 2. Write the skeleton of a GMM class. In the `__init__` method, it should accept the non-null parameter *n_ components*, as well as parameters for the weights, means, and covariance matrices which define the GMM. Include a function to generate data from a fully defined GMM (you may use your code from the CDHMM lab for this), as well as the density function you recently defined.

The main focus of this lab will be to estimate the parameters of a GMM, given observed multivariate data $Y = y_1, y_2, \cdots, y_T$. This can be done via Gibbs sampling, as well as with EM (Expectation Maximization). We choose the latter approach for this lab. To do this, we must compute the probability of an observation being from each component of a GMM with parameters $\lambda^{(n)} = \left(w^{(n)}, \mu^{(n)}, \Sigma^{(n)}\right)$. This is simply

$$\mathbb{P}(x_t = i | y_t, \lambda) \propto w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$$

Just as with HMMs, we refer to these probabilities as $\gamma_t(i)$, and this is the E-step in the algorithm. This might seem straightforward, except this direct computation will likely lead to numerical issues. Instead, we work in the log space, which means we have to be a bit more careful.

It is feasible (and occurs quite often) that each term $w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$ is 0, because of underflow in the computation of the multivariate normal density. Letting $l_i^{(n)} = \ln w_i^{(n)} + \ln \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$, we can compute these probabilities more carefully, as follows:

$$\begin{aligned}
\mathbb{P}(x_t = i | y_t, \lambda) &= \frac{e^{l_i}}{\sum_{j=1}^{K} e^{l_j}} \\
&= \frac{e^{l_i} e^{-\max_k l_k}}{\sum_{j=1}^{K} e^{l_j} e^{-\max_k l_k}} \\
&= \frac{e^{l_i - \max_k l_k}}{\sum_{j=1}^{K} e^{l_j - \max_k l_k}}
\end{aligned}$$

which will effectively avoid underflow problems.

Problem 3. Add a method to your class to compute $\gamma_t(i)$ for $t = 1, \cdots, T$ and $i = 1, \cdots, K$. Don't forget to do this intelligently to avoid underflow!

Given our matrix γ, we can reestimate our weights, means, and covariance matrices as follows:

$$w_i^{(n+1)} = \sum_{t=1}^{T} \gamma_t(i)$$
$$\mu_i^{(n+1)} = \frac{\sum_{t=1}^{T} \gamma_t(i) y_t}{\sum_{t=1}^{T} \gamma_t(i)}$$
$$\Sigma_i^{(n+1)} = \frac{\sum_{t=1}^{T} \gamma_t(i)(y_t - \mu_i^{(n+1)})(y_t - \mu_i^{(n+1)})^T}{\sum_{t=1}^{T} \gamma_t(i)}$$

for $i = 1, \cdots, K$. These updates are the M-step in the algorithm.

Problem 4. Add methods to your class to update w, μ and Σ as described above.

With the above work, we are almost ready to complete our class. To train, we will randomly initialize our parameters λ, and then iteratively update them as above.

Problem 5. Add a method to initialize λ. Do this intelligently, i.e. your means should not be far from your actual data used for training, and your covariances should neither be too big nor too small. Your weights should roughly be equal, and still sum to 1. Also add a method to train your model, as described previously, iterating until convergence within some tolerance.

We will use our work to train the "Mickey Mouse" GMM, which has parameters

$$w = \begin{bmatrix} 0.7 & 0.15 & 0.15 \end{bmatrix}$$
$$\mu_1 = \begin{bmatrix} 0.0 & 0.0 \end{bmatrix}$$
$$\mu_2 = \begin{bmatrix} -1.5 & 2.0 \end{bmatrix}$$
$$\mu_3 = \begin{bmatrix} 1.5 & 2.0 \end{bmatrix}$$
$$\Sigma_1 = I_3$$
$$\Sigma_2 = 0.25 \cdot I_3$$
$$\Sigma_3 = 0.25 \cdot I_3$$

To look at this GMM, we will evaluate the density at each point on a grid, as follows:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(-3, 3, 0.1)
>>> y = np.arange(-2, 3, 0.1)
>>> X, Y = np.meshgrid(x, y)
>>> N, M = X.shape
>>> immat = np.array([[model.dgmm(np.array([X[i,j],Y[i,j]])) for j in xrange(M)↩
    ] for i in xrange(N)])
```

```
>>> plt.imshow(immat, origin='lower')
>>> plt.show()
```

See Figure 4.2 for this plot.

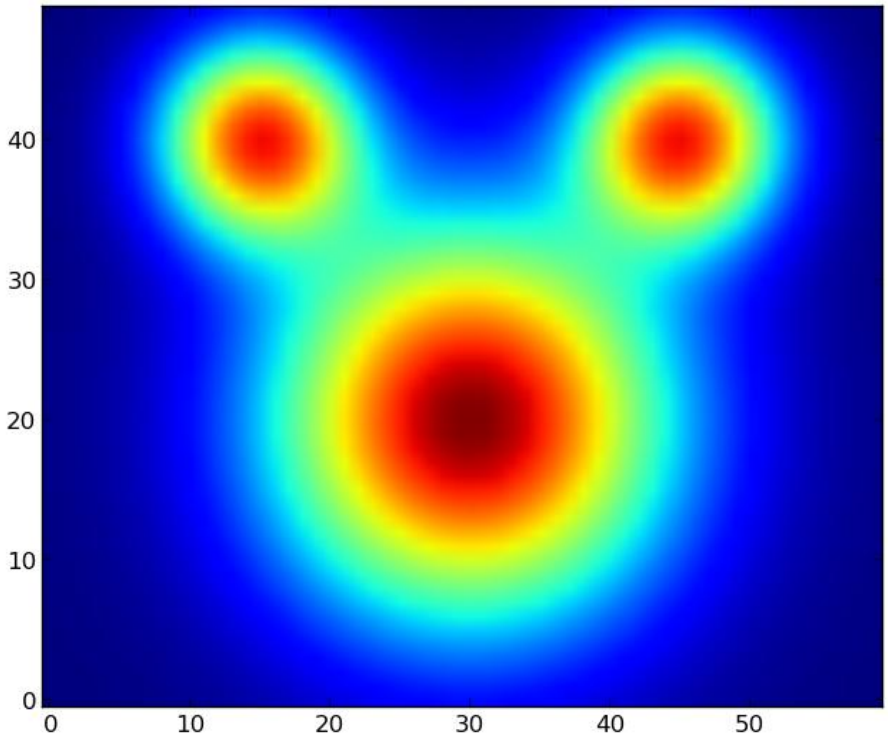


Figure 4.2: Density of true "Mickey Mouse" GMM.

Problem 6. Generate 750 samples from the above mixture model. Using just the drawn samples, retrain your model. Evaluate and plot your density on the grid used above. How similar is your density to the original?

How close is our trained model to the original one? We can use the symmetric Kullback-Liebler divergence to measure the distance between two probability distributions with densities $p(x)$ and $p'(x)$:

$$SKL(p, p') = \left| \frac{1}{2} \int p(x) \ln \frac{p(x)}{p'(x)} dx + \frac{1}{2} \int p'(x) \ln \frac{p'(x)}{p(x)} dx \right|$$

We cannot analytically compute this, so we use a Monte Carlo approximation, which uses the fact that

$$\frac{1}{N} \sum_{i=1}^{N} f(x_i) \to \int f(x) p(x) dx$$

as $N \to \infty$, assuming that each $x_i \sim p$. Then we have the following approximation of the symmetric KL divergence:

$$SKL(p, p') \approx \frac{1}{2N} \left| \sum_{i=1}^{N} \ln \frac{p(x_i)}{p'(x_i)} + \sum_{i=1}^{N} \ln \frac{p'(x'_i)}{p(x'_i)} \right|$$

where $x_i \sim p$ and $x'_i \sim p'$, for large N.

Problem 7. Write a function to compute the approximate the SKL of two GMMs. Compute the SKL between a randomly initialized GMM and the known GMM. Compute the SKL between the trained GMM and the known GMM. Is our trained model a good fit?

5 Speech Recognition using CDHMMs

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

5.0.1 Continuous Density Hidden Markov Models

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

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

$$f(x) = \sum_{i=1}^{M} c_i N(x;\, \mu_i, \Sigma_i),$$

where $N(\cdot;\, \mu_i, \Sigma_i)$ denotes the probability density function for the normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$. See Figure 5.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

In a GMMHMM, we seek to model a hidden state sequence $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ and a corresponding observation sequence $\{O_1, \dots, O_T\}$, just as with discrete HMMs. The major difference, of course, is that each observation O_t is a real-valued vector of length K distributed according to a mixture of Gaussians with M components. The parameters for such a model include the initial state distribution π and the state transition matrix A (just as with discrete HMMs). Additionally, for each state $i = 1, \dots, N$, we have component weights $\{c_{i,1}, \dots, c_{i,M}\}$, component means $\{\mu_{i,1}, \dots, \mu_{i,M}\}$, and component covariance matrices $\{\Sigma_{i,1}, \dots, \Sigma_{i,M}\}$.

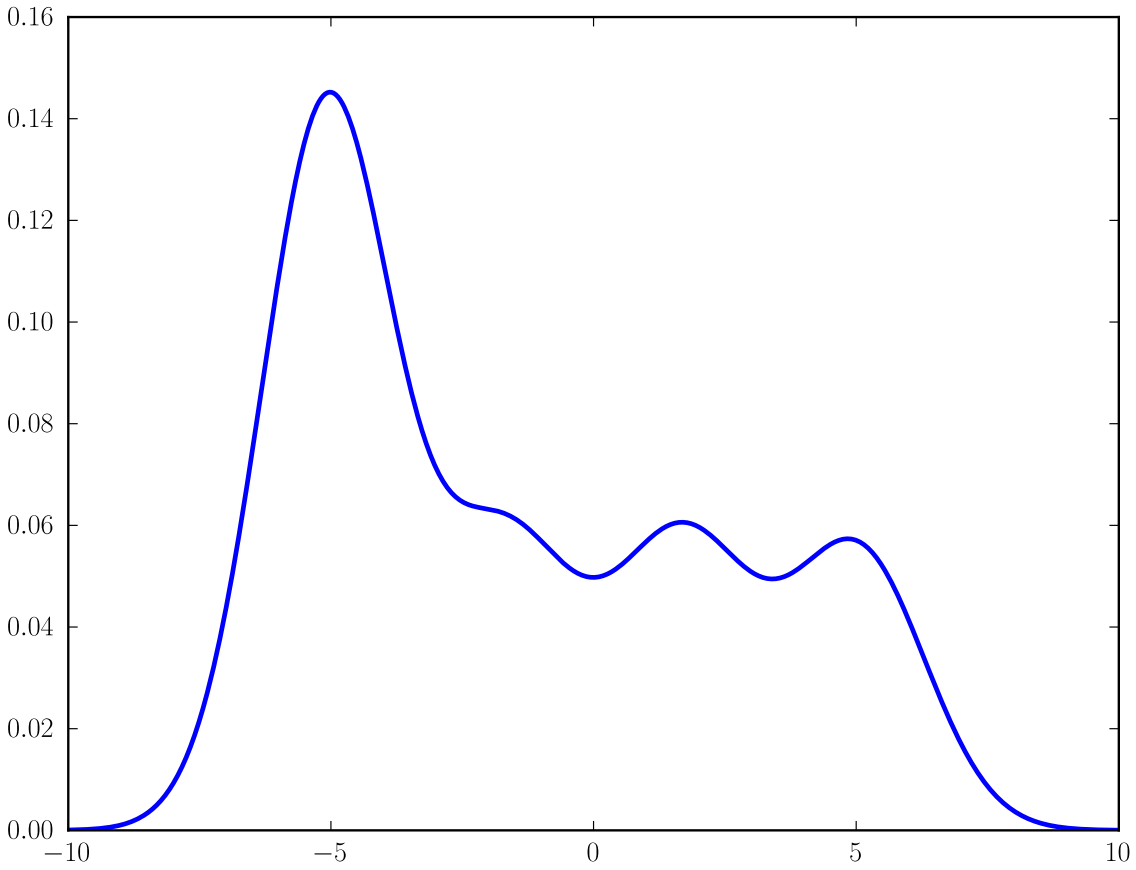


Figure 5.1: The probability density function of a mixture of Gaussians with four components.

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

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

We can draw a random sample from the GMMHMM corresponding to the second state as follows:

```
>>> sample_component = np.argmax(np.random.multinomial(1, weights[1,:]))
```

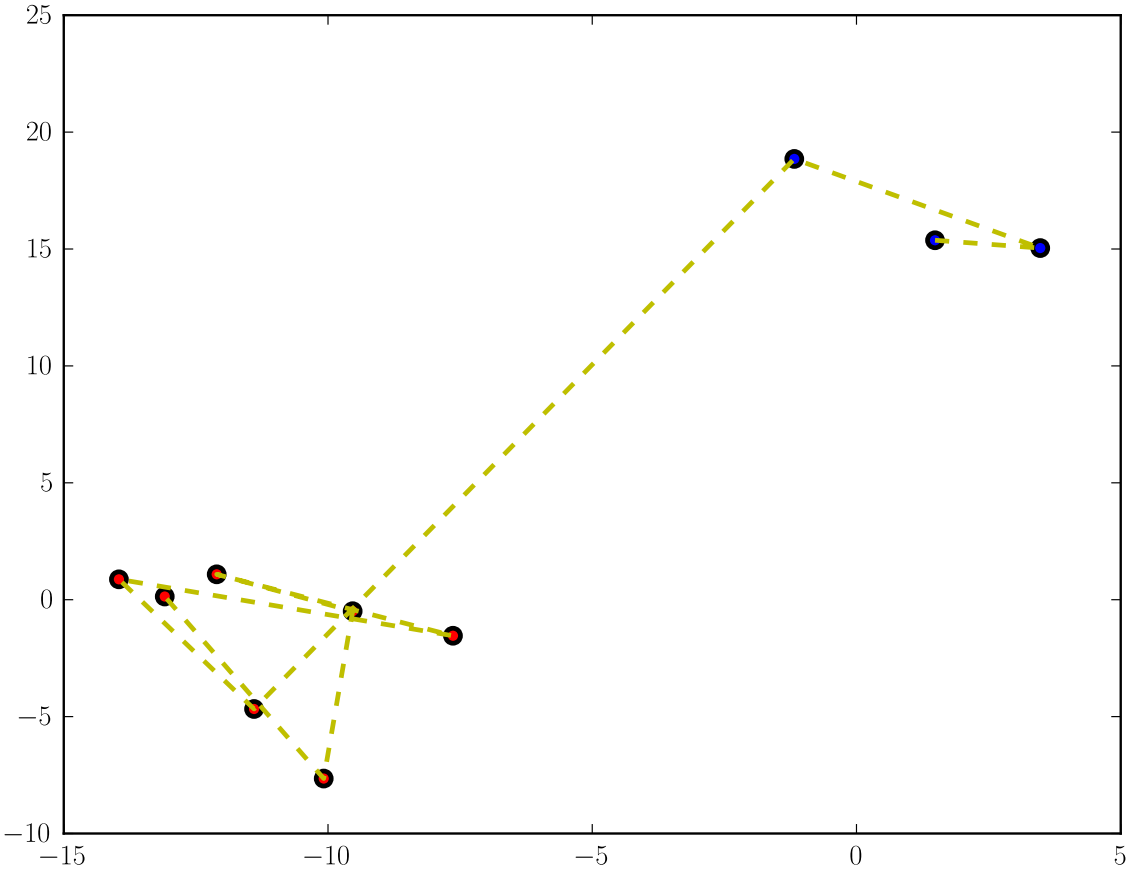


Figure 5.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

```
>>> sample = np.random.multivariate_normal(means[1, sample_component, :], ↩
    covars[1, sample_component, :, :])
```

Figure 5.2 shows an observation sequence generated from a GMMHMM with one mixture component and two states.

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

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

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

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

Speech Recognition and Hidden Markov Models

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

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

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

Problem 2. Obtain 30 (or more) recordings for each of the words/phrases *mathematics*, *biology*, *political science*, *psychology*, and *statistics*. These audio samples should be 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. Load the recordings into Python using `scipy.io.wavfile.read`.

If the audio files have two channels, average these channels to obtain an array of length 88200 for each sample. Extract the MFCCs from each sample using code from the file `MFCC.py`:

```
>>> import MFCC
>>> # assume sample is an array of length 88200
>>> mfccs = MFCC.extract(sample)
```

Store the MFCCs for each word in a separate list. You should have five lists, each containing 50 MFCC arrays, corresponding to each of the five words under consideration.

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

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

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

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

Problem 3. Partition each list of MFCCs into a training set of 20 samples, and a test set of the remaining 10 samples.

Using the training sets, train a GMMHMM on each of the words from the previous problem with at least 10 random restarts, keeping the best model for each word (the one with the highest log-likelihood). This process may take several minutes. Since you will not want to run this more than once, you will want to save the best model for each word to disk using the `pickle` module so that you can use it later.

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

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

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

Problem 4. Classify the 10 test samples for each word. How does your system perform? Which words are the hardest to correctly classify? Make a dictionary containing the accuracy of the classification of your five testing sets. Specifically, the words/phrases will be the keys, and the values will be the percent accuracy.

6 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 6.1 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 6.1. 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.csv`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean μ and variance σ^2. Because we are unsure of the true value of μ and σ^2, we take a Bayesian approach and place priors on each parameter to quantify this uncertainty:

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

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

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

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

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

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

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

where

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

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

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

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

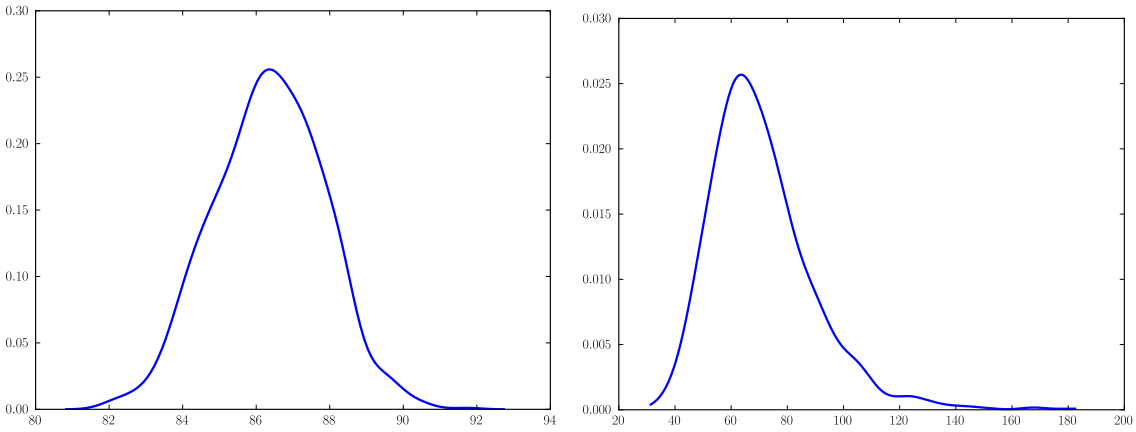
Problem 1. Implement a Gibbs sampler for the exam scores problem using the following function declaration.

```
def gibbs(y, nu, tau2, alpha, beta, n_samples):
    """
    Assuming a likelihood and priors
        y_i    ~ N(mu, sigma2),
        mu     ~ N(nu, tau2),
        sigma2 ~ IG(alpha, beta),
    sample from the posterior distribution
        P(mu, sigma2 | y, nu, tau2, alpha, beta)
    using a gibbs sampler.

    Parameters
    ----------
    y : ndarray of shape (N,)
        The data
    nu : float
        The prior mean parameter for mu
    tau2 : float > 0
        The prior variance parameter for mu
    alpha : float > 0
        The prior alpha parameter for sigma2
    beta : float > 0
        The prior beta parameter for sigma2
    n_samples : int
        The number of samples to draw

    Returns
    -------
    samples : ndarray of shape (n_samples,2)
        1st col = mu samples, 2nd col = sigma2 samples
    """
    pass
```

Test it with priors $\nu = 80, \tau^2 = 16, \alpha = 3, \beta = 50$, collecting 1000 samples. Plot your samples of μ and your samples of σ^2. How long did it take for each to converge? It should have been very quick.

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

```
>>> import numpy as np
>>> from scipy.stats import gaussian_kde
>>> import matplotlib.pyplot as plt
>>> mu_kernel = gaussian_kde(mu_samples)
>>> x_min = min(mu_samples) - 1
```



(a) Posterior distribution of μ.

(b) Posterior distribution of σ^2.

Figure 6.1: Posterior marginal probability densities for μ and σ^2.

```
>>> x_max = max(mu_samples) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
```

Problem 2. Plot the kernel density estimators for the posterior distributions of μ and σ^2. You should get plots similar to those in Figure 6.1.

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

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

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

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

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

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.

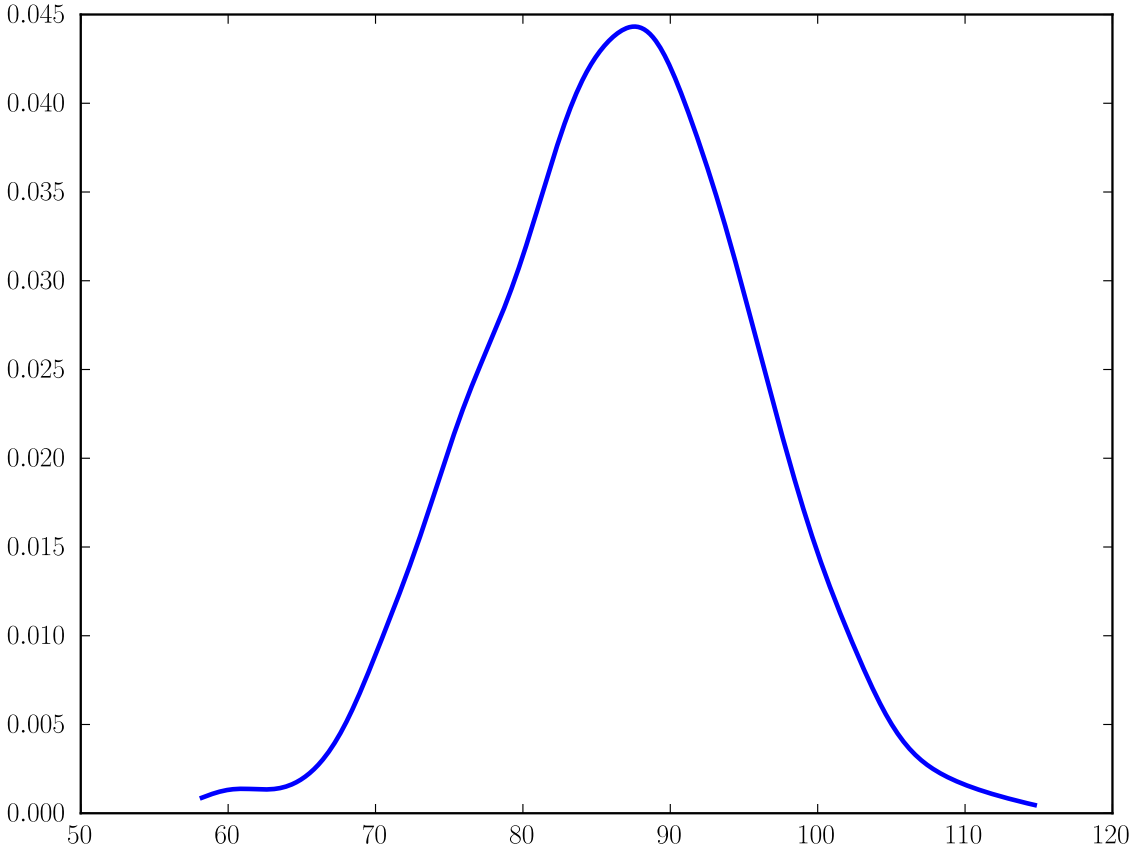


Figure 6.2: Predictive posterior distribution of exam scores.

Problem 3. Use your samples of μ and σ^2 to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. It should resemble the plot in Figure 6.2.

Latent Dirichlet Allocation

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

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

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

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

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

This is typically depicted with graphical plate notation as in Figure 6.3.

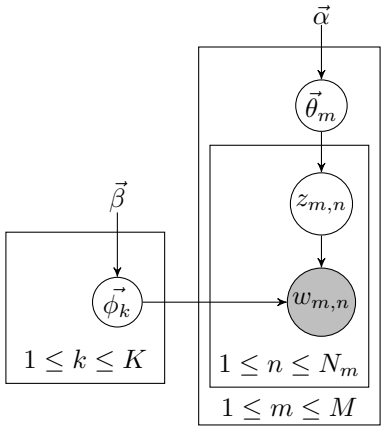


Figure 6.3: Graphical plate notation for LDA text generation.

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

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

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

These conditional distributions have the following form:

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

where

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

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

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

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

Problem 4. Complete the method `initialize`. By randomly assigning initial topics, fill in the count matrices and topic assignment dictionary. In this method, you will initialize the count matrices (among other things). Note that the notation provided in the code is slightly different than that used above. Be sure to understand how the formulae above connect with the code.

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

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

Problem 5. Complete the method `_sweep`, which needs to iterate through each word of each document. It should call on the method `_conditional` to get the conditional distribution at each iteration.

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

We are now prepared to write the full Gibbs sampler.

Problem 6. Complete the method `sample`. The argument *filename* is the name and location of a .txt file, where each line is considered a document. The corpus is built by method `buildCorpus` , and stopwords are removed (if argument *stopwords* is provided). Burn in the Gibbs sampler, computing and saving the log-likelihood with the method `_loglikelihood`. After the burn in, iterate further, accumulating your count matrices, by adding `nzw` and `nmz` to `total_nzw` and `total_nmz` respectively, where you only add every $sample\_rate^{th}$ iteration. Also save each log-likelihood.

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

Problem 7. Create an `LDACGS` object with 20 topics, letting `alpha` and `beta` be the default values. Load in the stop word list provided. Run the Gibbs sampler, with a burn in of 100 iterations, accumulating 10 samples, only keeping the results of every 10th sweep. Plot the log-likelihoods. How long did it take to truly burn in?

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

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

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

Problem 8. Using the methods described above, examine the topics for Reagan's addresses. As best as you can, come up with labels for each topic. Note that if $ntopics = 20$ and $n = 10$, we will get the top 10 words that represent each of the 20 topics. What you will want to do for each topic is decide what these ten words jointly represent represent. Save your topic labels in a list or an array.

We can use $\widehat{\theta}$ 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.

7 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 $f(\theta) = c \cdot h(\theta)$ for some nonzero constant c (in practice, we assume that f is an easy function to evaluate, while h is difficult). Let $Q : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a symmetric *proposal function* (so that $Q(\cdot, y)$ is a probability density function for all $y \in \mathbb{R}^n$, and $Q(x, y) = Q(y, x)$ for all $x, y \in \mathbb{R}^n$) and let $A : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be an *acceptance function* defined by

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

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 7.1. The Metropolis algorithm can be interpreted as follows: given our current state y, we propose a new state according to the distribution $Q(\cdot, y)$. We then accept or reject it according to A. We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h. Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

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

Algorithm 7.1 Metropolis Algorithm

procedure METROPOLIS ALGORITHM
  Choose initial point x_0.
  for $t = 1, 2, \ldots$ **do**
    Draw $x' \sim Q(\cdot, x_{t-1})$
    Draw $a \sim \text{unif}(0, 1)$
    if $a \leq A(x', x_{t-1})$ **then**
      $x_t = x'$
    else
      $x_t = x_{t-1}$
  Return $x_1, x_2, x_3, \ldots$

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

$$\mu \sim N(\mu_0 = 80, \sigma_0^2 = 16)$$
$$\sigma^2 \sim IG(\alpha = 3, \beta = 50).$$

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

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

. However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to μ and σ^2, the numerator can serve as the function f in the Metropolis algorithm, and the denominator can serve as the constant c. We choose our proposal function to be based on a bivariate Normal distribution:

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

where I is the 2×2 identity matrix and s is some positive scalar. Let's create these functions in Python:

```
import numpy as np
from math import sqrt, exp, log
import scipy.stats as st
from matplotlib import pyplot as plt
from scipy.stats import gaussian_kde

# load in the data
scores = np.loadtxt('examscores')

# initialize the hyperparameters
alpha = 3
beta = 50
mu0 = 80
sig20 = 16
```

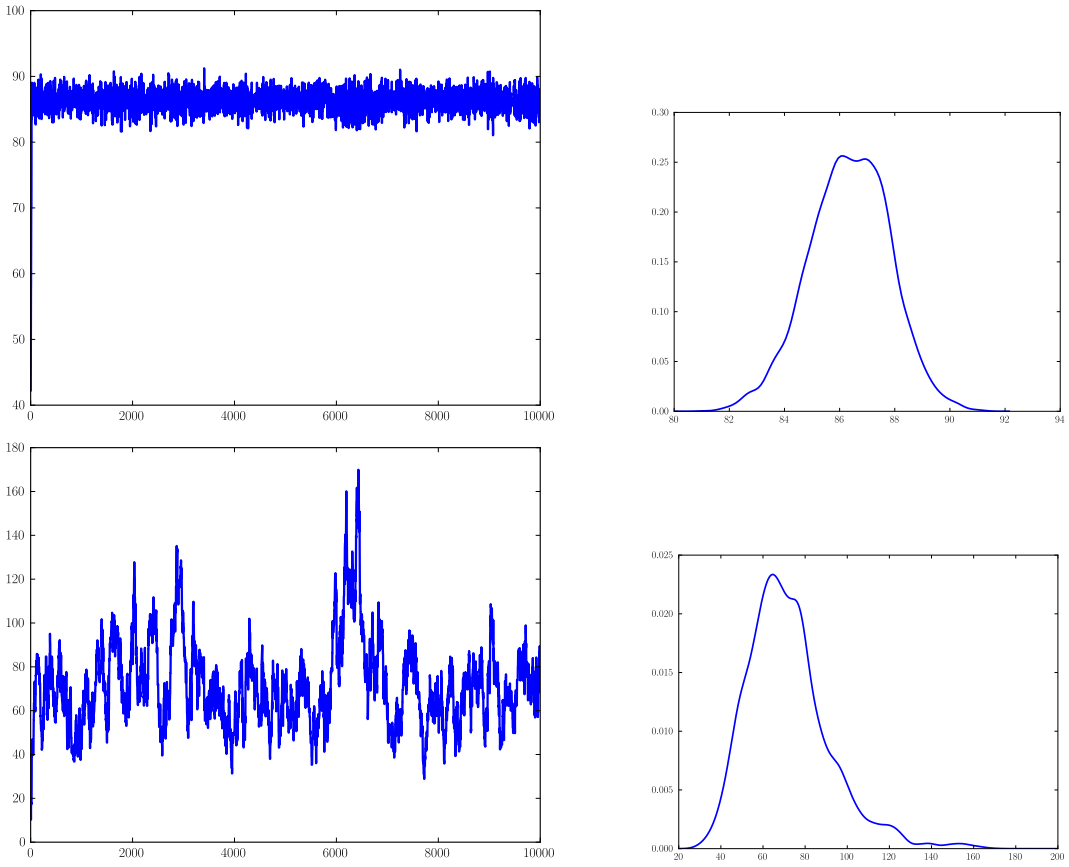


Figure 7.1: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

```
# initialize the prior distributions
muprior = st.norm(loc=mu0, scale=sqrt(sig20))
sig2prior = st.invgamma(alpha,scale=beta)

# define the proposal function
def proposal(y, s):
    return st.multivariate_normal.rvs(mean=y, cov=s*np.eye(len(y)))

# define the log of the proportional density
def propLogDensity(x):
    return muprior.logpdf(x[0])+sig2prior.logpdf(x[1])+st.norm.logpdf(scores,↩
        loc=x[0],scale=sqrt(x[1])).sum()
```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log densities of the samples and the proportion of proposed samples that were accepted. Study the implementation below to make sure you understand the process:

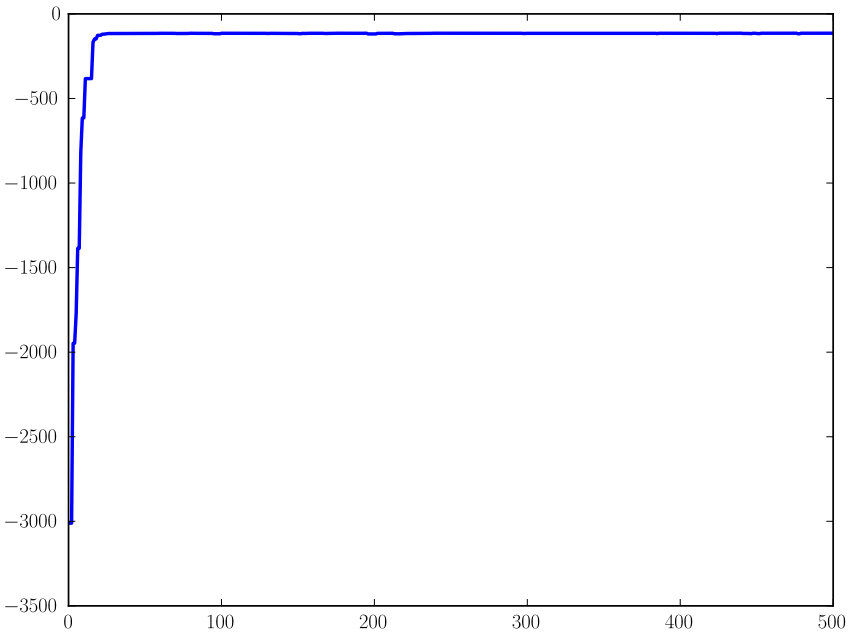


Figure 7.2: Log densities of the first 500 Metropolis samples.

```
def metropolis(x0, s, n_samples):
    """
    Use the Metropolis algorithm to sample from posterior.

    Parameters
    ----------
    x0 : ndarray of shape (2,)
        The first entry is mu, the second entry is sigma2
    s : float > 0
        The standard deviation parameter for the proposal function
    n_samples : int
        The number of samples to generate

    Returns
    -------
    draws : ndarray of shape (n_samples, 2)
        The MCMC samples
    logprobs : ndarray of shape (n_samples)
        The log density of the samples
    accept_rate : float
        The proportion of proposed samples that were accepted
    """
    accept_counter = 0
    draws = np.empty((n_samples,2))
    logprob = np.empty(n_samples)
```

```
    x = x0.copy()
    for i in xrange(n_samples):
        xprime = proposal(x,s)
        u = np.random.rand(1)[0]
        if log(u) <= propLogDensity(xprime) - propLogDensity(x):
            accept_counter += 1
            x = xprime
        draws[i] = x
        logprob[i] = propLogDensity(x)
    return draws, logprob, accept_counter/float(n_samples)
```

Now let's sample from the posterior. We will choose an initial guess of $\mu = 40$ and $\sigma^2 = 10$, and we will set $s = 20$. We draw 10000 samples as follows:

```
>>> draws, lprobs, rate = metropolis(np.array([40, 10], dtype=float), 20., ↩
    10000)
>>> print "Acceptance Rate:", r
Acceptance Rate: 0.3531
```

We can evaluate the quality of our results by plotting the log probabilities, the μ samples, the σ^2 samples, and kernel density estimators for the marginal posterior distributions of μ and σ^2. The code below will accomplish this task:

