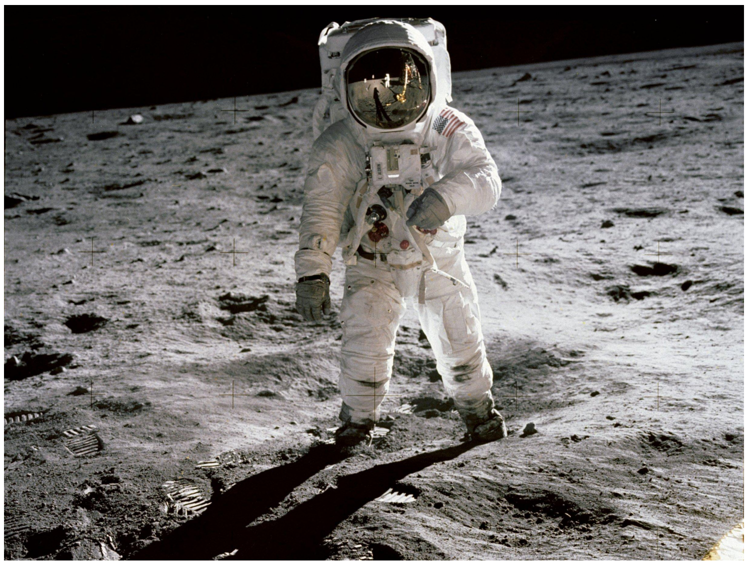
Labs for Foundations of Applied Mathematics

Volume 3
Modeling with Uncertainty and Data

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

E. Hannesson
Brigham Young University

K. Harmer
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

R. Jenkins
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
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

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

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

Preface

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

Contents

Part I

Labs

1 Metropolis Algorithm

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

The Metropolis Algorithm

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

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

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

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 1.1. The Metropolis algorithm can be interpreted as follows: given our current state $\mathbf{y}$, we propose a new state according to the distribution $Q(\cdot, \mathbf{y})$. We then accept or reject it according to A. We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h. Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

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

Algorithm 1.1 Metropolis Algorithm

```
1: procedure Metropolis Algorithm
2:     Choose initial point $\mathbf{y}_0$.
3:     for $t = 1, 2, \ldots$ do
4:         Draw $\mathbf{x} \sim Q(\cdot, \mathbf{y}_{t-1})$
5:         Draw $a \sim \text{unif}(0, 1)$
6:         if $a \leq A(\mathbf{x}, \mathbf{y}_{t-1})$ then
7:             $\mathbf{y}_t = \mathbf{x}$
8:         else
9:             $\mathbf{y}_t = \mathbf{y}_{t-1}$
10:    Return $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots$
```

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

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

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

$$p(\mu, \sigma^2 \,|\, y_1, \ldots, y_N) = \frac{p(\mu)p(\sigma^2)\prod_{i=1}^n \mathcal{N}(y_i \,|\, \mu, \sigma^2)}{\int_{-\infty}^{\infty}\int_0^{\infty} p(\mu)p(\sigma^2)\prod_{i=1}^n \mathcal{N}(y_i \,|\, \mu, \sigma^2)\, d\sigma^2 d\mu}.$$

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

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

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

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

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

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

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

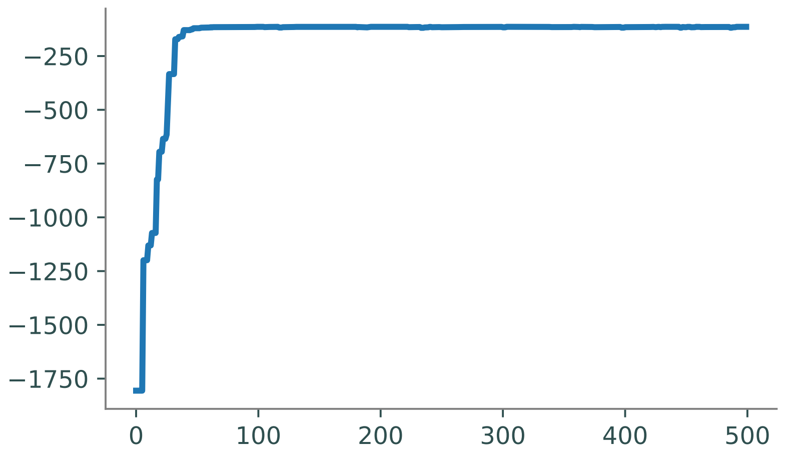


Figure 1.1: Log densities of the first 500 Metropolis samples.

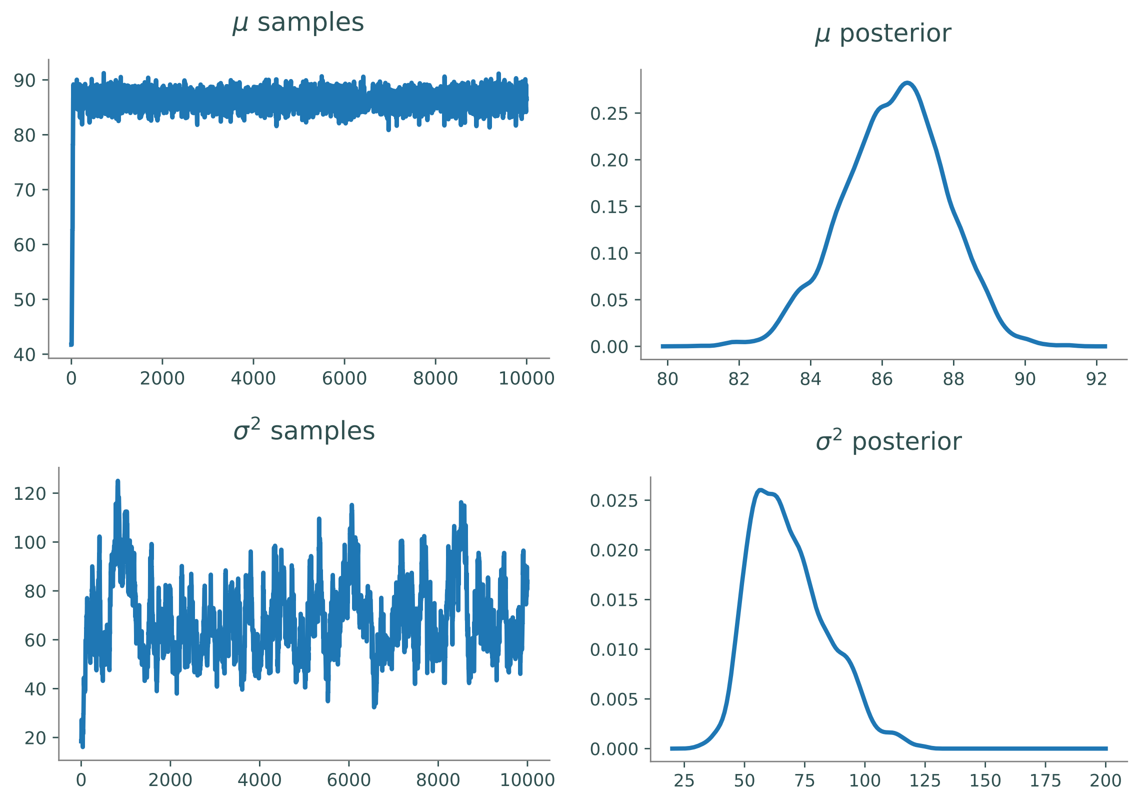


Figure 1.2: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

Problem 1. Write a function that uses the Metropolis Hastings algorithm to draw from the posterior distribution over the mean μ and variance σ^2. Use the given functions and algorithm 1.1 to complete the problem.

Your function should return an array of draws, an array of the log probabilities, and an acceptance rate. Use the following code to check your work. Plot the first 500 log probabilities, the μ samples and posterior distribution, and the σ^2 samples and posterior distribution. The results should be *similar* to Figures 1.1 and 1.2.

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

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

#Prior mu ~ N(m, s)
>>> m = 80
>>> s = 4
```

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

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

The Ising Model

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

With any spin configuration σ, there is an associated energy

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

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

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

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

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

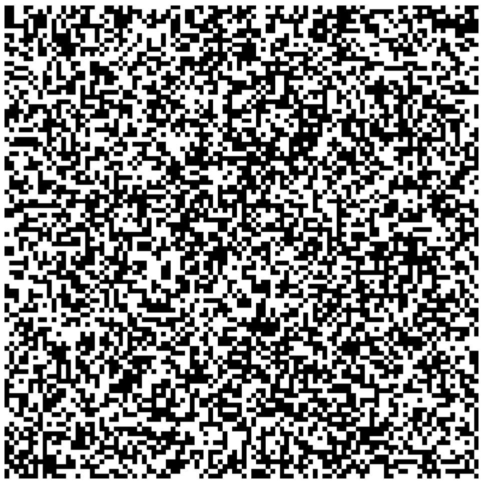


Figure 1.3: Spin configuration from random initialization.

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

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

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

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

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

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

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma)-H(\sigma^*))} & \text{otherwise.} \end{cases} \tag{1.1}$$

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

$$H(\sigma^*) = H(\sigma) + 2 \sum_{j:j\sim i} \sigma_i \sigma_j. \tag{1.2}$$

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

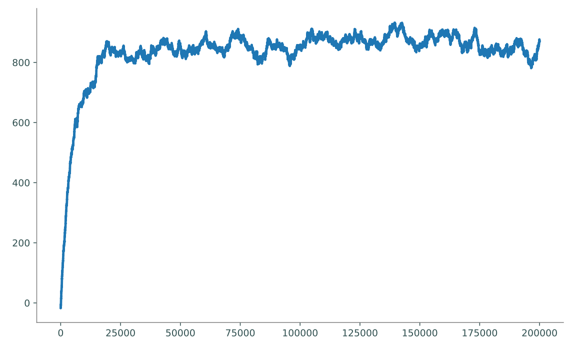
Problem 4. Write a function that accepts a spin configuration σ, its energy $H(\sigma)$, and integer indices i and j. Use (1.2) to compute the energy of the new spin configuration σ^*, which is σ but with the spin flipped at the (i, j)th entry of the corresponding lattice. Do not explicitly construct the new lattice for σ^*.

Problem 5. Write a function that accepts a float β and spin configuration energies $H(\sigma)$ and $H(\sigma^*)$. Using (1.1), calculate whether or not the new spin configuration σ^* should be accepted (return `True` or `False`). Consider doing the calculations in log space.

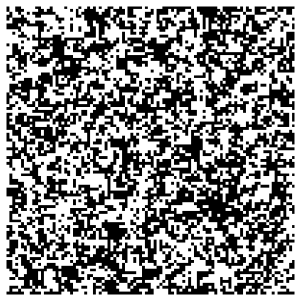
To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator Z_β, which is generally the reason we have to use a Metropolis algorithm to begin with. We can get away with examining only $-\beta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

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

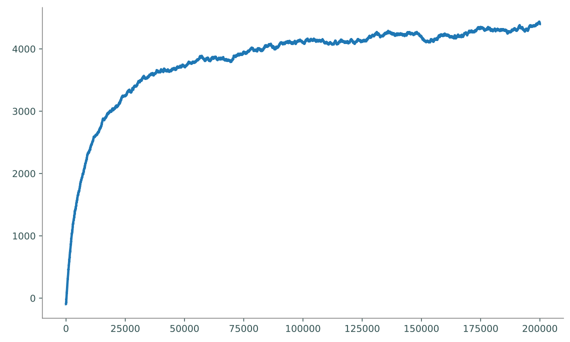
1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration σ^*.
2. Use Problem 4 to calculate the energy $H(\sigma^*)$ of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set $\sigma = \sigma^*$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



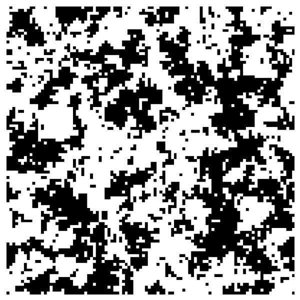
(a) Proportional log probs when $\beta = 0.2$.



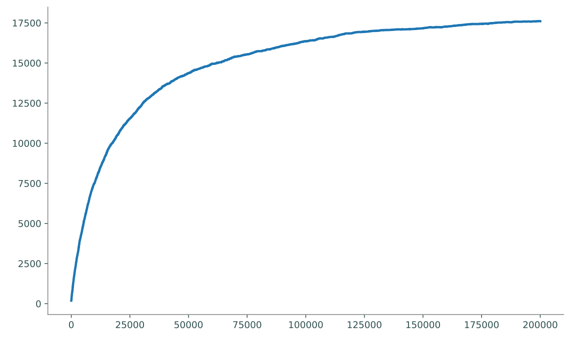
(b) Spin configuration sample when $\beta = 0.2$.



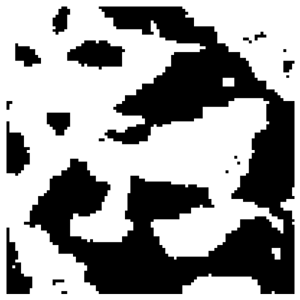
(c) Proportional log probs when $\beta = 0.4$.



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



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



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

Figure 1.4

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

Test your sampler on a 100×100 grid with 200000 total iterations, with `n_samples` large enough so that you will keep 50 samples, for $\beta = 0.2, 0.4, 1$. Plot the proportional log probabilities, as well as a late sample from each test. How does the ferromagnetic material behave differently with differing temperatures? Recall that β is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figure 1.4.

2 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 2.1 Basic Gibbs Sampling Process.