```
>>> # plot the first 500 log probs
>>> plt.plot(lprobs[:500])
>>> plt.show()
>>> # plot the mu samples
>>> plt.plot(draws[:,0])
>>> plt.show()
>>> # plot the sigma2 samples
>>> plt.plot(draws[:,1])
>>> plt.show()
>>> # build and plot KDE for posterior mu
>>> mu_kernel = gaussian_kde(draws[50:,0])
>>> x_min = min(draws[50:,0]) - 1
>>> x_max = max(draws[50:,0]) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
>>> # build and plot KDE for posterior sigma2
>>> sig_kernel = gaussian_kde(draws[50:,1])
>>> x_min = 20
>>> x_max = 200
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,sig_kernel(x))
>>> plt.show()
```

Your results should be close to those given in Figures 7.1 and 7.2.

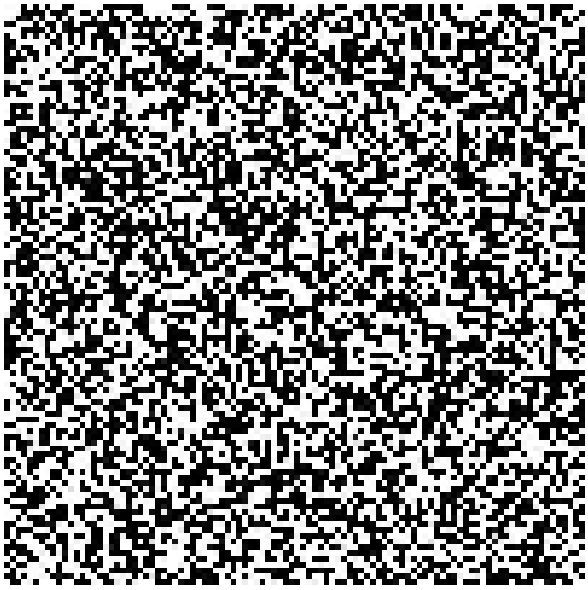


Figure 7.3: Spin configuration from random initialization.

The Ising Model

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

With any spin configuration σ, there is an associated energy

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

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

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

Problem 1. Write a function that initializes a spin configuration for an $n \times n$ lattice. It should return an $n \times n$ array, each entry of which is either 1 or -1, chosen randomly. Test this for the grid described above, and plot the spin configuration using `matplotlib.pyplot.imshow`. It should look fairly random, as in Figure 7.3.

Problem 2. Write a function that computes the energy of a wrap-around $n \times n$ lattice with a given spin configuration, as described above. Make sure that you do not double count site pair interactions!

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

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

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

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

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

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

By choosing our transition matrix Q cleverly, we can also make it easy to compute the energy for any proposed spin configuration. We restrict our possible proposals to only those spin configurations in which we have flipped the spin at exactly one lattice site, i.e. we choose a lattice site i and flip its spin. Thus, there are only L possible proposal spin configurations σ^* given σ, each being proposed with probability $\frac{1}{L}$, and such that $\sigma^*_j = \sigma_j$ for all $j \neq i$, and $\sigma^*_i = -\sigma_i$. Note that we would never actually write out this matrix (it would be $2^{10000} \times 2^{10000}$!!!). Computing the proposed site's energy is simple: if the spin flip site is i, then we have $H(\sigma^*) = H(\sigma) + 2\sum_{j:j\sim i} \sigma_i \sigma_j$.

Problem 3. Write a function that proposes a new spin configuration given the current spin configuration on an $n \times n$ lattice, as described above. This function simply needs to return a pair of indices (i, j), chosen with probability $\frac{1}{n^2}$.

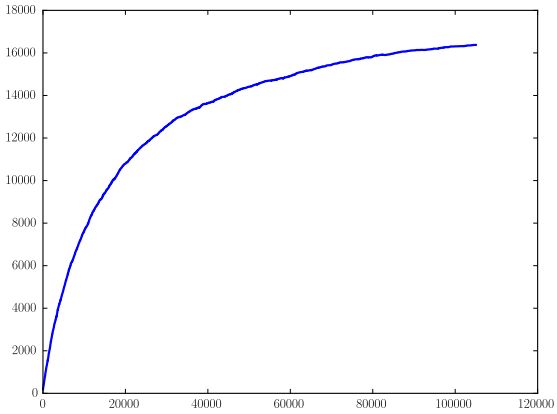
Problem 4. Write a function that computes the energy of a proposed spin configuration, given the current spin configuration, its energy, and the proposed spin flip site indices.

Problem 5. Write a function that accepts or rejects a proposed spin configuration, given the current configuration. It should accept the current energy, the proposed energy, and β, and should return a boolean.

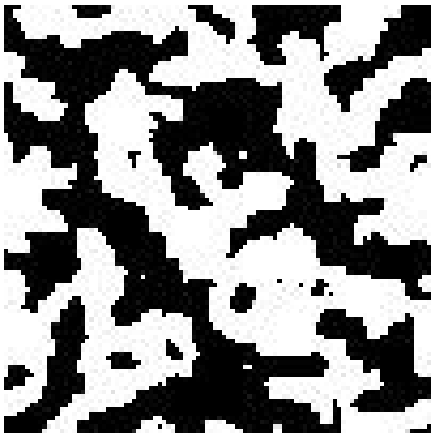
To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator Z_β, which–as we explained previously–is generally the reason we have to use a Metropolis algorithm to begin with. We can get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

Problem 6. Write a function that initializes a spin configuration for an $n \times n$ lattice as done previously, and then performs the Metropolis algorithm, choosing new spin configurations and accepting or rejecting them. It should burn in first, and then iterate *n_samples* times, keeping every 100^{th} sample (this is to prevent memory failure) and all of the above values for $-\beta H(\sigma)$ (keep the values even for the burn-in period). It should also accept β as an argument, allowing us to effectively adjust the temperature for the model.

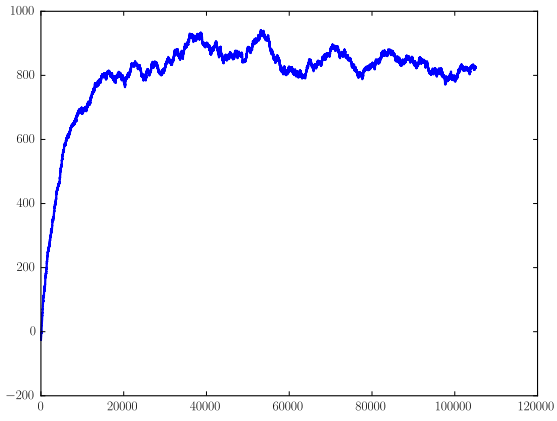
Problem 7. Test your Metropolis sampler on a 100×100 grid, with 200000 iterations, with *n_samples* large enough so that you will keep 50 samples, testing with $\beta = 1$ and then with $\beta = 0.2$. Plot the proportional log probabilities, and also plot a late sample from each test using `matplotlib.pyplot.imshow`. How does the ferromagnetic material behave differently with differing temperatures? Recall that β is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figures 7.4b and 7.4d.



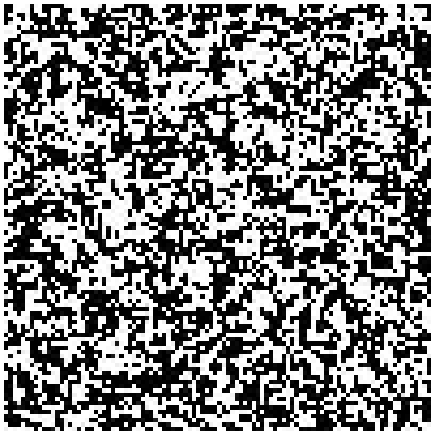
(a) Proportional log probs when $\beta = 1$.



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



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



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

8 Principal Component Analysis and Latent Semantic Indexing

Lab Objective: *Understand the basics of principal component analysis and latent semantic indexing.*

Principal Component Analysis

Understanding the variance in complex data is one of the first tasks encountered in exploratory data analysis. For an example, consider the scatter plot displaying the sepal and petal lengths of 100 different irises shown in Figure 8.1. There are three distinct types of iris flowers present: *setosa*, *versicolor*, and *virginica*. Considering this data, we might ask how to best distinguish the different types of irises based on their given sepal and petal lengths. We can answer this question by finding the characteristic that causes the greatest variance in the data. (Greater variance implies a greater ability to distinguish between data points. If the variance is very small, the data are clustered tightly together, and it is difficult to distinguish well.)

Upon examination, we see that the petal length ranges between 3 and 7 cm, while the sepal length only ranges between 5 and 8 cm. We might be tempted to say that the most distinguishing aspect of irises is their petal length, but this is only considering the features of the data individually, and not collectively. The two features of the data are clearly correlated, and a more careful consideration would lead us to conclude that the most distinguishing aspect of irises is their overall size. Some irises are are much larger than others, while the sepal and petal lengths stay roughly in proportion.

Principal Component Analysis (PCA) is a multivariate statistical tool used to orthogonally change the basis of a set of observations from the basis of original features (which may be correlated) into a basis of uncorrelated (in fact, orthonormal) variables called the *principal components*. It is a direct application of the singular value decomposition (SVD) from linear algebra. More specifically, the first principal component will account for the greatest variance in the set of observations, the second principal component will be orthogonal to the first, accounting for the second greatest variance in the set of observations, etc. The first several principal components capture most of the variance in the observation set, and hence provide a great deal of information about the data. By projecting the observations onto the space spanned by the principal components, we can reduce the dimensionality of the data in a manner that preserves most of the variance.

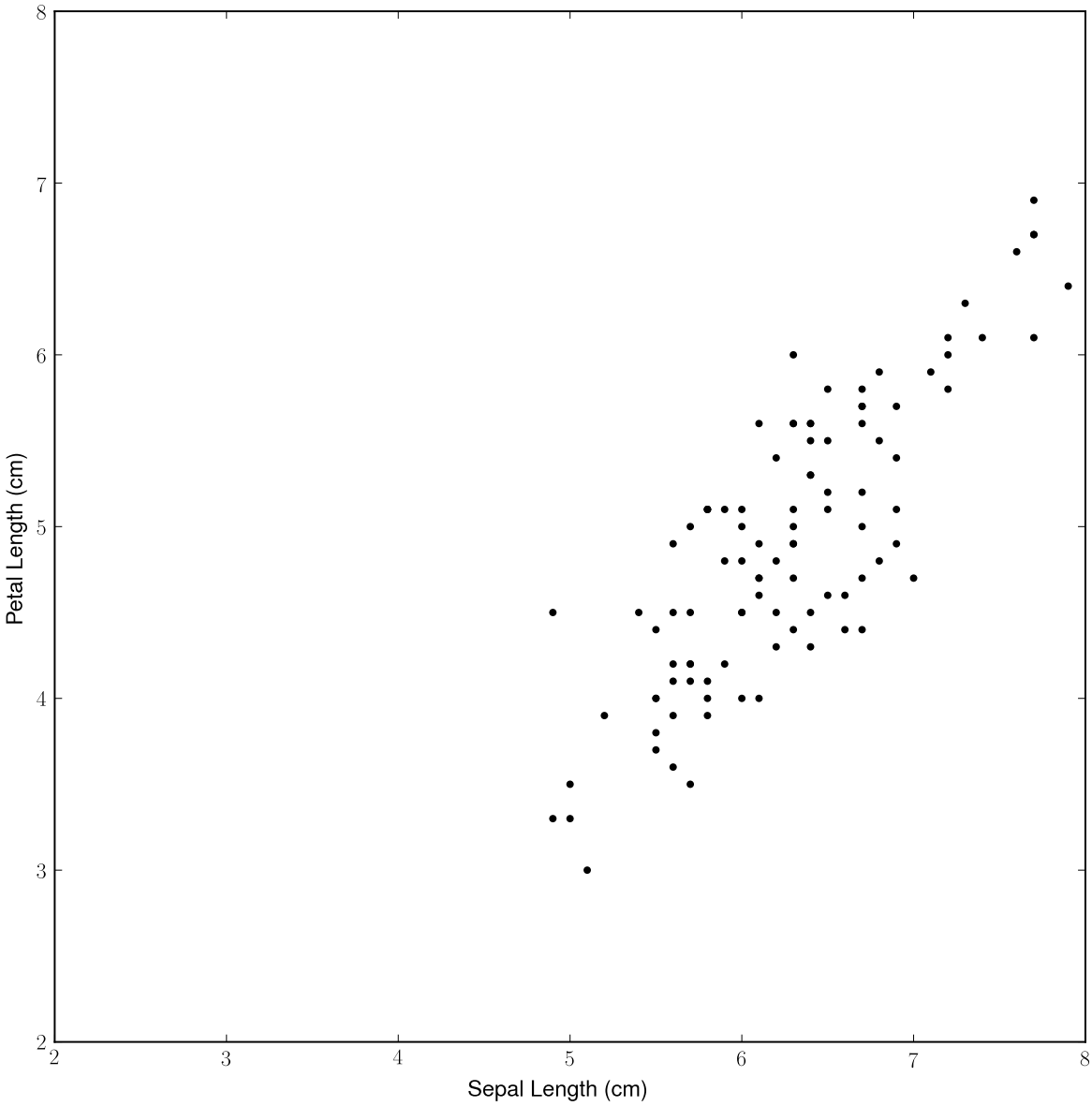


Figure 8.1: Sepal Length vs. Petal Length for 100 iris flowers. Note the strong correlation of these variables.

In our iris example, the two principal components are shown in Figure 8.2. The first principal component, corresponding intuitively to iris size, accounts for 96% of the variance in the data. The second, which accounts for only 4% of the variance, corresponds to the relative sepal and petal length of irises of the same size.

Computing the Principal Components

We now explore how to use the SVD to compute the principal components of a dataset. Throughout this lab we will use the `sklearn` iris data set, which can be obtained as follows:

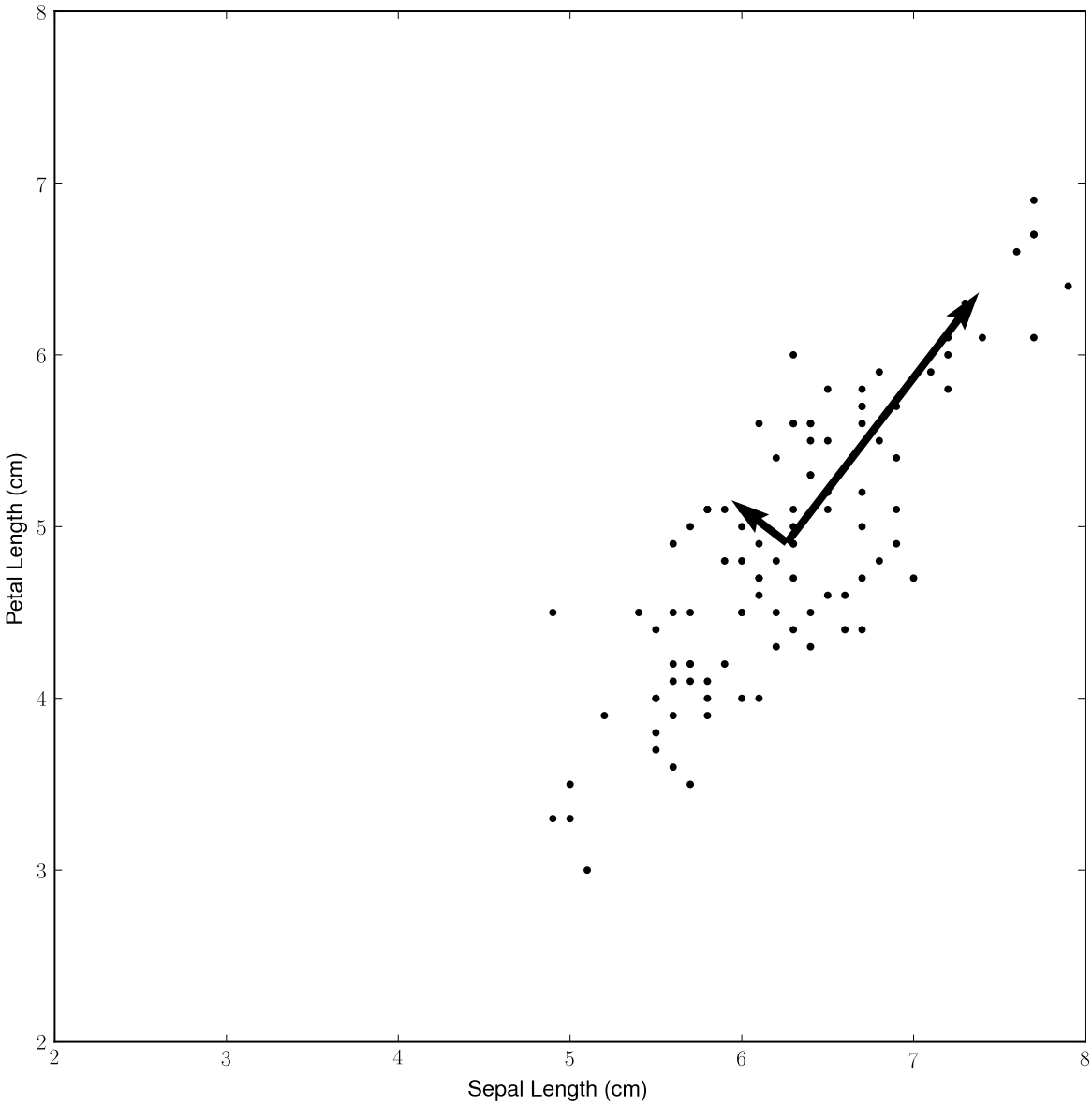


Figure 8.2: The vectors indicate the two principal components, which are weighted by their contribution to the variance.

```
>>> import numpy as np
>>> from scipy import linalg as la
>>> import sklearn.datasets as datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
```

We represent the collection of observations as an $n \times m$ matrix X, where each row of X is an observation, and each column is a specific feature. Let $k = \min(m, n)$. We will use this later. In the iris example, X contains 150 observations, each consisting of 4 features (so $k = 4$), as shown below:

```
>>> X.shape
(150L, 4L)
>>> iris.feature_names
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The first step in PCA is to pre-process the data. In particular, we first translate the columns of X to have mean 0. The data may then be optionally scaled to remove discrepancies arising from different units of measure (i.e. centimeters vs meters), and we call the new matrix containing the centered and scaled data Y. In this lab, we will not have any scaling issues, so we won't address this issue any further. Thus we can pre-process our iris data simply as follows:

```
>>> Y = X - X.mean(axis=0)
```

We next compute the truncated SVD of our centered and scaled data,

$$Y = U\Sigma V^T$$

where U is $n \times k$, Σ is a $k \times k$ diagonal matrix containing the singular values of Y in decreasing order along the diagonal, and V is $m \times k$. The columns of V are the principal components (which form an orthonormal basis for the space spanned by the observations), and the corresponding singular values provide us information about how much variance is captured in each principal component. More specifically, let σ_i be the i-th non-zero singular value. Then the value

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

is the percentage of the variance captured by the i-th principal component. We compute the truncated SVD of the iris data and show the variance percentages for each component below:

```
>>> U,S,VT = la.svd(Y, full_matrices=False)
>>> S**2/(S**2).sum() # variance percentages
array([ 0.92461621,  0.05301557,  0.01718514,  0.00518309])
```

In general, we are only interested with the first several principal components. But just how many principal components should we keep? There are a number of ways to decide this. One is to only keep the first two principal components, as these enable us to project the data into 2-dimensional space, which is easy to visualize. Another way is to only keep the set of principal components accounting for a certain percentage (say 80%) of the variance. A third method is to examine the *scree plot* of the variance percentages for each principal component, as in Figure 8.3. Upon examination of the iris scree plot, we see that there is a distinct change after the first principal component. This method is referred to as finding the "elbow" of the scree plot, and we keep all the principal components on the left of the elbow. In the case of the iris data, that is simply the first principal component, which accounts for 92% of the variance.