```
1: procedure Gibbs Sampler
2:     Randomly initialize $x_1, x_2, \ldots, x_n$.
3:     for $k = 1, 2, 3, \ldots$ do
4:         for $i = 1, 2, \ldots, n$ do
5:             Draw $x \sim \mathbb{P}(x_i | \mathbf{x}_{-i}, \mathbf{y})$
6:             Fix $x_i = x$
7:         $\mathbf{x}^{(k)} = (x_1, x_2, \ldots, x_n)$
```

A Gibbs sampler proceeds according to Algorithm 2.1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of $\mathbf{x}^{(k)}$ after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible $\mathbf{x}$. The unique invariant distribution for the chain is the desired joint distribution

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

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

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

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

Letting $\mathbf{y} = (y_1, \ldots, y_N)$ 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:

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

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

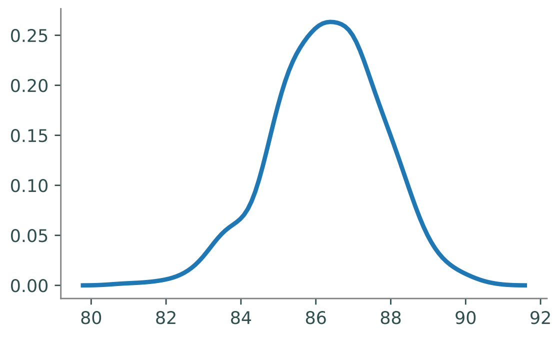
Problem 1. Write a function that accepts data $\mathbf{y}$, prior parameters ν, τ^2, α, and β, and an integer n. Use Gibbs sampling to generate n samples of μ and σ^2 for the exam scores problem.

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

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

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

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


(a) Posterior distribution of μ.

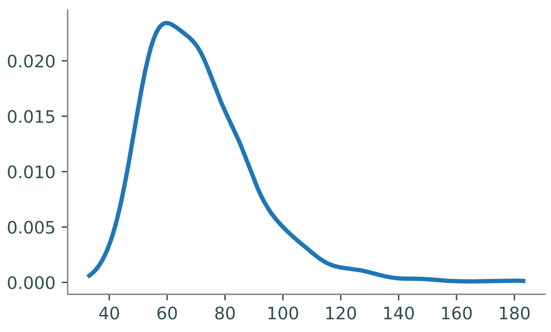

(b) Posterior distribution of σ^2.

Figure 2.1: Posterior marginal probability densities for μ and σ^2.

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

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

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

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

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

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

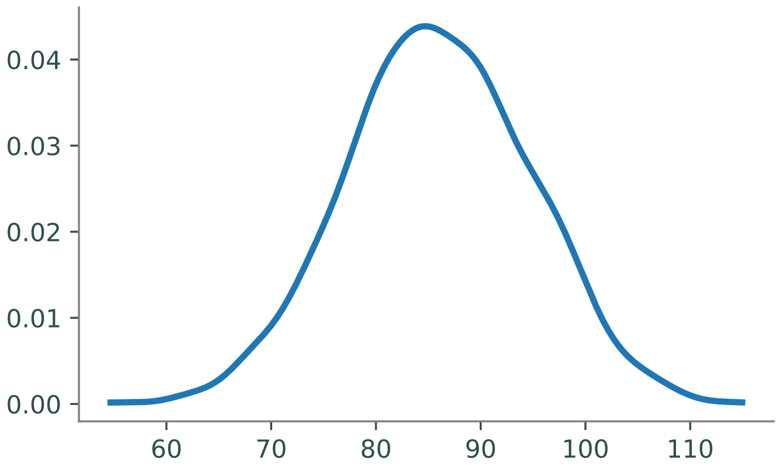


Figure 2.2: Predictive posterior distribution of exam scores.

Problem 2. Plot the kernel density estimators for the posterior distributions of μ and σ^2. You should get plots similar to those in Figure 2.1.

Next, use your samples of μ and σ^2 to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. Compare your plot to Figure 2.2.

Latent Dirichlet Allocation

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

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

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

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

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

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

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

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

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

This is typically depicted with graphical plate notation as in Figure 2.3.

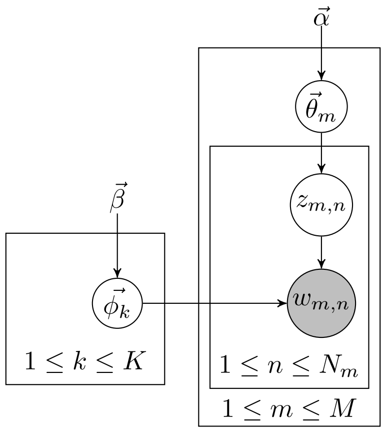


Figure 2.3: Graphical plate notation for LDA text generation.

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:

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

where

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

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

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

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

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

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

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

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

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

We are now prepared to write the full Gibbs sampler.

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

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

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

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

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

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

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

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

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

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

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

Additional Material

LDACGS Source Code

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

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

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

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

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

        # Initialize the topic assignment dictionary.
        self.topics = {} # key-value pairs of form (m,i):z
        for m in range(self.n_docs):
            for i in self.documents[m]:
                # Get random topic assignment, i.e. z = ...
                # Increment count matrices
                # Store topic assignment, i.e. self.topics[(m,i)]=z
                raise NotImplementedError("Problem 3 Incomplete")

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

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

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

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

    def toplines(self,n_lines=5):
        lines = np.zeros((self.n_topics,n_lines))
        for i in range(self.n_topics):
            args = np.argsort(self._theta[:,i]).tolist()
            args.reverse()
            lines[i,:] = np.array(args)[0:n_lines] + 1
        return lines

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

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

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

    def _loglikelihood(self):
        lik = 0

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

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

        return lik
```

3 Speech Recognition using CDHMMs

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

3.0.1 Continuous Density Hidden Markov Models

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

In order to understand GMMHMMs, we need to be familiar with a particular continuous, multivariate distribution called a *mixture of Gaussians*. A mixture of Gaussians is a distribution composed of several Gaussian (or Normal) distributions with corresponding weights. Such a distribution is parameterized by the number of mixture components M, the dimension N of the normal distributions involved, a collection of component weights $\{c_1, \ldots, c_M\}$ that are nonnegative and sum to 1, and a collection of mean and covariance parameters $\{(\mu_1, \Sigma_1), \ldots, (\mu_M, \Sigma_M)\}$ for each Gaussian component. To sample from a mixture of Gaussians, one first chooses the mixture component i according to the probability weights $\{c_1, \ldots, c_M\}$, 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 3.1 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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ and a corresponding observation sequence $\{O_1, \ldots, O_T\}$, 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 $i = 1, \ldots, N$, we have component weights $\{c_{i,1}, \ldots, c_{i,M}\}$, component means $\{\mu_{i,1}, \ldots, \mu_{i,M}\}$, and component covariance matrices $\{\Sigma_{i,1}, \ldots, \Sigma_{i,M}\}$.

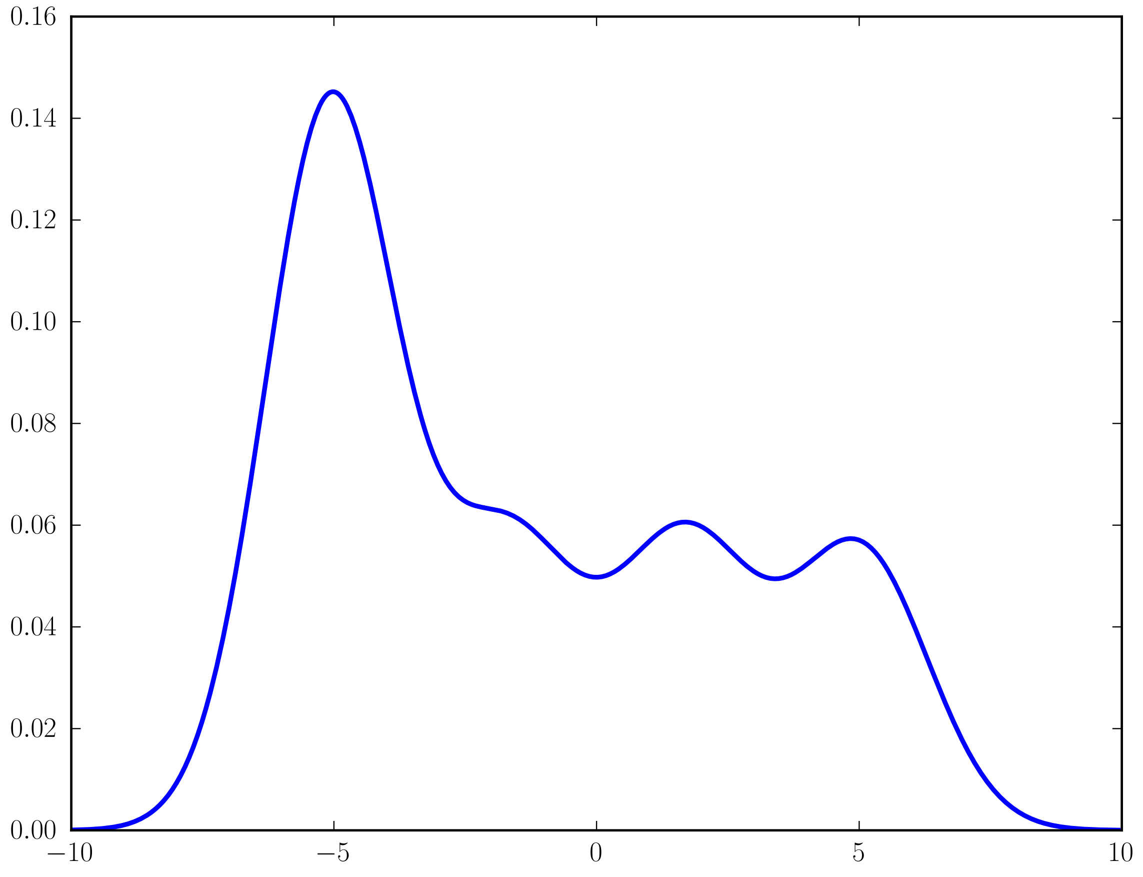


Figure 3.1: The probability density function of a mixture of Gaussians with four components.