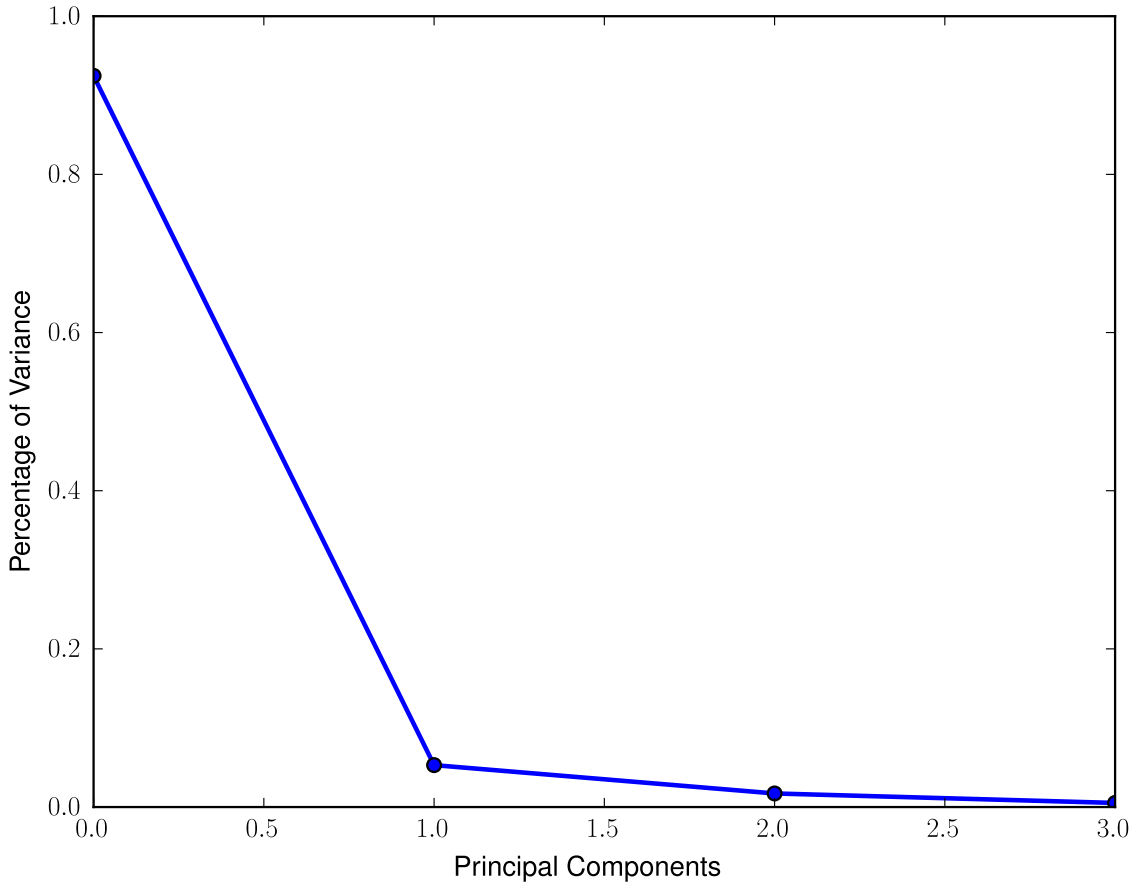


Figure 8.3: Scree plot of the percentage of variance for PCA on the iris dataset.

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

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

where $\Sigma_{:l,:l}$ is the first l rows and columns of Σ and $U_{:,:l}$ is the first l columns of U. Using the SVD formula, note that

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

where $V_{:,:l}$ is the first l columns of V. In this way, we see that the i-th row of $\widehat{Y}$ is simply the projection of the i-th observation onto the orthonormal set of the first l principal components. Under this projection, the data is represented in fewer dimensions, and in such a way that accentuates the variance (which can help with finding patterns within the data).

In Figure 8.4 we display the transformed iris data set, plotting the first principal component against the second. This reduction helps us to see the distinctions between the three different species, using only two dimensions instead of the full four dimensions of the feature space.

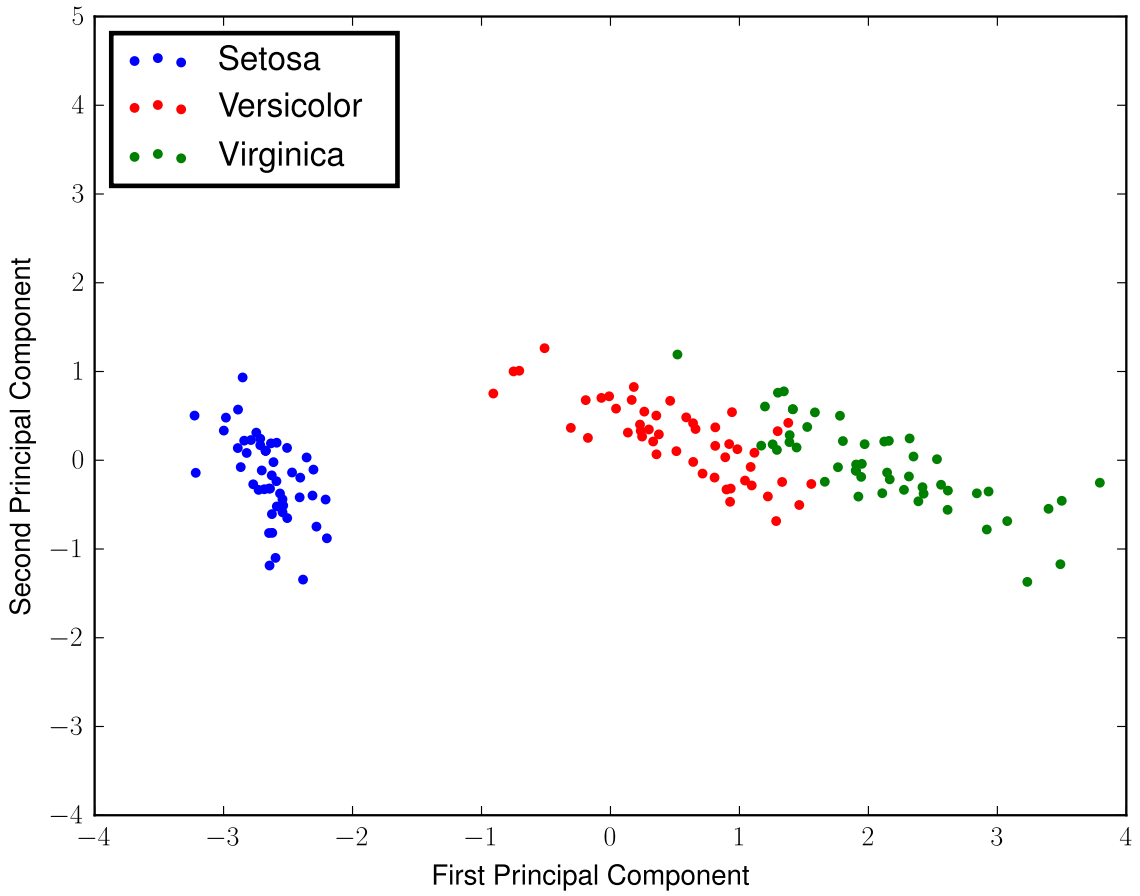


Figure 8.4: Plot of the transformed iris data, keeping only the first two principal components.

Problem 1. Recreate the plot shown in Figure 8.4 by performing PCA on the iris dataset, keeping the first two principal components.

Note: If `Yhat` is your 150×2 array of transformed observations, you can access the rows corresponding to the setosa flowers as follows:

```
>>> Yhat[iris.target==0]
```

To get the rows corresponding to versicolor and virginica specimens, simply replace the 0 with 1 and 2, respectively.

Latent Semantic Indexing

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

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

In order to do so, we need to represent the documents as numerical vectors. A standard way of doing this is to define an ordered set of words occurring in the collection of documents (called the *vocabulary*), and then to represent each document as a vector of word counts from the vocabulary. More formally, let our vocabulary be $V = \{w_1, w_2, \ldots, w_m\}$. 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, each row being a document vector. As expected, we let $X_{i,j}$ be the number of times term j occurs in document i. Note that X is often a sparse matrix, as any one document likely doesn't contain most of the vocabulary words. This mode of representation is called the *bag of words* model for documents.

We calculate the SVD of X without centering or scaling the data so that we may retain the sparsity. We now have $X = U\Sigma V^T$. Once we have selected the number of principal components to keep, say l, we can represent the corpus of documents by the matrix

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

Note that $\widehat{X}$ will no longer be a sparse matrix, but it has dimensions $n \times l$, which is much smaller than $n \times m$ when $l \ll m$.

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

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

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

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

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

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

```
>>> # get list of filepaths to each text file in the folder
>>> import string
>>> from os import listdir
>>> path_to_addresses = "./Addresses/"
>>> paths = [path_to_addresses + p for p in os.listdir(path_to_addresses) if p↩
    [-4:]==".txt"]

>>> # helper function to get list of words in a string
>>> def extractWords(text):
>>>     trans = string.maketrans("", "")
>>>     return text.strip().translate(trans, string.punctuation+string.digits).↩
    lower().split()

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

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

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

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

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

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

```
>>> word_index = [] # holds the column index of X

>>> # iterate through the documents
>>> for doc, p in enumerate(paths):
>>>     with open(p, 'r') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             ctr.update(extractWords(line))
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_index.append(vocab[word])
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>             except KeyError: # if word isn't in vocab, skip it
>>>                 pass

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

Problem 2. Using the techniques of LSI discussed above–applied to the word count matrix X, and keeping the first 7 principal components–find the most similar and least similar speeches to both Bill Clinton's 1993 speech and to Richard Nixon's 1974 speech. Are the results plausible?

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

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

Let t_j be the total number of times term j appears in the entire collection of documents. Define

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

We then let

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

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

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

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

To calculate the matrix A in a streaming manner, we must alter our code above somewhat:

```
>>> from itertools import izip
>>> from math import log
>>> t = np.zeros(len(vocab))
>>> counts = []
>>> doc_index = []
>>> word_index = []

>>> # get doc-term counts and global term counts
>>> for doc, path in enumerate(paths):
>>>     with open(path, 'r') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             words = extractWords(line)
>>>             ctr.update(words)
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_ind = vocab[word]
>>>                 word_index.append(word_ind)
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>                 t[word_ind] += count
>>>             except KeyError:
>>>                 pass

>>> # get global weights
>>> g = np.ones(len(vocab))
>>> logM = log(len(paths))
>>> for count, word in izip(counts, word_index):
>>>     p = count/float(t[word])
>>>     g[word] += p*log(p+1)/logM

>>> # get globally weighted counts
>>> gwcounts = []
>>> for count, word in izip(counts, word_index):
>>>     gwcounts.append(g[word]*log(count+1))

>>> # create sparse matrix holding these globally weighted word counts
>>> A = sparse.csr_matrix((gwcounts, [doc_index,word_index]), shape=(len(paths)↩
    ,len(vocab)), dtype=np.float)
```

Problem 3. Repeat Problem 2 using the matrix A. Do your answers seem more reasonable than before?

9 Naive Bayes

Lab Objective: *Implement Naive Bayes Classification Models.*

Introduction

Naive Bayes classification methods are a good introduction to machine learning techniques. They are relatively straightforward to understand and to implement, while they are also very effective for certain applications. However, they are somewhat limited by their strong dependence on assumptions of independence.

Recall that "the classification problem" tries to assign the correct label to a given set of features (called a *feature vector*). For example, suppose we wish to give the correct labels (names) to two different pieces of fruit. The given features of the first fruit are that it is red and round, and the features of the second are that the fruit is long and yellow. If we assign to these fruits the names "apple" and "banana" respectively, then these are our labels.

It is common in classification problems to start with a set of correctly-labeled feature vectors called a *training set*. We use the training set to train our algorithm to make predictions. It is also common to have another smaller set of correctly-labeled feature vectors called a *test set*. To verify the effectiveness of our algorithm, we predict the labels of the test set, and then compare the predicted labels to the true labels.

Recall that Bayes rule for random variables gives that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{\int P(X|Y)P(Y)dy}$$

where $P(Y)$ is our prior distribution and $P(X|Y)$ is our likelihood function.

Suppose that we have a set of features that we wish to label. Let $x = (x_1, \dots, x_n)$ be this feature vector and let $C = \{c_1, \dots, c_k\}$ be our set of possible labels for x. We may apply Bayes rule to this problem as follows:

$$P(c_i|x) = P(c_i|x_1, \dots, x_n) = \frac{P(x_1, \dots, x_n|c_i)P(c_i)}{P(x_1, \dots, x_n)}$$

If we make no further assumptions, this problem is intractable. To effectively estimate even the simplest case where each feature is a boolean value would require us to estimate around $k2^n$ parameters. The problem gets exponentially worse if we were to consider non-boolean features.

However, if we can make the assumption that the features are conditionally-independent of one another, the problem can be simplified dramatically. If we make this assumption and apply Bayes rule, we have that

$$P(c_i|x_1,\dots,x_n) = \frac{P(x_1|c_i)P(x_2|c_i)\dots P(x_n|c_i)P(c_i)}{P(x_1,\dots,x_n)}.$$

In this case, we only need to estimate kn parameters. The Naive Bayes classification algorithm chooses the label with the highest probability. Since this is independent of the denominator in Bayes rule, we can simplify the problem further. Given an unlabeled feature vector $x = (x_1,\dots,x_n)$, we assign the label

$$c = \underset{i\in\{1,\dots,k\}}{\operatorname{argmax}} P(c_i)\prod_{j=1}^{n} P(x_j|c_i).$$

To assign a label to a set of features, we calculate each $P(x_j|c_i)$ and choose a prior $P(c_i)$. We then choose the argmax as above. The calculation of conditional probabilities and our choice of a prior will depend on the type of problem that we are solving.

Gaussian Classifiers

Gaussian classifiers are commonly used when dealing with continuous data. We assume that each feature is normally distributed and conditionally-independent of all other features. We may then calculate $\mu_{j,i}$ and $\sigma^2_{j,i}$ (the mean and variance) corresponding to each feature of each class. Then $P(x_j|c_i)$ can be calculated using the gaussian pdf

$$P(x_j|c_i) = \frac{1}{\sqrt{2\pi\sigma^2_{j,i}}}\exp -\frac{(x_j-\mu_{j,i})^2}{2\sigma^2_{j,i}}.$$

For example, suppose we have a training set with labels indicating the sex of a person (1 for female, 2 for male), and features consisting of hair length and height (features 1 and 2, respectively). Further suppose that the mean hair length of the women in the training set is 15 centimeters with standard deviation 1.5 cm, and the mean height is 1.25 meters with standard deviation 6 cm. Similarly, suppose that the mean hair length of the men in the training set is 5 centimeters with standard deviation 2.5 cm, and the mean height is 1.75 meters with standard deviation 7 cm. Under this setup, we have

$$\begin{aligned} \mu_{1,1} &= 15, & \sigma_{1,1} &= 1.5,\\ \mu_{2,1} &= 1.25, & \sigma_{2,1} &= .06,\\ \mu_{1,2} &= 5, & \sigma_{1,2} &= 2.5,\\ \mu_{2,2} &= 1.75, & \sigma_{2,2} &= .07. \end{aligned}$$

If we wish to classify a person with hair length 17 centimeters who is 1.4 meters tall, we calculate the probability of each label using the parameters given above and a uniform prior ($P(F) = P(M) = \frac{1}{2}$) as follows:

$$\begin{aligned} P(F \mid 17, 1.4) &= P(F) \left(\frac{1}{\sqrt{2\pi\sigma_{1,1}^2}} \exp - \frac{(17-\mu_{1,1})^2}{2\sigma_{1,1}^2} \right) \left(\frac{1}{\sqrt{2\pi\sigma_{2,1}^2}} \exp - \frac{(1.4-\mu_{2,1})^2}{2\sigma_{2,1}^2} \right) \\ &= .016 \\ P(M \mid 17, 1.4) &= P(M) \left(\frac{1}{\sqrt{2\pi\sigma_{1,2}^2}} \exp - \frac{(17-\mu_{1,2})^2}{2\sigma_{1,2}^2} \right) \left(\frac{1}{\sqrt{2\pi\sigma_{2,2}^2}} \exp - \frac{(1.4-\mu_{2,2})^2}{2\sigma_{2,2}^2} \right) \\ &= 1.7 \times 10^{-11} \end{aligned}$$

The Female label has a greater probability given the feature vector, and so we classify the person as Female.

A nice way to visualize how a classifier works is to plot the decision boundaries for two-dimensional subspaces of the feature vector space. For example, the decision boundaries for a Gaussian Naive Bayes classifier trained on a dataset consisting of the sepal widths and sepal lengths of three different types of flowers are shown in Figure 9.1.

Working in Log Space

In the example presented above, notice that the value of $P(M \mid 17, 1.4)$ is very small. This is often the case in classification problems; certain classes may be very unlikely, and so calculating these probabilities may lead to numerical underflow. This is especially pronounced in the Naive Bayes model, which involves taking the product of several numbers between 0 and 1. A useful technique to avoid underflow is to perform all of the computations in logarithmic space, where the products all become sums. When we do so, the Naive Bayes label assignment is

$$c = \underset{i \in \{1,\ldots,k\}}{\operatorname{argmax}} \log P(c_i) + \sum_{j=1}^{n} \log P(x_j | c_i).$$

Since the logarithm is a monotone increasing function, the argmax is the same whether in log space or in the original formulation.

Problem 1. Download the `seeds_dataset.txt` file. This file contains 7 features describing 3 species of wheat.

1. Area
2. Perimeter
3. Compactness
4. Length
5. Width
6. Asymmetry Coefficient

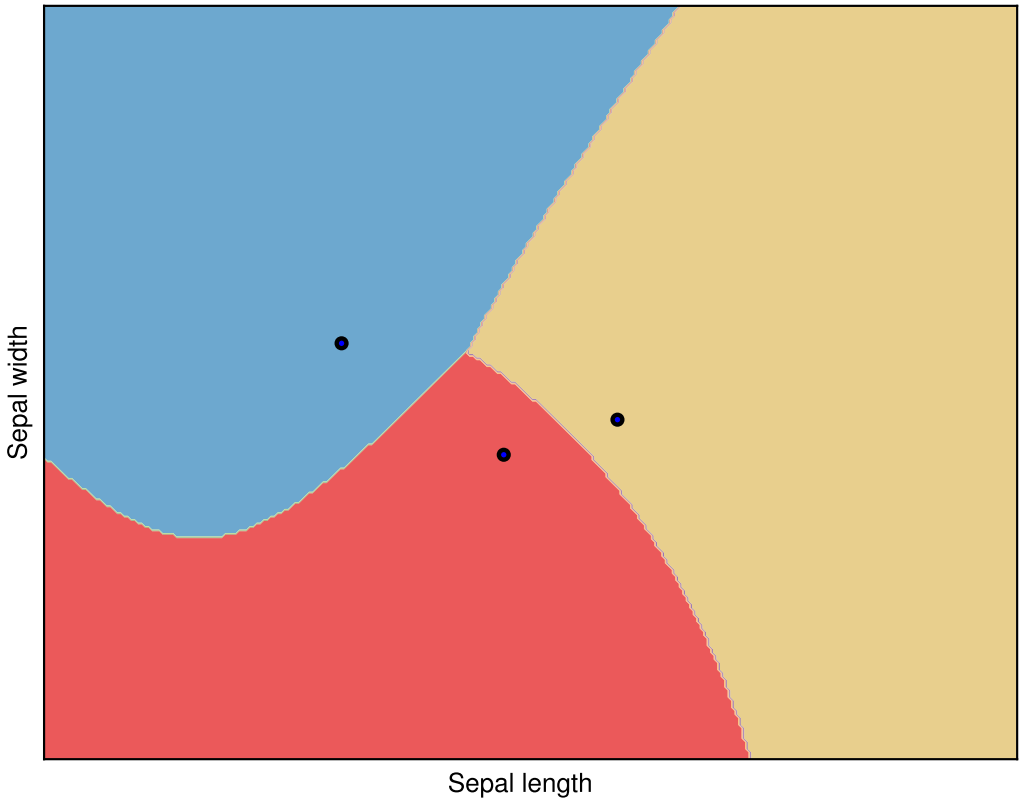


Figure 9.1: Decision boundaries for a Gaussian Naive Bayes classifier on the iris flower dataset, together with the means for each flower. Each point in the plane represents a 2-dimensional feature vector, and the color associated with each point indicates which class label was assigned to that feature vector.

7. Groove length

The species of wheat are

1. Kama
2. Rosa
3. Canadian

The measurements of the kernels are real valued, making this a good example on which to try our Gaussian classifier. Make a function that classifies a subset of the data in this file and returns the accuracy of your calculations by proceeding in the following manner:

1. Randomly select a test set consisting of 40 vectors. Make the remaining vectors into your training set.
2. Calculate the mean and variance for each feature of each label using the training set.

3. Using a uniform prior, predict the labels of your test set.
4. Compare your predictions to the correct labels of the test set. In particular, report the accuracy of the prediction, which is the number of correctly predicted test samples divided by the total number of test samples.

We may also use SciPy's `sklearn` library to implement a Gaussian classifier. After importing the library, we can create a new classifier and train it with just a few lines of code. To create a Gaussian classifier, use the following code.

```
from sklearn.naive_bayes import GaussianNB
nb_classifier = GaussianNB()
```

Given a training set, we can also quickly train the classifier to a certain problem. This requires the training set and labels as two arguments.

```
nb_classifier.fit(training_set, labels)
```

Once the classifier has been trained, we can predict the labels for a test set.

```
pred_labels = nb_classifier.predict(test_set)
```

The `predict` method returns an array of labels for the test set.

Problem 2. Repeat the previous problem using `sklearn`'s Naive Bayes classifier. Check that your implementation from the previous problem predicts the same labels as the `sklearn` implementation does.

Document Classification and Spam Filters

Naive Bayes classifiers are often used in document classification, a major example being spam detection. When it comes to document classification, a common choice for the feature vector is simply a count for the number of times each word in the specified vocabulary occurs in the document. For example, suppose we are trying to classify a document, and the vocabulary of relevant words is the ordered set

$$(\text{bank, tree, wealth, money, river, water}).$$

Suppose that the document is the sentence

> "The woman deposited her money in the bank, and then made her way down to the bank of the river, contemplating her wealth."

Then the feature vector for this document is

$$(2, 0, 1, 1, 1, 0).$$

Notice in particular that the i-th entry of the feature vector indicates the number of occurrences of the i-th vocabulary word in the document. Such a feature vector is often called a *word-count vector*. Notice that the count vector ignores words in the document that are not part of the vocabulary, and it also disregards the order of the words. This simple representation of text documents is known as the *bag-of-words model* or the *vector space model*. The hypothesis that drives spam filters is that spam messages will use words with different frequencies. For example, a spam message will often have a sales pitch, so the word "buy" and "cheap" will appear often. On the other hand, legitimate messages will probably use different language. Thus, it is reasonable to use word-count vectors as our feature vectors when attempting to distinguish between spam and legitimate email.

We now introduce formalisms to derive the Naive Bayes model for document classification. Let $V = (v_1, v_2, \ldots, v_n)$ be an ordered list of words, called the vocabulary, and let $x = (x_1, x_2, \ldots, x_n)$ be a word-count vector. Let $\{c_1, c_2, \ldots, c_k\}$ be the set of classification labels. In the case of continuous data and Gaussian Classifiers, each class label was associated with corresponding mean and variance parameters, and these determined the likelihood of the feature vector given the class label. In the case of document classification, each class label c_i has a corresponding probability vector $(p_{i,1}, p_{i,2}, \ldots, p_{i,n})$ whose entries are nonnegative and sum to 1. This probability vector defines a categorical probability distribution over the vocabulary V, where $p_{i,j}$ represents the probability of seeing word v_j given class label c_i. With this notation in place, our Naive Bayes model takes the form

$$c = \underset{i \in \{1,\ldots,k\}}{\operatorname{argmax}} P(c_i) \prod_{j=1}^{n} p_{i,j}^{x_j}.$$

Given a training set of labeled documents, we can calculate the prior probabilities $P(c_i)$ and word probabilities $p_{i,j}$ as follows. Each prior probability $P(c_i)$ is simply the proportion of training documents that have the label c_i. Next, let $count(c_i, v_j)$ denote the number of occurrences of word v_j among all training documents that have label c_i. Then we have

$$p_{i,j} = \frac{count(c_i, v_j) + 1}{\sum_{j=1}^{n} (count(c_i, v_j) + 1)}.$$

(Note that adding 1 to the number of occurrences of each word is known as *add-one smoothing*, and is a common technique to prevent over-fitting.)

Once we have calculated these parameters (the "fitting" stage), we are ready to classify new documents (the "prediction" stage) using the argmax equation given above. Remember to perform calculations in log space to prevent numerical underflow.

Problem 3. Implement a Naive Bayes model for document classification. We provide an interface below.

```
class naiveBayes(object):
    """
    This class performs Naive Bayes classification for word-count document features.
    """
    def __init__(self):
        """
        Initialize a Naive Bayes classifier.
        """
        pass
```

```
    def fit(self,X,Y):
        """
        Fit the parameters according to the labeled training data (X,Y).

        Parameters
        ----------
        X : ndarray of shape (n_samples, n_features)
            Each row is the word-count vector for one of the documents
        Y : ndarray of shape (n_samples,)
            Gives the class label for each instance of training data. ↩
                Assume class labels are in {0,1,...,k-1} where k is the ↩
                number of classes.
        """
        # get prior class probabilities P(c_i)
        # (you may wish to store these as a length k vector as a class ↩
            attribute)

        # get (smoothed) word-class probabilities
        # (you may wish to store these in a (k, n_features) matrix as a ↩
            class attribute)

        pass

    def predict(self, X):
        """
        Predict the class labels of a set of test data.

        Parameters
        ----------
        X : ndarray of shape (n_samples, n_features)
            The test data

        Returns
        -------
        Y : ndarray of shape (n_samples,)
            Gives the classification of each row in X
        """
        pass
```

Problem 4. In this problem, you will train a Naive Bayes classifier using a corpus of emails extracted from the Enron dataset.

Load in the data from `SpamFeatures.txt`. This is a text file containing a whitespace-delimited numerical array with several thousand columns and several thousand rows, each row representing an email as a count vector. Also load in the data from `SpamLabels.txt`, which is a text file containing a 1 (for legitimate email) or 0 (for spam email) on each line, in correspondence with the rows of the count vector array. Using your document classification implementation, do the following:

1. Randomly create a test set from the data (500 documents), leaving the remaining documents as the training set.
2. Create a Naive Bayes classifier and fit it using the training set.
3. Predict the labels of the test set and compare them to the true labels (by reporting the classification accuracy).

Next, perform the same task using `sklearn`'s implementation and the same training and testing sets:

```
>>> # assume train_vectors, train_labels, and test_vectors are defined
>>> from sklearn.naive_bayes import MultinomialNB
>>> mnb = MultinomialNB()
>>> mnb.fit(train_vectors, train_labels)
>>> predicted = mnb.predict(test_vectors)
```

Again report the accuracy of the predicted labels. The result should be on par with those produced by your own implementation.