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]
```

As explained above, to sample from a GMMHMM, we draw a sample from one of the Gaussians $\mathcal{N}(\mu_i, \Sigma_i)$, with component i chosen according to the probably weights $\{c_1, \ldots, c_M\}$. We 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,:]))
```

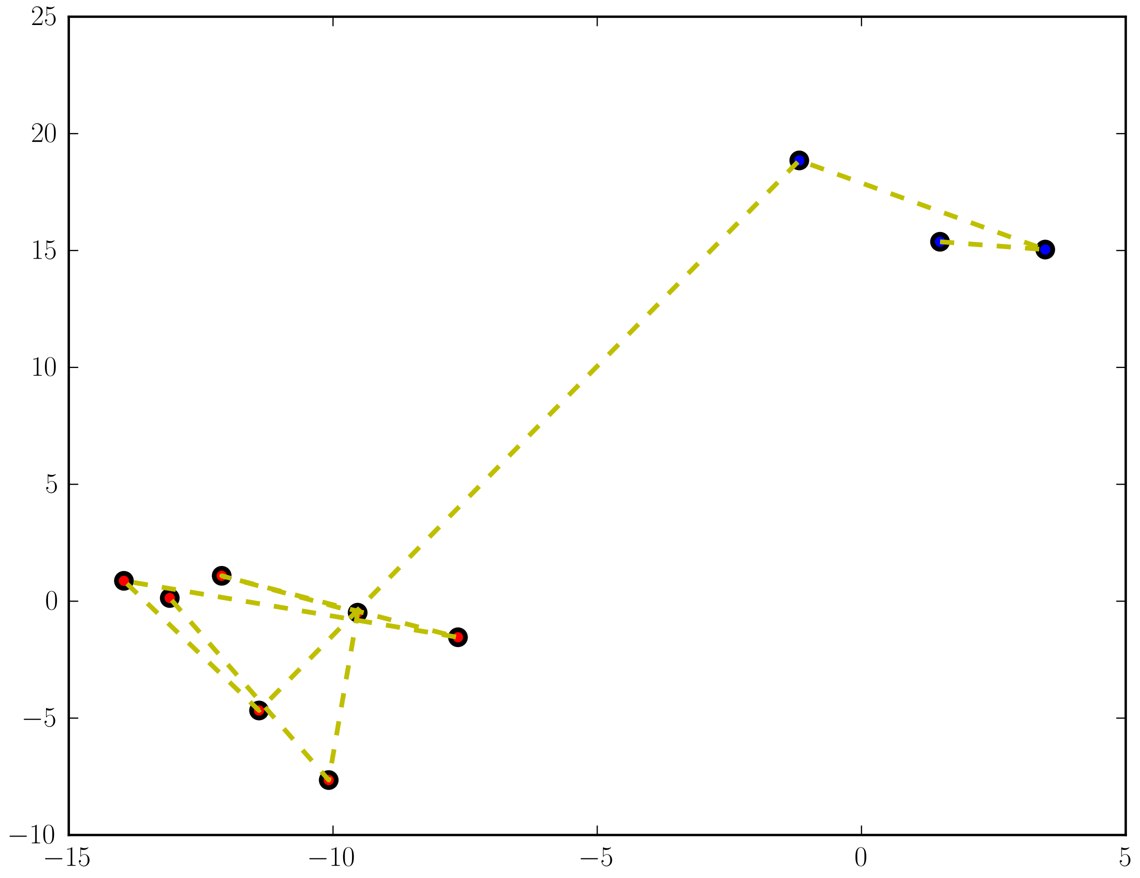


Figure 3.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

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

Figure 3.2 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 30 MFCC arrays, corresponding to each of the five words under consideration.

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

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

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

```
>>> import gmmhmm
>>> 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.

4 K-Means Clustering

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

Clustering

Previously, we analyzed the iris dataset from `sklearn` using PCA; we have reproduced the first two principal components of the iris data in Figure 4.1. 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 $X = \{x_1, x_2, \ldots, x_n\}$. Let $N \in \mathbb{N}$ and let $\mathcal{S}$ be the set of all N-partitions of X, where an N-partition is a partition with exactly N nonempty elements. We can represent a typical partition in $\mathcal{S}$ as $S = \{S_1, S_2, \ldots, S_N\}$, where

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

and

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

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

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

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

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

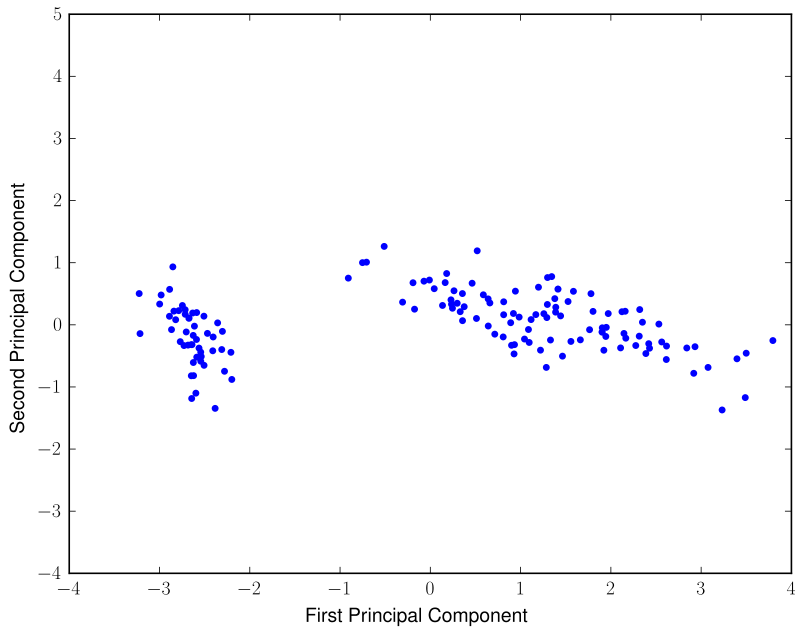


Figure 4.1: The first two principal components of the iris dataset.

The K-Means Algorithm

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

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

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

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

Figure 4.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations. The algorithm can be summarized as follows.

1. Choose k initial cluster centers.
2. For $i = 0, \ \ldots, $ `max_iter`,
   (a) Assign each data point to the cluster center that is closest, forming k clusters.
   (b) Recompute the cluster centers as the means of the new clusters.
   (c) If the old cluster centers and the new cluster centers are sufficiently close, terminate early.

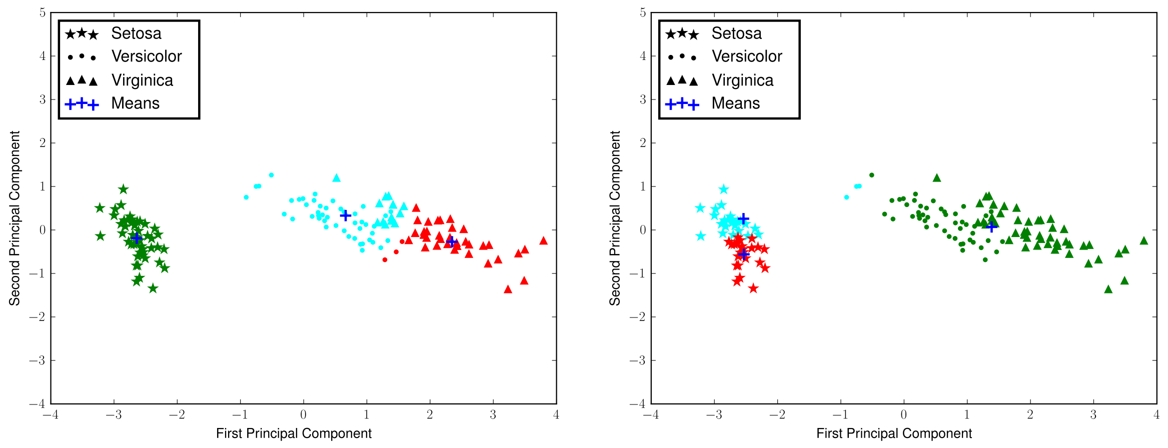


Figure 4.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

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

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

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

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

Detecting Active Earthquake Regions

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

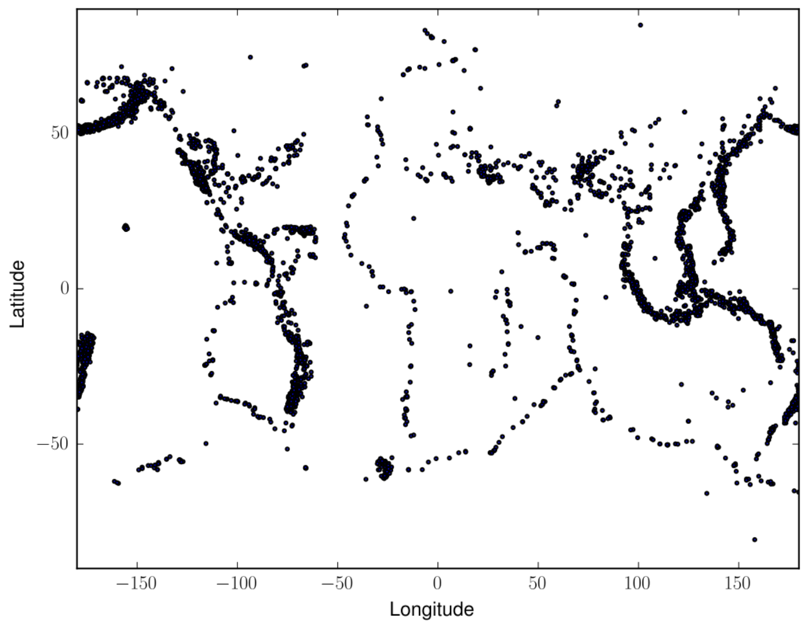


Figure 4.3: Earthquake epicenters over a 6 month period.

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

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

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

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

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

Problem 2. Add a keyword argument `normalize=False` to your `KMeans` constructor. Modify `fit()` so that if `normalize` is `True`, the cluster centers are normalized at each iteration.

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

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

With 15 clusters, your plot should resemble the Figure 4.4.

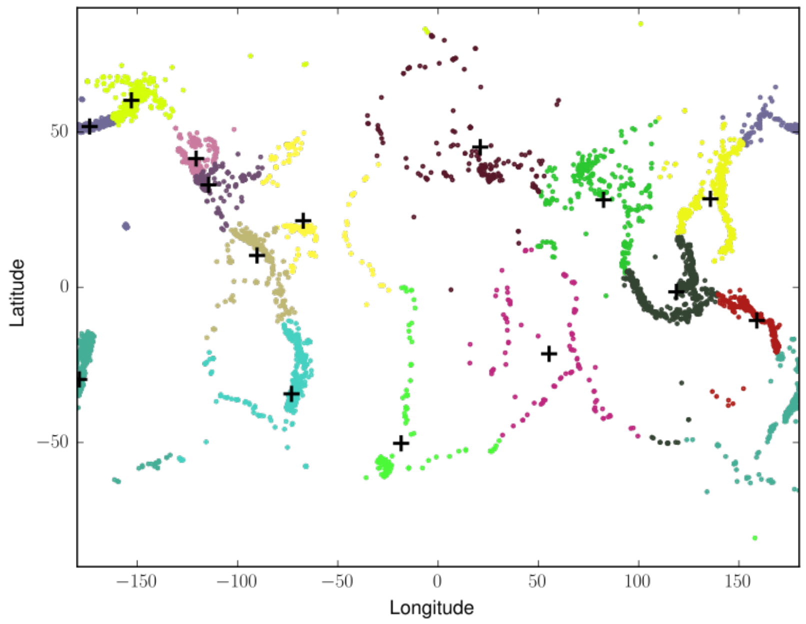


Figure 4.4: Earthquake epicenter clusters with $k = 15$.

Color Quantization

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

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

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

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

Test your function on some of the provided NASA images.

Additional Material

Spectral Clustering

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

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

for some set value σ.

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

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

1. The *unnormalized Laplacian*, $L = D - W$
2. The *symmetric normalized Laplacian*, $L_{sym} = I - D^{-1/2}WD^{-1/2}$
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 4. 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 5. 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
```

5 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

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + B_k\mathbf{u}_k + \boldsymbol{\varepsilon}_k \tag{5.1}$$

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

$$\mathbf{z}_k = H_k\mathbf{x}_k + \boldsymbol{\delta}_k \tag{5.2}$$

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:

```python
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 5.1 and 5.2.

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:

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

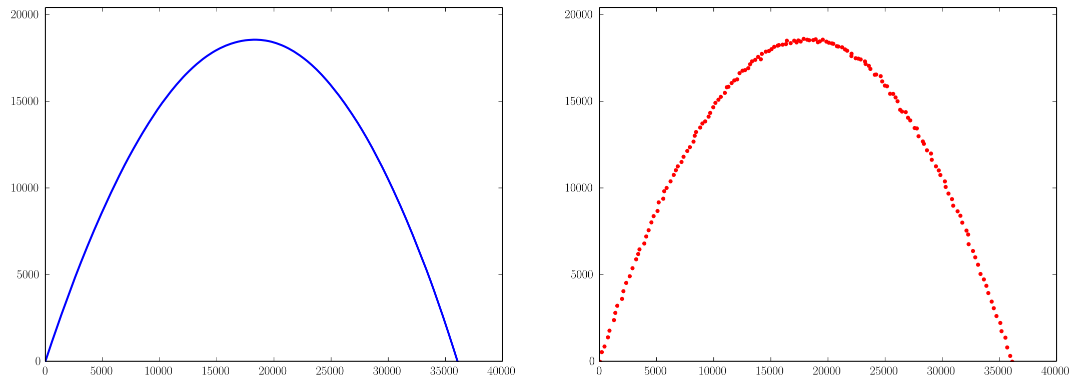


Figure 5.1: State sequence (left) and sampling of observation sequence (right).

```
    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 5.1.

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:

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

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:

```python
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).
```

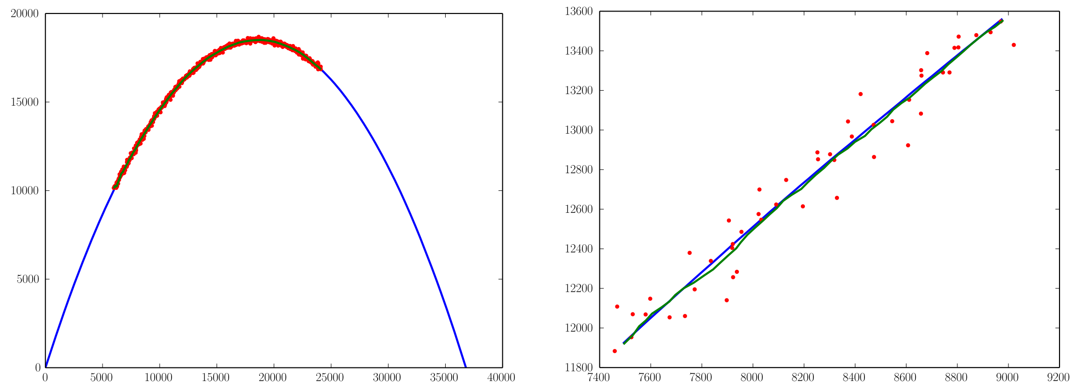


Figure 5.2: State estimates together with observations and true state sequence (detailed view on the right).

```
    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 5.2.

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

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

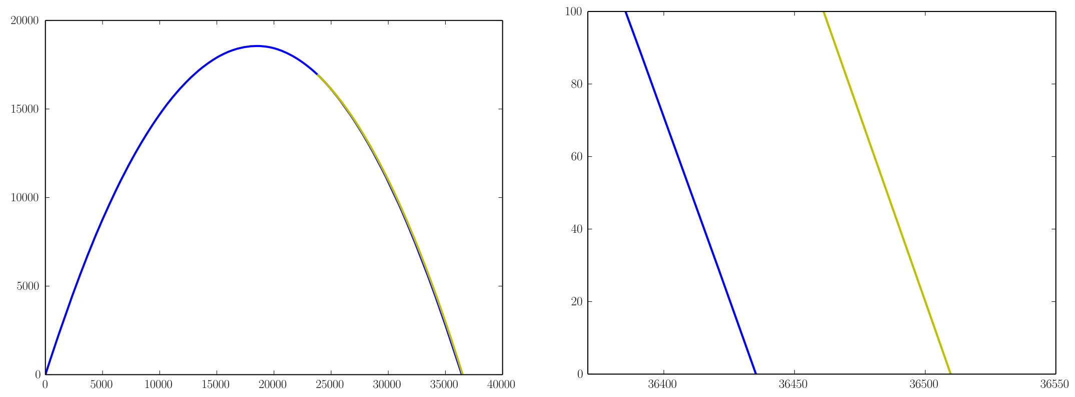


Figure 5.3: Predicted vs. actual point of impact (detailed view on right).

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 5.3.

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.

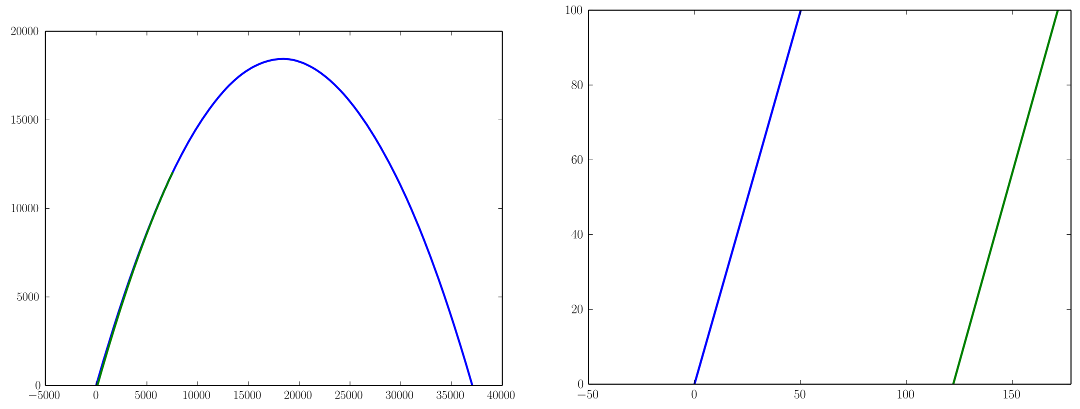


Figure 5.4: Predicted vs. actual point of origin (detailed view on right).

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:

```python
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 5.4.

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

6 ARMA Models

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

Time Series

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

ARMA(p, q) Models

One way to forecast a time series is using an ARMA model. An ARMA(p, q) model combines an autoregressive model of order p and a moving average model of order q on a time series $\{y_t\}_{t=1}^n$. This model is a dependent model as it is non-independent of previous data. Because of this, the model needs to become stationary in order to compensate for the dependency of the data. To make data stationary, we look at the time series $\{z_t\}_{t=1}^n$ where $z_t = y_t - y_{t-1}$. The model itself is a stochastic process on z_t, satisfying the equation

$$z_t = \underbrace{\left(\sum_{i=1}^{p} \phi_i z_{t-i}\right)}_{\text{AR(p)}} + \varepsilon_t + \underbrace{\left(\sum_{j=1}^{q} \theta_j \varepsilon_{t-j}\right)}_{\text{MA(q)}} \tag{6.1}$$

where each ε_t is an identically-distributed Gaussian variable $\mathscr{N}(\mu, \sigma^2)$, and ϕ_i and θ_j are constants.

AR(p) Models

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

$$z_t = c + \varepsilon_t + \sum_{i=1}^{p} \phi_i z_{t-i} \tag{6.2}$$

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

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

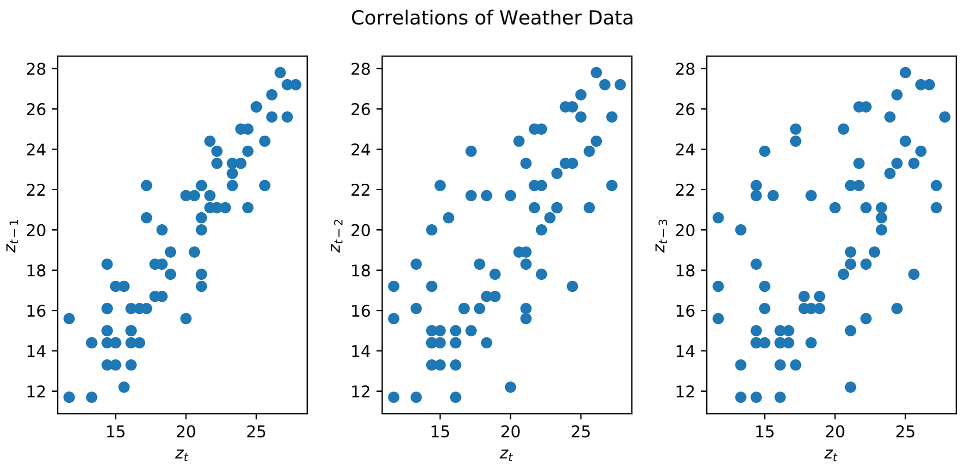


Figure 6.1: These graphs show that the weather data is correlated to its previous values. The correlation is weaker in each graph successively, showing that the further in the past the data is, the less correlated the data becomes.

MA(q)

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

$$z_t = c + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \tag{6.3}$$

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

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

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

Finding Parameters

One of the most difficult parts of using an ARMA(p, q) model is identifying the proper parameters of the model. These parameters include $\{\phi_i\}_{i=1}^{p}$, $\{\theta_i\}_{i=1}^{q}$, μ, and σ, where μ and σ are the mean and variance of the error. Note that $\{\phi_i\}_{i=1}^{p}$ and $\{\theta_i\}_{i=1}^{q}$ determine the order of the ARMA model.