10 K-Nearest Neighbors and Support Vector Machines

Lab Objective: *Implement the k-Nearest Neighbor (KNN) and binary Support Vector Machine (SVM) classifiers.*

For numerical data, one of the most simple classification methods is the k-nearest neighbor (KNN) classifier, which labels a new sample according to the majority vote of the nearest k training samples. As k is the only parameter for the model, this choice determines the effectiveness of the classifier. Throughout this lab we will explore how different values of k affect the accuracy of our classifier.

Suppose we have numerical data $\mathbf{x}_1, \cdots, \mathbf{x}_N$ and associated labels $y_1, \cdots, y_N$, along with a metric d on our feature space. We define the k-neighborhood of a new sample $\mathbf{x}$ to be

$$n(\mathbf{x}, k) = \{\mathbf{x}_i : d(\mathbf{x}_i, \mathbf{x}) < d(\mathbf{x}_j, \mathbf{x}) \text{ for all but fewer than } k \text{ samples } \mathbf{x}_j\}$$

Thus the k-neighborhood of a new sample is the set of samples from our training set which are the k closest samples according to our metric.

Problem 1. Write a function that computes the k-neighborhood of a sample $\mathbf{x}$ given k and a training set. Assume the use of the Euclidean metric.

We define the k-neighborhood votes of a new sample $\mathbf{x}$ to be

$$v(\mathbf{x}, k) = \{y_i : \mathbf{x}_i \in n(\mathbf{x}, k)\}$$

The label assigned to $\mathbf{x}$ according to the standard KNN classifier is the mode of the k-neighborhood votes.

Problem 2. Write a function that labels a new sample $\mathbf{x}$ given k and a training set. Assume the use of the Euclidean metric.

Problem 3. Write a KNN class which accepts initial training data and training labels. It should have a method to classify new samples, given a value of k. Load the iris dataset from `sklearn.datasets`, and by separating the data into training and testing sets, implement your class. Test your classifier on the test data given different values of k. What are the misclassification rates?

Different values of k lead to different results. Essentially, our choice of k determines how far-reaching we would like the influence of a sample to be. Larger k means that a sample is influenced by points farther away from it. Smaller k means that a sample is influenced only by the few points nearest it. In either case, extreme choices of k (too small or too big) often yield poor results.

Another powerful classifier is the support vector machine (SVM). There are two main ideas in this classifier: maximum-margin hyperplanes and kernel functions. The first is simply the thought that the simplest binary classifier is a separating hyperplane, the best being the hyperplane that is "farthest" from the nearest two points of opposing classes, while perfectly partitioning the training data. There are very few interesting classification problems where this is possible in the standard feature space. However, if we can transform the feature space into a higher-dimensional space, then we might be able to find such a hyperplane. Unfortunately, working in this higher-dimensional space can be quite costly, which is where the kernel functions come into play.

The second big idea is that instead of working directly in the higher-dimensional space, we can choose our transformation in such a way that we can use *kernel* functions for any necessary computations whose domain is the product space of the original feature space with itself. There are many, sometimes exotic, kernel functions to choose from, though in practice only a few forms are used. We let $\phi(\mathbf{x})$ be the transformation of $\mathbf{x}$ into the higher-dimensional space, and we let this transformation be determined by some kernel function

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j).$$

We assume now that our labels are simply ± 1, and we assume $\phi(\mathbf{x}) \in \mathbb{R}^K$. We consider the hyperplane defined by $f(\phi(\mathbf{x})) = \mathbf{w}^T\phi(\mathbf{x}) + b$, where $\mathbf{w} \in \mathbb{R}^K$ and $b \in \mathbb{R}$. We wish to find $\mathbf{w}$ and b such that $f(\phi(\mathbf{x}_i)) > 0$ if $y_i = 1$ and $f(\phi(\mathbf{x}_i)) < 0$ if $y_i = -1$. Thus we need $\mathbf{w}$ and b to satisfy $y_i\left(\mathbf{w}^T\phi(\mathbf{x}) + b\right) > 0$ for all $i = 1, \cdots, N$. Additionally, we would like the distance between the boundary $\mathbf{w}^T\phi(\mathbf{x}) + b = 0$ and the nearest points to be maximized. We can determine this distance geometrically to be

$$\frac{2}{\|\mathbf{w}\|},$$

and is called the margin. Thus we would like to solve the following optimization problem:

$$\begin{aligned} &\text{minimize } \|\mathbf{w}\| \\ &\text{subject to } y_i\left(\mathbf{w}^T\phi(\mathbf{x}) + b\right) > 0 \quad \text{for } i = 1, \cdots, N. \end{aligned}$$

Considering the Lagrangian of this optimization problem, we have the dual formulation of this optimization problem as

$$\begin{aligned} &\text{maximize } \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j y_i y_j \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j) \\ &\text{subject to } a_i \geq 0 \quad i = 1, \cdots, N \\ &\qquad\qquad \sum_{i=1}^{N} a_i y_i = 0. \end{aligned}$$

This is simply a quadratic programming problem, where the objective function is $\mathbf{a}^T\mathbf{1} - \frac{1}{2}\mathbf{a}^T Q\mathbf{a}$, where

$$Q_{ij} = y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)^T = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j).$$

Quadratic programming problems have nice solutions, so this will be easy to solve. The classifier function $f(\mathbf{x}) = \sum_{i=1}^{N} a_i y_i k(\mathbf{x}, \mathbf{x}_i)$ also has a nice closed-form solution.

Given a training set of size `n_samples` and a kernel k, with data X and target Y, we can use `cvxopt` to solve this quadratic programming problem:

```
>>> import cvxopt
>>> import numpy as np
>>> K = np.zeros((n_samples,n_samples))
>>> for i in xrange(n_samples):
>>>     for j in xrange(n_samples):
>>>         K[i,j] = k(X[i,:], X[j,:])
>>> Q = cvxopt.matrix(np.outer(Y, Y) * K)
>>> q = cvxopt.matrix(np.ones(n_samples) * -1)
>>> A = cvxopt.matrix(Y, (1, n_samples))
>>> b = cvxopt.matrix(0.0)
>>> G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
>>> h = cvxopt.matrix(np.zeros(n_samples))
>>> solution = cvxopt.solvers.qp(Q, q, G, h, A, b)
>>> a = np.ravel(solutions['x'])
```

From this value a, our kernel k, a training set X, and target Y, we have everything we need to build an SVM classifier. But what should our kernel k be? There are three common kernels used:

$$\begin{aligned} \text{Polynomial: } & k(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + a)^d \\ \text{Radial Basis Function: } & k(\mathbf{x}, \mathbf{y}) = e^{-\gamma\|\mathbf{x}-\mathbf{y}\|^2} \\ \text{Sigmoid: } & k(\mathbf{x}, \mathbf{y}) = \tanh\left(\mathbf{x}^T\mathbf{y} + r\right) \end{aligned}$$

Problem 4. Write an SVM class. Upon initialization, it should accept a training data set and training target set. It should have a method called `setKernel` which accepts one of the three kernel types, and defines a kernel function for the object. It should also have a method to train the classifier, and another method to predict the class of a new sample. It should predict 1 if $f(\mathbf{x}) > 0$ and -1 if $f(\mathbf{x}) < 0$.

A data set on breast cancer has been provided to you. This is from a breast cancer database from the University of Wisconsin Hospital, Madison, from Dr. W. H. Wolberg. The attributes are (in order): clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitoses, all on a scale from 1 to 10. The targets are either 1 or -1, signifying malignant or benign, respectively.

Problem 5. Load the data set. Separate it into a training set and a test set. Train SVMs on the data, trying each kernel with various parameter values. What are your misclassification rates?

11 Image Recognition Tasks

Lab Objective: *Use the KNN and SVM algorithms to solve two image recognition problems.*

Two important image recognition problems are character recognition and face recognition. The first is generally framed as a post office problem. Every day, millions of pieces of mail are sent through the US Postal Service each day. This requires an automated way of routing much of the mail. The problem is to automatically determine the zip code of the addressee for a piece of mail. There are two parts to this problem: find the zip code on the letter, and then determine what it is. We will only consider the second part in this lab.

Given that we have an image of a single digit, how can we decide what it is without human intervention? This is a classification problem, with the classes being the digits 0 through 9. We will use both the KNN and SVM classifiers to predict each digit.

We will use the `digits` data set from `sklearn.datasets` for our data, using the method `sklearn.datasets.load_digits()`. Each sample is an 8×8 image which has been flattened into a length-64 vector.

Problem 1. Load the digits data and separate it into a training set and a test set.

The module `sklearn.neighbors` has a nice class `KNeighborsClassifier` that implements the KNN classifier.

Problem 2. Find and read some of the documentation for the aforementioned class. Implement a KNN classifier on the training set. What is the misclassification rate on your test set?

While the SVM was originally designed as a binary classifier, it has been extended into a multi-class classifier as well. We won't go into the details here, but one common extension is to train K different SVMs (where K is the number of classes), each being a "one-versus-all" classifier. After some calibration, a new sample is predicted to be the class k where $f_k(\mathbf{x})$ is greatest, f_k being the function defining the hyperplane for class k against all other classes. Again, `sklearn.svm` has a nice class `SVC` that implements this.

The module `sklearn.grid_search` provides a nice way to find the classifier that performs the best on the training set, considering a grid of parameters. We will use this to help us find an optimal SVM for the digits data set, where we use a radial basis function for the kernel.

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5], 'gamma': [0.0001, 0.0005, ↩
    0.001, 0.005, 0.01, 0.1],}
>>> clf = GridSearchCV(SVC(kernel='rbf',class_weight='auto'), param_grid)
>>> clf = clf.fit(training_data, training_target)
```

Problem 3. Predict the values for the test set using the SVM we just trained. What is your misclassification rate? How does the SVM's performance compare to the KNN classifier?

We now consider our second image recognition problem: face recognition. This is much harder than simply recognizing a digit on a white envelope, as there is bound to be so much more noise and variation.

We use as our data set the "Labeled Faces in the Wild", a database of face photographs. In fact, we will only consider a small subset of this database, including only images of Ariel Sharon, Colin Powell, Donald Rumsfeld, George W Bush, Gerhard Schroeder, Hugo Chavez, and Tony Blair. Through `sklearn.datasets` we can download and process this data set rather easily, though it might take some time.

```
>>> from sklearn.datasets import fetch_lfw_people
>>> people = fetch_lfw_people(min_faces=70,resize=0.4)
>>> data = people.data
>>> target = lfw_people.target
```

This data set consists of 1288 images of size 50×37, each flattened. The targets are digits from 0 to 6, corresponding with the ordered names above. This might seem counterintuitive, considering the main ideas of SVMs, but it is sometimes useful to reduce the dimensionality of our feature space before implementing an SVM, using PCA so we can retain as much information as possible while still reducing the dimensionality.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=150, whiten=True).fit(data)
>>> data_pca = pca.transform(data)
```

Problem 4. Separate the data set into training data and test data. Create a PCA object fit to the training data. With this object, reduce the dimensionality of both the training data and the test data.

Problem 5. Train an SVM on the image set with the same parameter grid search used above. Label the test data. What is your misclassification rate?

Let's also compare and see how well the KNN classifier does on this data set.

Problem 6. Train a KNN classifier on the image data set. What is your misclassification rate on the test data? Does this surprise you?

This allows us to end this course with a very important take-home message: the No Free Lunch Theorem. In essence, this theorem states that there is no single machine learning classifier to rule them all–each has its strengths and weaknesses. More specifically, if there is a ML classifier that outperforms all other classifiers on a data set, then we can find a data set where a different classifier will be superior. This means that we have to be intelligent in how we choose which classifiers to try, because there isn't any "go-to" classifier that will always work.

12 K-Means Clustering

Lab Objective: *Understand the basics of* k-means *clustering, and apply to the problem of clustering earthquake epicenters.*

Clustering

In Lab 8, we analyzed the iris dataset using PCA; we have reproduced the first two principal components of the iris data in Figure 12.1. Upon inspection, a human can easily see that there are two very distinct groups of irises. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*.

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

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

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

and

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

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

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

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

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

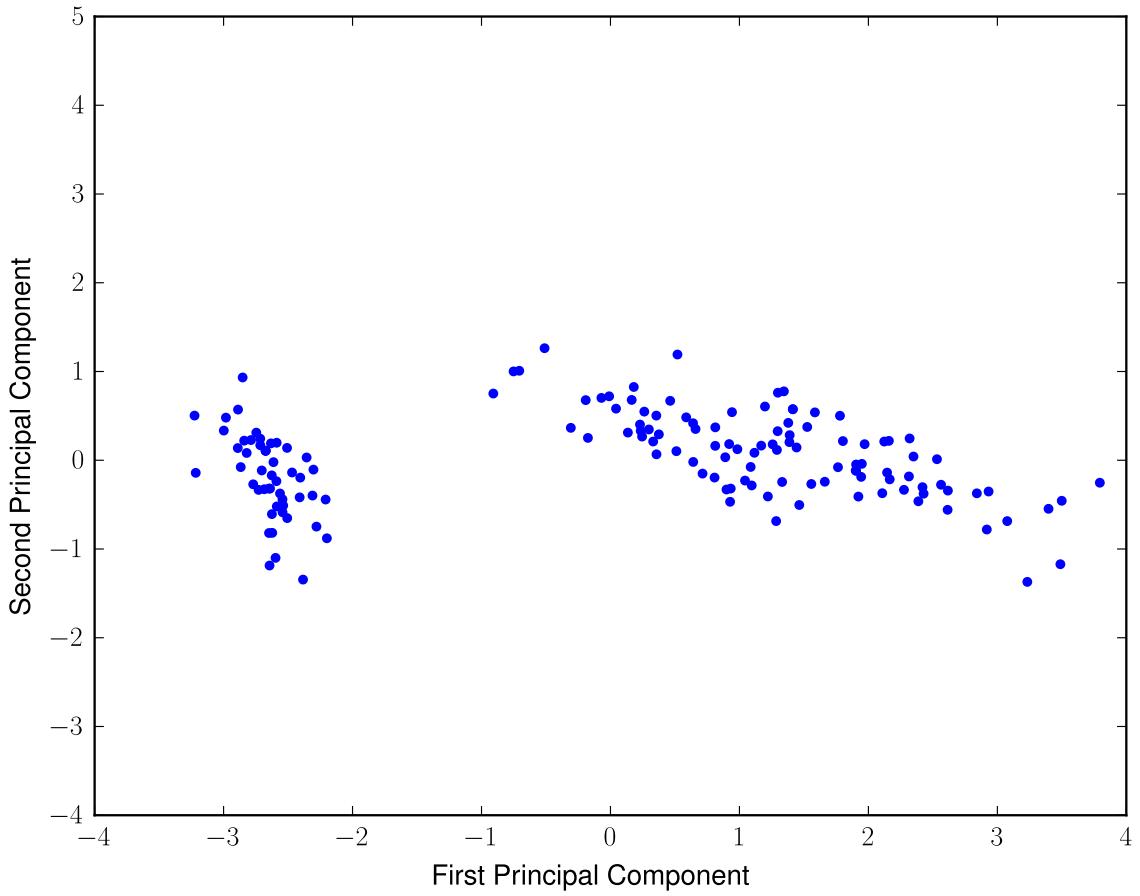


Figure 12.1: The first two principal components of the iris dataset.