A naive way to use an ARMA model is to choose p and q based on intuition. Figure 6.1 showed that there is a strong correlation between z_t and z_{t-1} and between z_t and z_{t-2}. The correlation is weaker between z_t and z_{t-3}. Intuition then suggests to choose $p = 2$. By looking at the correlations between the current noise with previous noise, similar to Figure 6.1, it can also be seen that there is a weak correlation between z_t and ε_t and between z_t and ε_{t-1}. Between z_t and ε_{t-2} there is no correlation. For more on how these error correlations were found, see Additional Materials. Intuition from these correlations suggests to choose $q = 1$. Thus, a naive choice for our model is an ARMA$(2, 1)$ model.

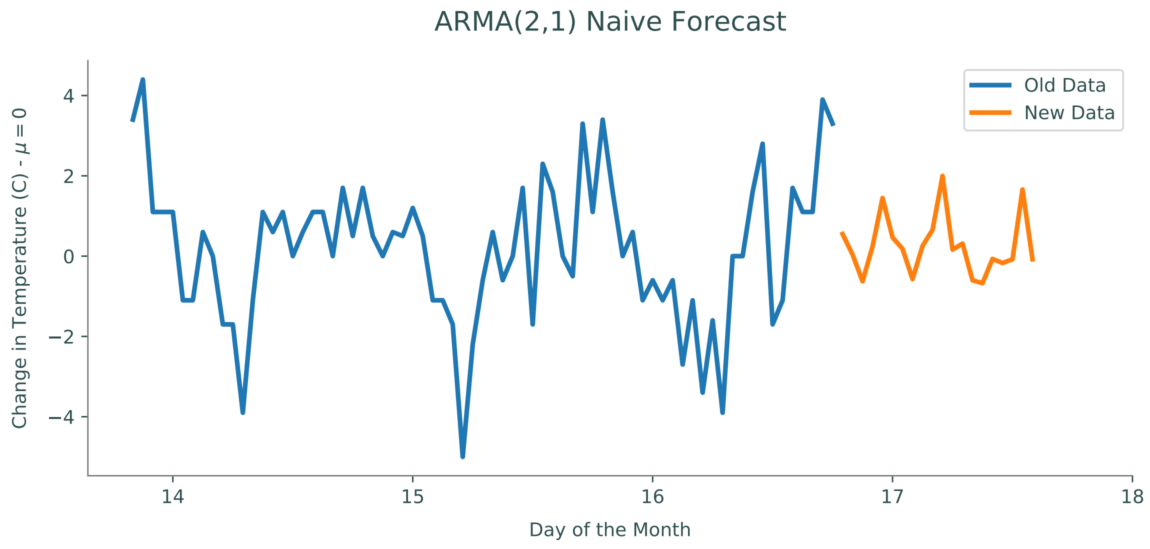


Figure 6.2: Naive forecast on `weather.npy`

Problem 1. Write a function `arma_forecast_naive()` that builds an ARMA(p,q) model where the values of $\phi_i = .5$ and $\theta_i = .1$ for all i. Let $\varepsilon_i \sim \mathcal{N}(0, 1)$ for all i. Use your function to predict the next n values of the time series. The function should accept a parameter p, q, and n (the number of observations to predict). Plot $\{z_t\}_{t=1}^{n}$ followed by your predicted observations of z_t.

The file `weather.npy` contains data on the temperature in Provo, Utah from 7:56 PM May 13, 2019 to 6:56 PM May 16, 2019, taken every hour. Use this file to test your code. For $p = 2$, $q = 1$, and $n = 20$, your plot should look similar to Figure 6.2, however, due to the variance of the error ε_t, the plot will not look exactly like Figure ??. The predictions may be higher or lower on the x-axis.

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

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} p(z_t|z_{t-1}, \dots, z_1, \Theta) \tag{6.4}$$

State Space Representation

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

$$\hat{\mathbf{x}}_{t|t-1} = \begin{bmatrix} x_{t-1} & x_{t-2} & \cdots & x_{t-r} \end{bmatrix}^T \tag{6.5}$$

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{6.6}$$

$$H = \begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_{r-1} \end{bmatrix} \tag{6.7}$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{6.8}$$

$$w_t \sim \text{MVN}(0, Q), \tag{6.9}$$

where $\phi_i = 0$ for $i > p$, and $\theta_j = 0$ for $j > q$. Note that Equation 6.2 gives

$$F\hat{\mathbf{x}}_{t-1|t-2} + w_t = \begin{bmatrix} \sum_{i=1}^r \phi_i x_{t-i} \\ x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-(r-1)} \end{bmatrix} + \begin{bmatrix} \varepsilon_t \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{6.10}$$

$$= \begin{bmatrix} x_t & x_{t-1} & \cdots & x_{t-(r-1)} \end{bmatrix}^T \tag{6.11}$$

$$= \hat{\mathbf{x}}_{t|t-1} \tag{6.12}$$

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

Then the linear stochastic dynamical system

$$\hat{\mathbf{x}}_{t+1|t} = F\hat{\mathbf{x}}_{t|t-1} + w_t \tag{6.13}$$

$$z_{t|t-1} = H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{6.14}$$

describes the same process as the original ARMA model.

Note

Equation 6.14 involves a deterministic component, namely μ. The Kalman filter theory developed in the previous lab, however, assumed $\mathbb{E}[\varepsilon_t] = 0$ for the observations $z_{t|t-1}$,. This means you should subtract off the mean μ of the error from the time series observations $z_{t|t-1}$ when using them in the predict and update steps.

Likelihood via Kalman Filter

We assumed in Equation 6.9 that the error terms of the model are Gaussian. This means that each conditional distribution in 6.4 is also Gaussian, and is completely characterized by its mean and variance. These two quantities are easily found via the Kalman filter:

$$\text{mean} \quad H\hat{\mathbf{x}}_{t|t-1} + \mu \tag{6.15}$$

$$\text{variance} \quad HP_{t|t-1}H^T \tag{6.16}$$

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

$$p(\{z_t\}|\Theta) = \prod_{t=1}^{n} N(z_t \mid H\hat{\mathbf{x}}_{t|t-1} + \mu,\ HP_{t|t-1}H^T) \tag{6.17}$$

Problem 2. Write a function `arma_likelihood()` that returns the log-likelihood of an ARMA model, given a time series $\{z_t\}_{t=1}^n$. This function should accept a `file` with the observations and each of the parameters in Θ. Return the log-likelihood of the ARMA(p, q) model as a `float`.

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

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

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

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

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

Model Identification

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

$$2k\left(1+\frac{k+1}{n-k}\right)-2\ell(\Theta) \tag{6.18}$$

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 6.17 over the space of parameters Θ. We can do so by using an optimization routine such as `scipy.optimize.fmin` on the function `arma_likelihood()` from Problem 2. Use the following code to run this routine.

```
>>> from scipy.optimize import fmin

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

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

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

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

Your code should produce the following output (it may take about two minutes to run):

```
>>> model_identification(filename='weather.npy',i=4,j=4)
(array([ 0.72135856]), array([-0.26246788]), 0.35980339870105321, ↩
    1.5568331253098422)
```

Forecasting with Kalman Filter

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

$$z_{t+k}|z_1, \cdots, z_t \sim N(z_{t+k};\ H\hat{x}_{t+k|t} + \mu,\ HP_{t+k|t}H^T) \tag{6.19}$$

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

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

Achtung!

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

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

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

The following code should create a plot similar to Figure 6.3.

```
>>> # Get optimal model
>>> phis, thetas, mu, std = model_identification(filename='weather.npy', i↩
    =4, j=4)

>>> # Forecast optimal mode
```

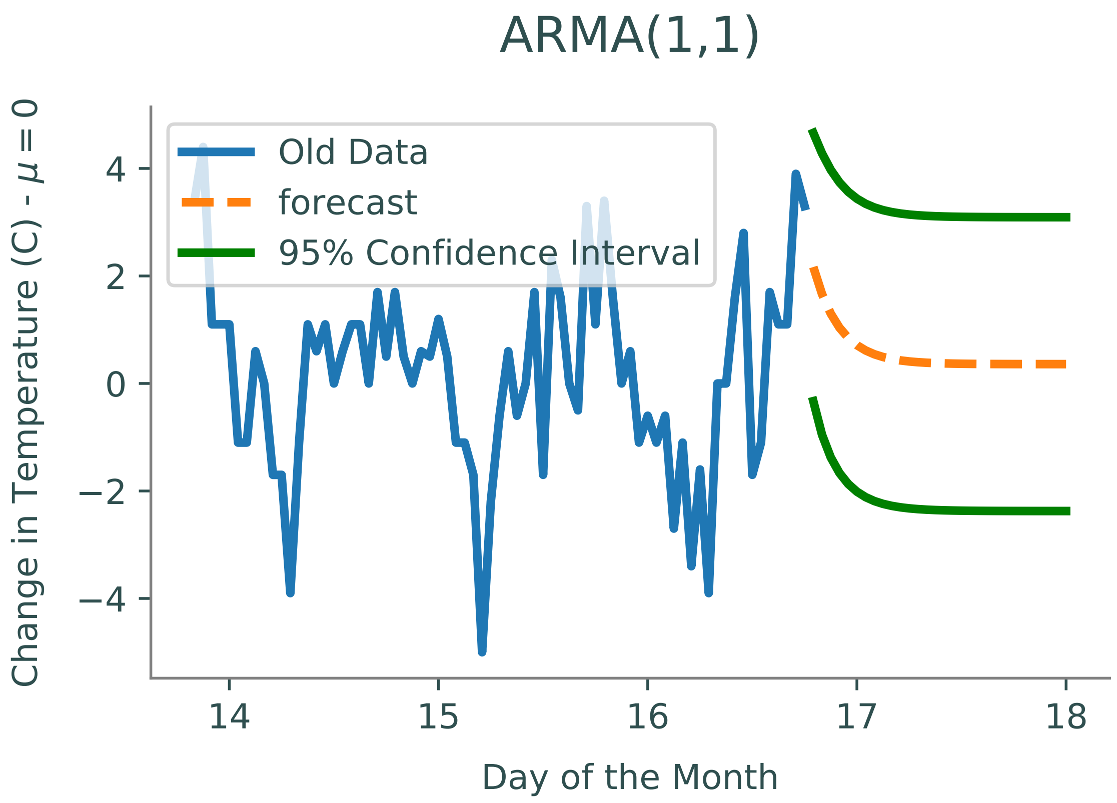


Figure 6.3: ARMA(1,1) forecast on `weather.npy`

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

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

Statsmodel ARMA

The module `statsmodels` contains a package that includes an ARMA model class. This class also uses a Kalman Filter to calculate the MLE. When creating an ARMA object, initialize the variables `endog` (the data) and `order` (the order of the model). The object can then be fitted based on the MLE using a Kalman Filter.

```
from statsmodels.tsa.arima_model import ARMA
# Intialize the object with weather data and order (1,1)
model = ARMA(data,order=(1,1))
# Fit model using MLE and allowing for a constant if needed
model.fit(method='mle', trend='c')
```

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

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

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

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

For $p = 3, q = 3$, and $n = 30$, your graph should look similar to Figure 6.4. How does this graph compare to Problem 1? Problem 4?

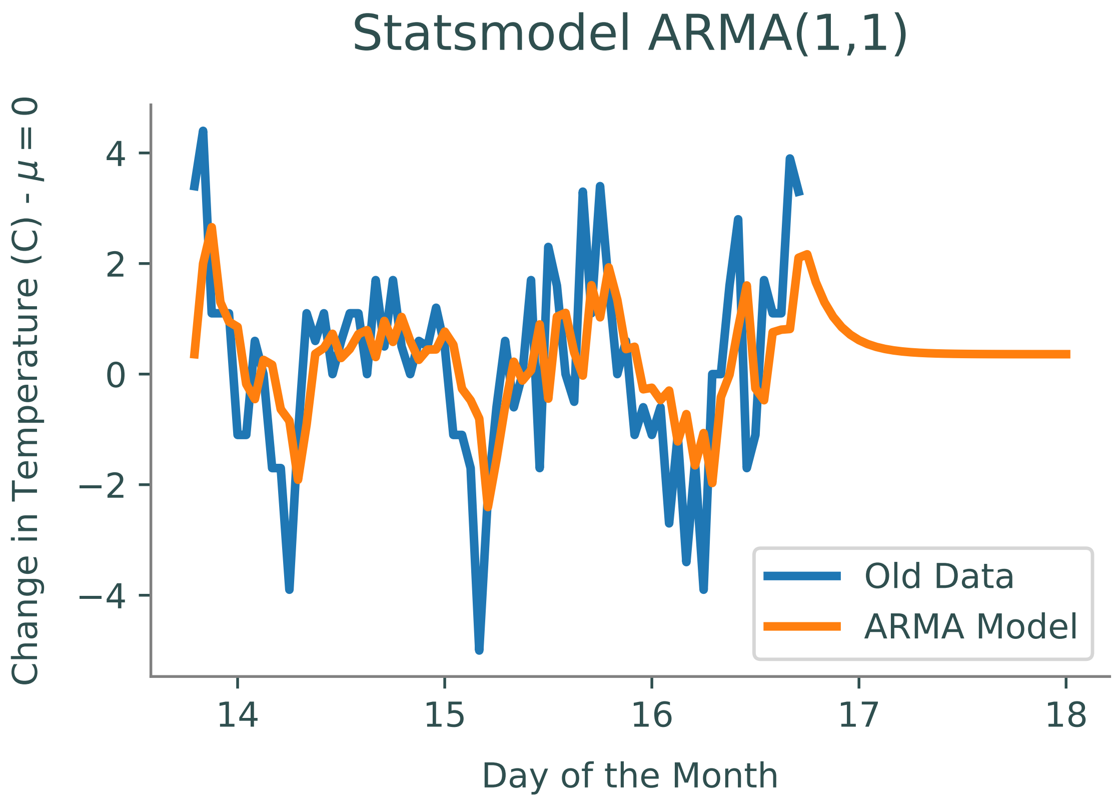


Figure 6.4: Statsmodel ARMA(3,1) forecast on `weather.npy`.

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

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

>>> # Get sunspot data and give DateTimeIndex
>>> sunspot = sm.datasets.sunspots.load_pandas().data[['SUNACTIVITY']]
>>> sunspot.index = pd.Index(sm.tsa.datetools.dates_from_range('1700', '2008'))

>>> # Find best order where p < 5 and q < 5
>>> # Use AICc as basis for minimization
>>> order = order_select(sunspot.values,max_ar=4,max_ma=4,ic=['aic','bic'],↩
    fit_kw={'method':'mle'})
>>> print(order['aic_min_order'])
(4,2)
>>> print(order['bic_min_order'])
(4,2)
```

The method plot_predict accepts a time series and plots the ARMA model alongside the original data in a given range. The plot of the ARMA model is the mean calculated by ARMA at each data point, both known and future. This method works by giving a range on which to plot the ARMA model. This range can be given by indices (as in Problem 5) or by a DateTimeIndex.

```
>>> # Fit model
>>> model = ARMA(dta, (4, 2)).fit(method='mle')

>>> # Create plot
>>> fig, ax = plt.subplots(figsize=(13,7))
>>> # Plot from 1950 to 2012.
>>> fig = model.plot_predict(start='1950', end='2012', ax=ax)

>>> ax.set_title('Sunspot Dataset')
>>> ax.set_xlabel('Year')
>>> ax.set_ylabel('Number of Sunspots')
>>> plt.show()
```

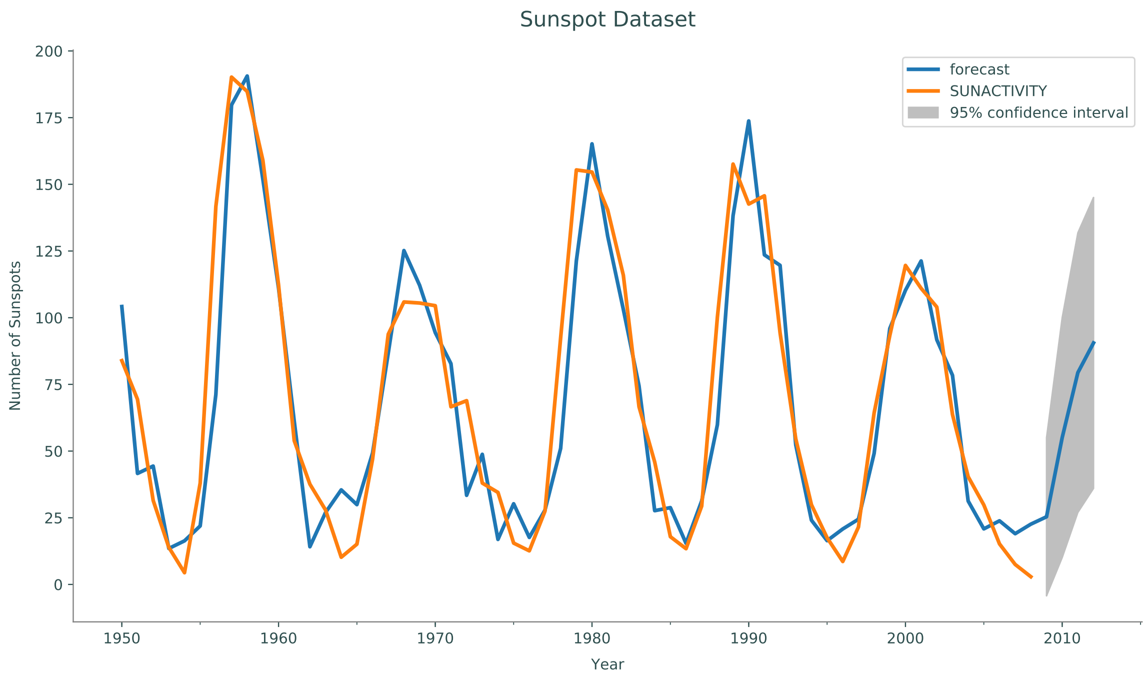


Figure 6.5: Sunspot activity data is forecasted four years in the future using `statsmodels`.

Problem 6. The dataset `manaus` contains data on the height of the Rio Negro from every month between January 1903 and January 1993. Write a function `manaus()` that accepts the forecasting range as strings `start` and `end`, the maximum parameter for the AR model `p` and the maximum parameter of the MA model `q`. The parameters `start` and `end` should be strings corresponding to a DateTimeIndex in the form `Y%M%D`, where `D` is the last day of the month.

The function should determine the optimal order for the ARMA model based on the AIC and the BIC. Then forecast and plot on the range given for both models and compare. Return the order of the AIC model and the order of the BIC model, respectively. For the range `'1983-01-31'` to `'1995-01-31'`, your plot should look like Figure 6.6.

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

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

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

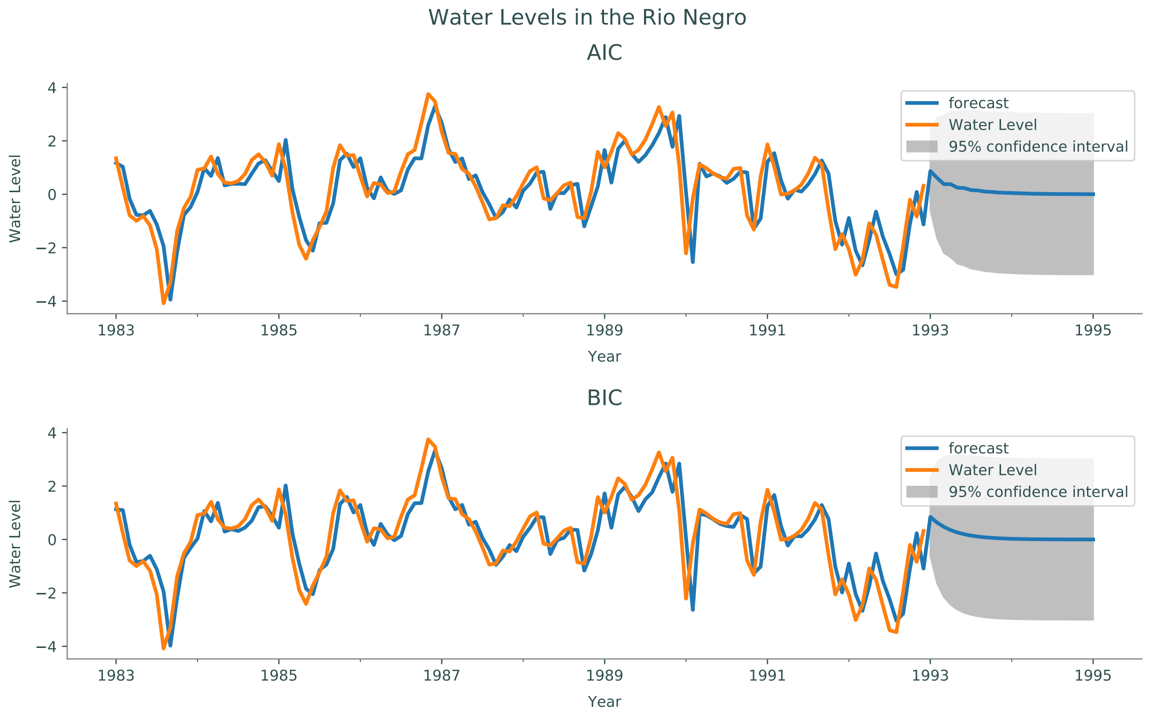


Figure 6.6: AIC and BIC based ARMA models of `manaus` dataset.

Additional Materials

Finding Error Correlation

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

$$y_t = T_t + S_t + R_t, \tag{6.20}$$

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

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

$$T_t = \frac{1}{M} \sum_{-M/2<i<M/2} y_{i+t}. \tag{6.21}$$

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

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

$$S_t = \frac{1}{K} \sum_{i \equiv t \pmod{M}} x_i \tag{6.22}$$

where K is the number of seasons.

With the overall and seasonal trend known, the noise of the data is simply $R_t = y_t - T_t - S_t$. To determine the strength of correlations with the current error and the past error, plot y_t vs. R_{t-i} as in Figure 6.1.