The K-Means Method

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

We begin by specifying an initial cluster mean $\mu_i^{(1)}$ for each $i = 1, \cdots, N$ (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.

Examine Figure 12.2, which shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

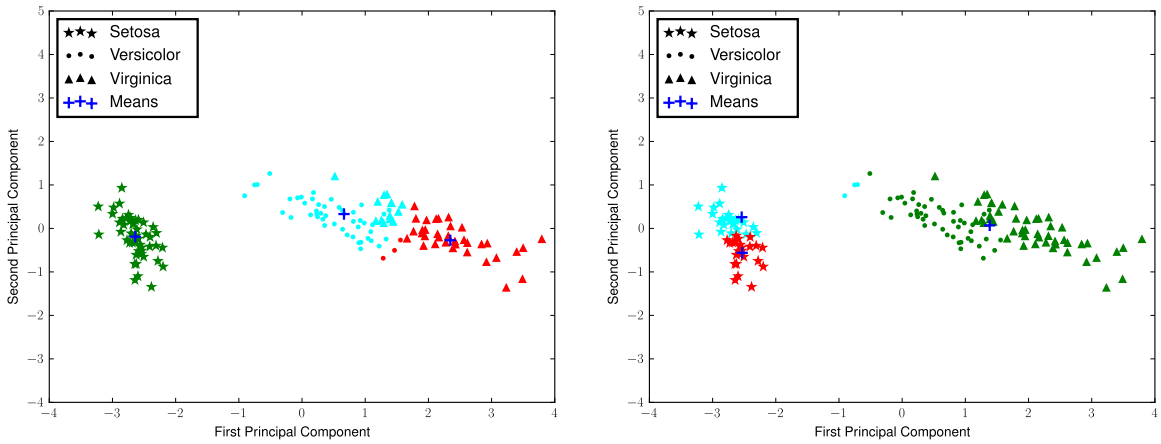


Figure 12.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

Problem 1. Implement the *k-means* algorithm using the following function declaration.

```
def kmeans(data,n_clusters,init='random',max_iter=300):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    ----------
    data : ndarray of shape (n,k)
        Each row is an observation.
    n_clusters : int
        The number of clusters.
    init : string or ndarray of shape (n_clusters,k)
        If init is the string 'random', then randomly initialize the ↩
            cluster means.
        Else, the initial cluster means are given by the rows of init.
    max_iter : int
        The maximum allowable number of iterations.

    Returns
```

```
    -------
    means : ndarray of shape (n_cluster,k)
        The final cluster means, given as the rows.
    labels : ndarray of shape (n,)
        The i-th entry is an integer in [0,n_clusters-1] indicating
        which cluster the i-th row of data belongs to relative to
        the rows of means.
    measure : float
        The within-cluster sum of squares quality measure.
    """
    pass
```

Test your function on the first two principal components of the iris dataset. Run it 10 times, using a different random initialization of the means each time. Retain the clustering with the smallest within-cluster sum of squares. Your clustering should be similar to the first clustering in Figure 12.2.

Detecting Active Earthquake Regions

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

Our data is contained in 6 text files, each with earthquake data throughout the world covering a time period of one month, giving us data from January 2010 through June 2010. These files contain a lot of information which isn't of interest to us at the present time; all we would like to extract from them is the location of each earthquake, which appears in characters 21 through 33 of each line. Characters 21 through 26 contain the latitude of each epicenter, character 26 denoting North or South, and characters 27 through 33 contain the longitude of each epicenter, character 33 denoting East or West. We need to divide each value by $1,000$ to represent these as degrees and decimals.

Problem 2. Load the earthquake data into a $n \times 2$ array, where each row gives the longitude and latitude of an earthquake in degrees. Multiply South latitudes and West longitudes by -1. Create a scatter plot of the resulting data. You should be able to see the outlines of some of the continents and tectonic plates (since these are often areas of significant seismic activity). Your plot should match Figure 12.3.

We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in $\mathbb{R}^2$ with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. We must recognize that latitude and longitude are best viewed as a variation of spherical coordinates in $\mathbb{R}^3$, and we should interpret them as such. Since our *k-means* algorithm is based on Euclidean distance, we need to transform our data into 3-dimensional Euclidean coordinates.

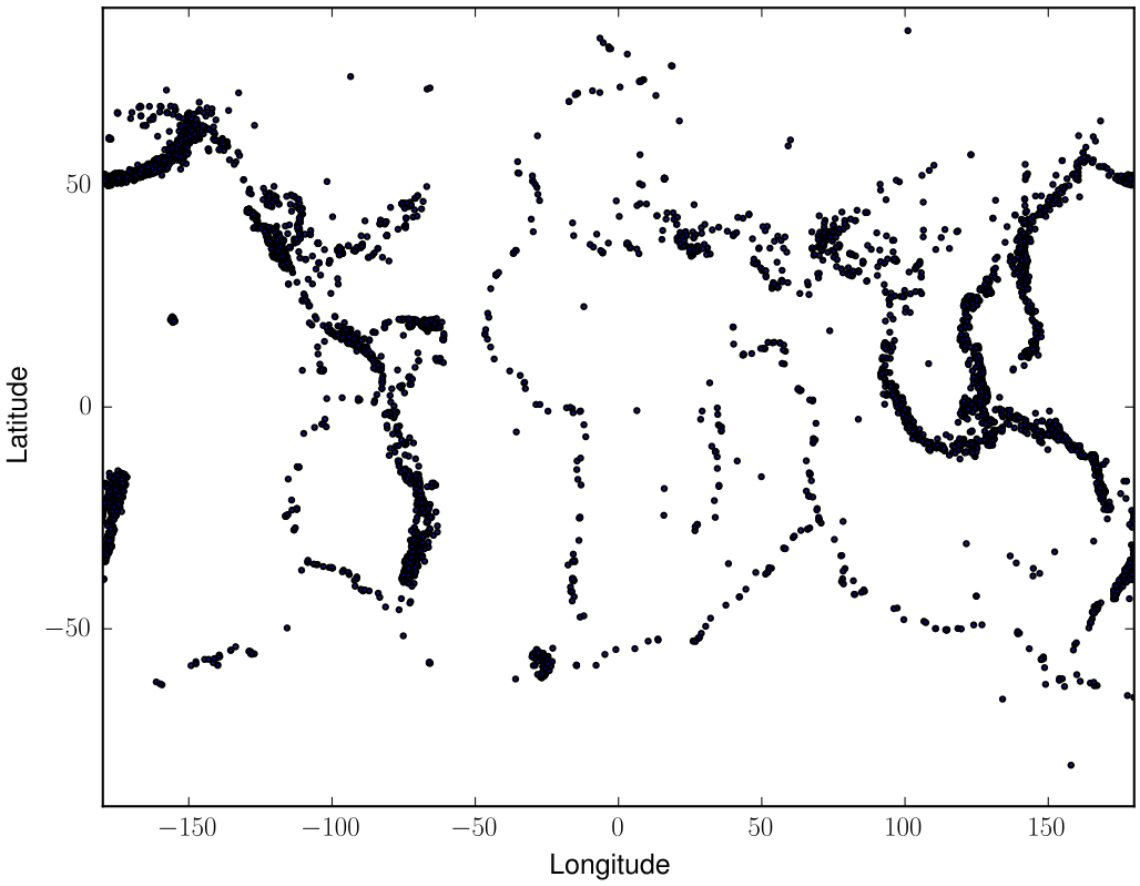


Figure 12.3: Earthquake epicenters over a 6 month period.

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

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

Problem 3. Transform your earthquake data into three dimensional Euclidean coordinates. Be sure to consider if and when you need to transform your data from degrees to radians.

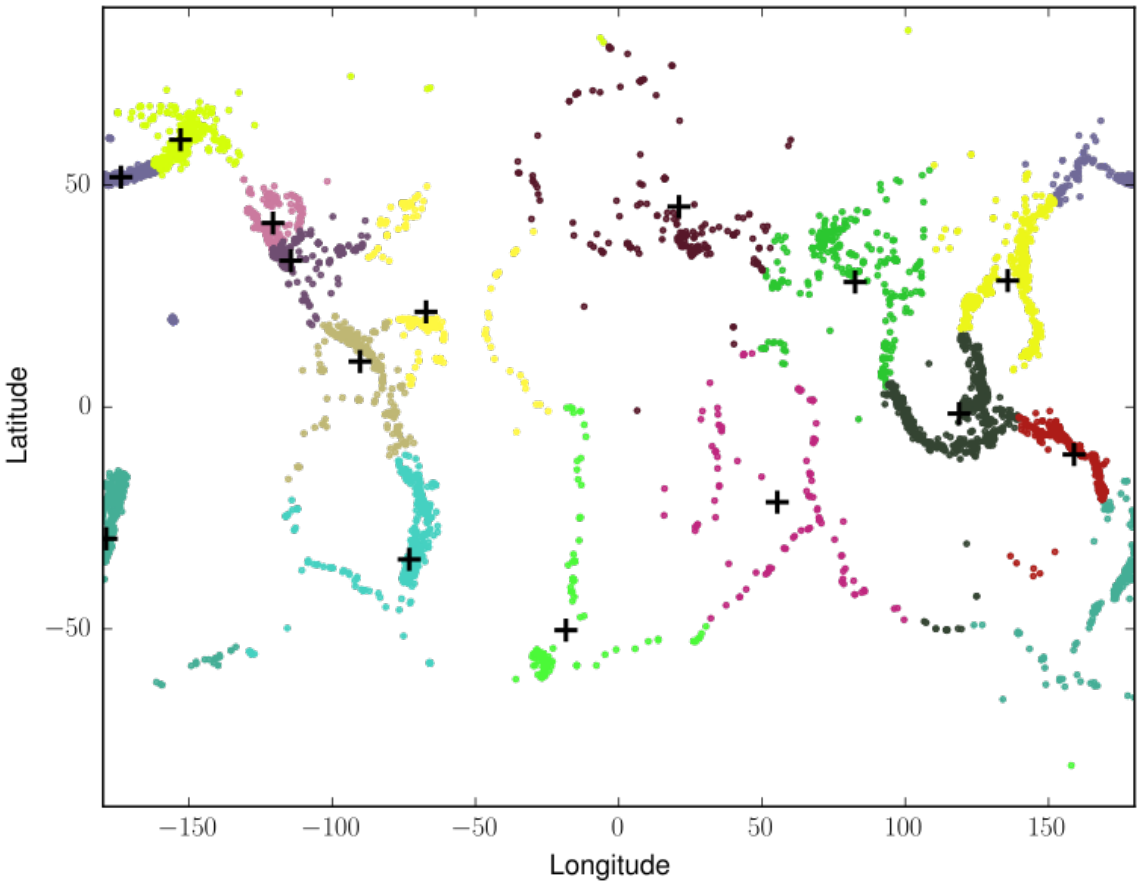


Figure 12.4: Earthquake epicenter clusters with $N = 15$.

We are now ready to cluster the earthquake data using the Euclidean coordinates. We need to address one further issue, however. Notice that each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. We also need to ensure that our cluster means have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth. Furthermore, the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors. Thus, we need to add optional functionality to our `kmeans` function.

Problem 4. Add a keyword argument `normalize=False` to your `kmeans` function, and add code to normalize the means at each iteration, should this argument be set to `True`. Use your function to cluster the earthquake data into 15 clusters. Run this 10 times, keeping the best clustering. Transform the cluster means back to latitude and longitude coordinates (when calculating θ using the inverse tangent, use `numpy.arctan2` or `math.arctan2`, so that that correct quadrant is chosen). Create a scatter plot showing each cluster mean, along with the earthquake epicenters color-coded according to their cluster. Your plot should resemble that of Figure 12.4.

Though plotting our results in two dimensions gives us a good picture, we can see that this is not entirely accurate. There are points that appear to be closer to a different cluster center than the one to which they belong. This comes from viewing the results in only two dimensions. When viewing in three dimensions, we can see more clearly the accuracy of our results.

Problem 5. Add a keyword argument `3d=False` to your `kmeans` function, and add code to show the three-dimensional plot instead of the two-dimensional scatter plot should this argument be set to `True`. Maintain the same color-coding scheme as before. Use `mpl_toolkits.mplot3d.Axes3D` to make your plot.

Spectral Clustering

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

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

for some set value σ.

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

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

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

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

- Compute W, D, and the appropriate Laplacian matrix.
- Compute the first k eigenvectors $u_1, \cdots, u_k$ 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 6. Implement the Spectral Clustering Algorithm by calling your `kmeans` function, using the following function declaration:

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

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

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

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

Problem 7. Create a function that will return the accuracy of your spectral clustering implementation, as follows:

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

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

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