Proof of Equation 6.14

$$\begin{aligned}
\sum_{i=1}^{p}\phi_i(z_{t-i}-\mu)+a_t+\sum_{j=1}^{q}\theta_j a_{t-j} &= \sum_{i=1}^{p}\phi_i(H\hat{\mathbf{x}}_{t-i})+a_t+\sum_{j=1}^{q}\theta_j a_{t-j} && (6.23)\\
&= \sum_{i=1}^{r}\phi_i(x_{t-i}+\sum_{k=1}^{r-1}\theta_k x_{t-i-k})+a_t+\sum_{j=1}^{r-1}\theta_j a_{t-j} && (6.24)\\
&= a_t+\sum_{i=1}^{r}\phi_i(x_{t-i})+\sum_{j=1}^{r-1}\theta_j\Big(\sum_{i=1}^{r}\phi_i x_{t-j-i}+a_{t-j}\Big) && (6.25)\\
&= a_t+\sum_{i=1}^{r}\phi_i(x_{t-i})+\sum_{j=1}^{r-1}\theta_j x_{t-k} && (6.26)\\
&= x_t+\sum_{j=1}^{r-1}\theta_j x_{t-k}\theta_k x_{t-k} && (6.27)\\
&= z_t. && (6.28)
\end{aligned}$$

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

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

Principal Component Analysis

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

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

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

$$\frac{\sigma_i^2}{\sum_{j=1}^{k} \sigma_j^2} \tag{7.1}$$

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

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

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

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

You can load this data using the following code.

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

Write a function that performs PCA on the breast cancer dataset. Graph the first two principal components, with the first along the x-axis. Your graph should resemble Figure 7.1 below. Include in the graph title the amount of variance captured by the first two principal components, calculated with Equation 7.1.

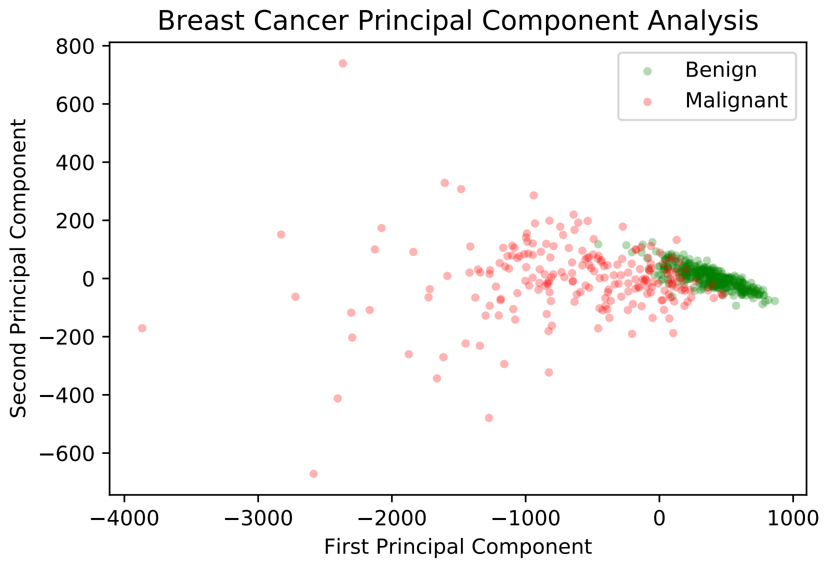


Figure 7.1: First two principal components of the transformed breast cancer data

Latent Semantic Indexing

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

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

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

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

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

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

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

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

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

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

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

Application: State of the Union

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We then let

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

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

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

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

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

Scikit-Learn

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

Transformers

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

Note

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

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

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

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

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

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

Validation Tools

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

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

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

With this terminology, we define the following metrics.

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

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

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

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

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

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

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

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

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

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

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

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

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

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

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

Grid Search

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

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

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

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

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

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

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

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

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

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

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

Pipelines

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8 Random Forests

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

Classification Trees

Classification trees are a class of decision trees used in a wide variety of settings where labeled training data is available. The desired outcome is a model that can accurately assign labels to unlabeled data.

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

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

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

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

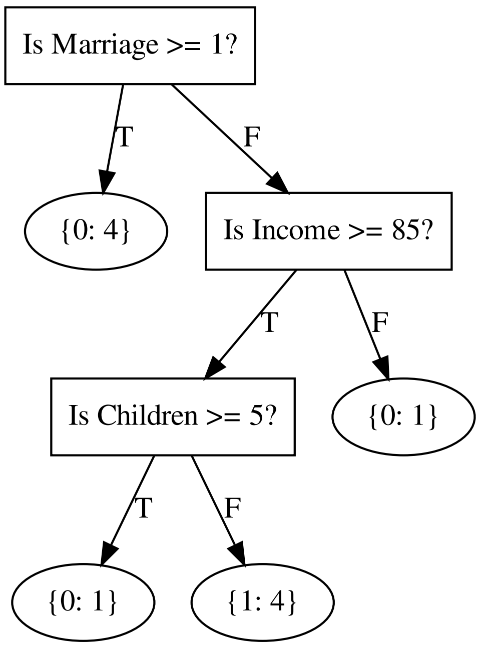


Figure 8.1: A classification tree built using Table 8.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

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

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

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

Next, write a function that splits a data set for a given question. Return the left and right regions of the partition in that order. If one region is empty, return it as `None`.

To decide on the best split, we need a few definitions.

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

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

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

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

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

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

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

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

# Test your functions
>>> gini(animals)
0.4758
>>> info_gain(animals[:50], animals[50:], gini(animals))
0.14579999999999999
```

The optimal split of a data set is the split that provides the greatest information gain. That is, the optimal split is

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

where

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

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

Problem 3. Write a function that computes the optimal split of a data set. Include a minimum leaf argument defaulting to 5. Do not allow the best split to include a leaf smaller than this size. Return the information gain and question associated with the best split. The output for the animals data set should be `(0.12259833679833688, Is # legs/tentacles >= 2.0?)`.

Next, let's build the framework for a full classification tree.

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

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

In addition to having a minimum leaf size, it's also important to have a maximum depth for trees. This helps to prevent the tree from overfitting to training data.

Problem 5. Write a function that uses your previous functions to build a classification tree. Include a minimum leaf argument defaulting to 5 and a maximum depth argument defaulting to 4. Start counting depth at 0. For comparison, the tree in Figure 8.1 has depth 3.

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

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

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

Note

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

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

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

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

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

Random Forest

A *random forest* is just what it sounds like–a collection of trees. Each tree is trained randomly, meaning that at each node, only a small, random subset of the features is available by which to determine the next split. The size of this subset should be small relative to the total number of features present. Let n be the total number of features in the data set. One common method, and the one we will use here, is to split on $\sqrt{n}$ features, rounding down where applicable.

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

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

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

Finally, write a method that accepts a labeled data set and a trained forest and analyzes the accuracy of the forest's predictions.

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

Scikit-Learn

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

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

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

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

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

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

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

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

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

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

Return the accuracies and times of each of these three forests.

A Getting Started

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

Submitting Assignments

Labs

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

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

Achtung!

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

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

Homework

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

Setup

Achtung!

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

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

Setup With Git

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7. (Optional) *Clone your repository.* If you want your repository on another computer after completing steps 1–4, use the following commands.

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

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

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

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

Setup Without Git

Even if you aren't using git to submit files, you must install it (`http://git-scm.com/downloads`) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 5 and 6 of the previous section to download the data files and install package dependencies.

Using Git

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

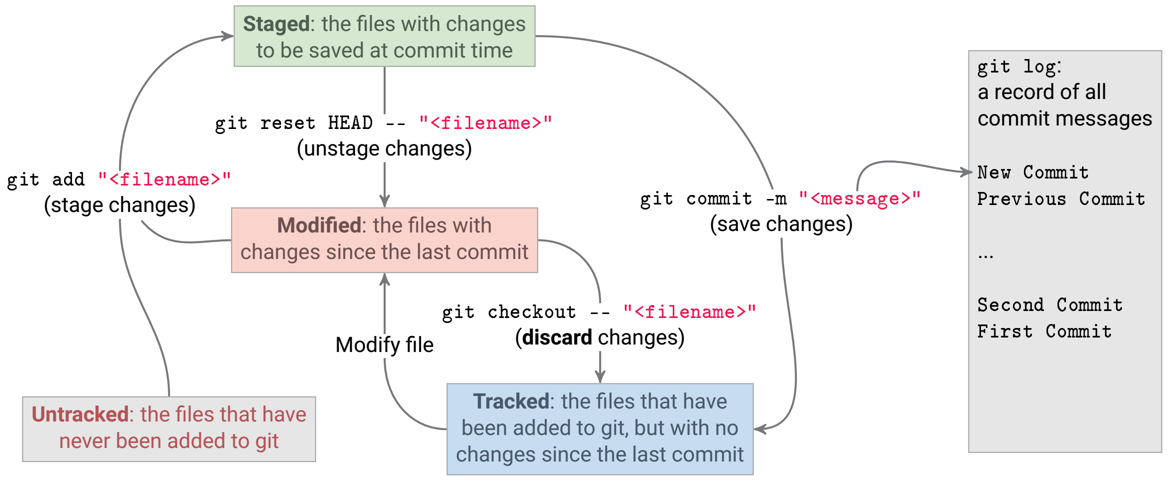

Staged: the files with changes to be saved at commit time
git reset HEAD -- "<filename>"
(unstage changes)
git add "<filename>"
(stage changes)
Modified: the files with changes since the last commit
git commit -m "<message>"
(save changes)
git checkout -- "<filename>"
(discard changes)
Modify file
Untracked: the files that have never been added to git
Tracked: the files that have been added to git, but with no changes since the last commit
git log:
a record of all commit messages
New Commit
Previous Commit
...
Second Commit
First Commit

Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

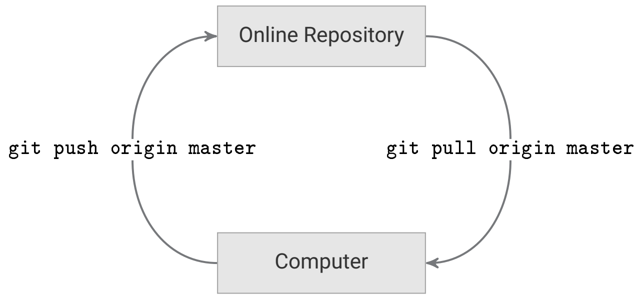


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Table A.1: Common git commands.

Note

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in *vim* (`https://en.wikipedia.org/wiki/Vim_(text_editor)`), is asking you to enter a message (or use the default message) to create a *merge commit* that will reconcile both changes. To close this screen and create the merge commit, type `:wq` and press `enter`.

Example Work Sessions

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master                                    # Pull updates.
### Make changes to a file.
$ git add -u                                                # Track changes.
$ git commit -m "Made some changes."                        # Commit changes.
$ git push origin master                                    # Push updates.
```

```
# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ git pull origin master
From https://bitbucket.org/username/repo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

### Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
 1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://username@bitbucket.org/username/repo.git
   5742a1b..fed9b34  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

B Installing and Managing Python

Lab Objective: *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to `https://www.anaconda.com/download/`.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see `https://docs.anaconda.com/anaconda/install/`. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

Achtung!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see `https://xkcd.com/349/`).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See `https://docs.anaconda.com/anaconda/packages/pkg-docs` for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
`conda install <package-name>`	Install the specified package.
`conda update <package-name>`	Update the specified package.
`conda update conda`	Update `conda` itself.
`conda update anaconda`	Update **all** packages included in Anaconda.
`conda --help`	Display the documentation for `conda`.

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda                # Make sure that conda is up to date.
$ conda install matplotlib          # Attempt to install matplotlib.
$ conda update matplotlib           # Attempt to update matplotlib.
```

See `https://conda.io/docs/user-guide/tasks/manage-pkgs.html` for more examples.

Note

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

Achtung!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
`pip install package-name`	Install the specified package.
`pip install --upgrade package-name`	Update the specified package.
`pip freeze`	Display the version number on all installed packages.
`pip --help`	Display the documentation for `pip`.

See `https://pip.pypa.io/en/stable/user_guide/` for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: `https://atom.io/`
- Sublime Text: `https://www.sublimetext.com/`
- Notepad++ (Windows): `https://notepad-plus-plus.org/`
- Geany: `https://www.geany.org/`
- Vim: `https://www.vim.org/`
- Emacs: `https://www.gnu.org/software/emacs/`

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                        # List the files in the current directory.
hello_world.py
$ cat hello_world.py        # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py     # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

NOTE

While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:

- Windows subsystem for linux: `docs.microsoft.com/en-us/windows/wsl/`.
- Git bash: `https://gitforwindows.org/`.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See `https://github.com/jupyter/jupyter/wiki/` for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Juptyer Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and LaTeX, and can embedded images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: `http://jupyterlab.readthedocs.io/en/stable/`
- PyCharm: `https://www.jetbrains.com/pycharm/`
- Spyder: `http://code.google.com/p/spyderlib/`
- Eclipse with PyDev: `http://www.eclipse.org/`, `https://www.pydev.org/`

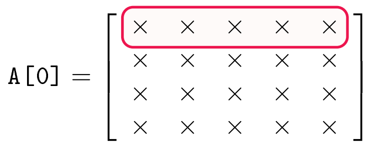
See `https://realpython.com/python-ides-code-editors-guide/` for a good overview of these (and other) workflow tools.

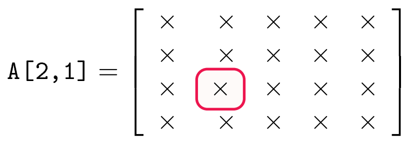
C NumPy Visual Guide

Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

Data Access

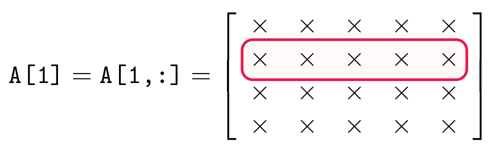
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

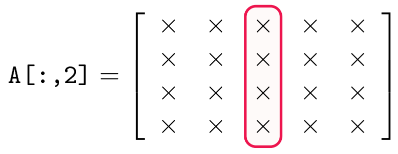


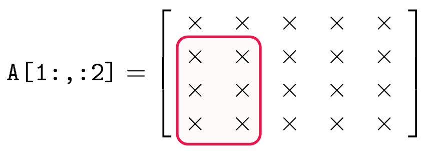


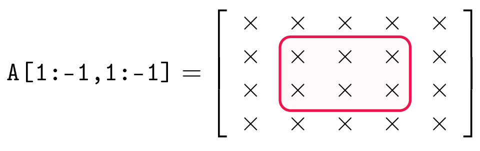
Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."









Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$\texttt{A} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad \texttt{B} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((A,B,A))} = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((A,B,A))} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$\texttt{x} = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad \texttt{y} = \begin{bmatrix} * & * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & * & * & * & * & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} \quad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{array}{c} \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \\ + \\ \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} \end{array} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} \boxed{\begin{matrix} 1 \\ 1 \\ 1 \\ 1 \end{matrix}} & \boxed{\begin{matrix} 2 \\ 2 \\ 2 \\ 2 \end{matrix}} & \boxed{\begin{matrix} 3 \\ 3 \\ 3 \\ 3 \end{matrix}} & \boxed{\begin{matrix} 4 \\ 4 \\ 4 \\ 4 \end{matrix}} \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} \boxed{\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}} \\ \boxed{\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}} \\ \boxed{\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}} \\ \boxed{\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}} \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

D Introduction to Scikit-Learn

Lab Objective: *Scikit-learn is the one of the fundamental tools in Python for machine learning. In this appendix we highlight and give examples of some popular scikit-learn tools for classification and regression, training and testing, data normalization, and constructing complex models.*

NOTE

This guide corresponds to scikit-learn version 0.20, which has a few significant differences from previous releases. See `http://scikit-learn.org/stable/whats_new.html` for current release notes. Install scikit-learn (the `sklearn` module) with `conda install scikit-learn`.

Base Classes and API

Many machine learning problems center on constructing a function $f : X \to Y$, called a *model* or *estimator*, that accurately represents properties of given data. The domain X is usually $\mathbb{R}^D$, and the range Y is typically either $\mathbb{R}$ (regression) or a subset of $\mathbb{Z}$ (classification). The model is trained on N *samples* $(\mathbf{x}_i)_{i=1}^N \subset X$ that usually (but not always) have N accompanying *labels* $(y_i)_{i=1}^N \subset Y$.

Scikit-learn [PVG+11, BLB+13] takes a highly object-oriented approach to machine learning models. Every major scikit-learn class inherits from `sklearn.base.BaseEstimator` and conforms to the following conventions:

1. The constructor `__init__()` receives *hyperparameters* for the classifier, which are parameters for the model f that are **not dependent on data**. Each hyperparameter must have a default value (i.e., every argument of `__init__()` is a keyword argument), and each argument must be saved as an instance variable of the **same name** as the parameter.

2. The `fit()` method constructs the model f. It receives an $N \times D$ matrix X and, optionally, a vector $\mathbf{y}$ with N entries. Each row $\mathbf{x}_i$ of X is one sample with corresponding label y_i. By convention, `fit()` always returns `self`.

Along with the `BaseEstimator` class, there are several other "mix in" base classes in `sklearn.base` that define specific kinds of models. The three listed below are the most common.[1]

[1]See `http://scikit-learn.org/stable/modules/classes.html#base-classes` for the complete list.

- `ClassifierMixin`: for *classifiers*, estimators that take on discrete values.
- `RegressorMixin`: for *regressors*, estimators that take on continuous values.
- `TransformerMixin`: for preprocessing data before estimation.

Classifiers and Regressors

The `ClassifierMixin` and `RegressorMixin` both require a `predict()` method that acts as the actual model f. That is, `predict()` receives an $N \times D$ matrix X and returns N predicted labels $(y_i)_{i=1}^N$, where y_i is the label corresponding to the ith row of X. Both of these base class have a predefined `score()` method that uses `predict()` to test the accuracy of the model. It accepts $N \times D$ test data and a vector of N corresponding labels, then reports either the classification accuracy (for classifiers) or the R^2 value of the regression (for regressors).

For example, a `KNeighborsClassifier` from `sklearn.neighbors` inherits from `BaseEstimator` and `ClassifierMixin`. This classifier uses a simple strategy: to classify a new piece of data $\mathbf{z}$, find the k training samples that are "nearest" to $\mathbf{z}$, then take the most common label corresponding to those nearest neighbors to be the label for $\mathbf{z}$. Its constructor accepts hyperparameters such as `n_neighbors`, for determining the number of neighbors k to search for, `algorithm`, which speficies the strategy to find the neighbors, and `n_jobs`, the number of parallel jobs to run during the neighbors search. Again, these hyperparameters are independent of any data, which is why they are set in the constructor (before fitting the model). Calling `fit()` organizes the data X into a data structure for efficient nearest neighbor searches (determined by `algorithm`). Calling `predict()` executes the search, determines the most common label of the neighbors, and returns that label.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.model_selection import train_test_split

# Load the breast cancer dataset and split it into training and testing groups.
>>> cancer = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(cancer.data,
...                                                     cancer.target)
>>> print(X_train.shape, y_train.shape)
(426, 30) (426,)        # There are 426 training points, each with 30 features.

# Train a KNeighborsClassifier object on the training data.
# fit() returns the object, so we can instantiate and train in a single line.
>>> knn = KNeighborsClassifier(n_neighbors=2).fit(X_train, y_train)
# The hyperparameter 'n_neighbors' is saved as an attribute of the same name.
>>> knn.n_neighbors
2

# Test the classifier on the testing data.
>>> knn.predict(X_test[:6])
array([0, 1, 0, 1, 1, 0])       # Predicted labels for the first 6 test points.
>>> knn.score(X_test, y_test)
0.8951048951048951              # predict() chooses 89.51% of the labels right.
```

The KNeighborsClassifier object could easily be replaced with a different classifier, such as a GaussianNB object from sklearn.naive_bayes. Since GaussianNB also inherits from BaseEstimator and ClassifierMixin, it has fit(), predict(), and score() methods that take in the same kinds of inputs as the corresponding methods for the KNeighborsClassifier. The only difference, from an external perspective, is the hyperparameters that the constructor accepts.

```
>>> from sklearn.naive_bayes import GaussianNB

>>> gnb = GaussianNB().fit(X_train, y_train)
>>> gnb.predict(X_test[:6])
array([1, 1, 0, 1, 1, 0])
>>> gnb.score(X_test, y_test)
0.9440559440559441
```

Roughly speaking, the GaussianNB classifier assumes all features in the data are independent and normally distributed, then uses Bayes' rule to compute the likelihood of a new point belonging to a label for each of the possible labels. To do this, the fit() method computes the mean and variance of each feature, grouped by label. These quantities are saved as the attributes theta_ (the means) and sigma_ (the variances), then used in predict(). Parameters like these that **are dependent on data** are only defined in fit(), not the constructor, and they are always named with a trailing underscore. These "non-hyper" parameters are often simply called *model parameters*.

```
>>> gnb.classes_        # The collection of distinct training labels.
array([0, 1])
>>> gnb.theta_[:,0]     # The means of the first feature, grouped by label.
array([17.55785276, 12.0354981 ])
# The samples with label 0 have a mean of 17.56 in the first feature.
```

The fit() method should do all of the "heavy lifting" by calculating the model parameters. The predict() method should then use these parameters to choose a label for test data.

	Hyperparameters	Model Parameters
Data dependence	No	Yes
Initialization location	`__init__()`	`fit()`
Naming convention	Same as argument name	Ends with an underscore
Examples	`n_neighbors`, `algorithm`, `n_jobs`	`classes_`, `theta_`, `sigma_`

Table D.1: Naming and initialization conventions for scikit-learn model parameters.

Building Custom Estimators

The consistent conventions in the various scikit-learn classes makes it easy to use a wide variety of estimators with near-identical syntax. These conventions also makes it possible to write custom estimators that behave like native scikit-learn objects. This usually only involves writing fit() and predict() methods and inheriting from the appropriate base classes. As a simple (though poorly performing) example, consider an estimator that either always predicts the same user-provided label, or that always predicts the most common label in the training data. Which strategy to use is independent of the data, so we encode that behavior with hyperparameters; the most common label must be calculated from the data, so that is a model parameter.

```
>>> import numpy as np
>>> from collections import Counter
>>> from sklearn.base import BaseEstimator, ClassifierMixin

>>> class PopularClassifier(BaseEstimator, ClassifierMixin):
...     """Classifier that always guesses the most common training label."""
...     def __init__(self, strategy="most_frequent", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         """Find and store the most common label."""
...         self.popular_label_ = Counter(y).most_common(1)[0][0]
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always guess the most popular training label."""
...         M = X.shape[0]
...         if self.strategy == "most_frequent":
...             return np.full(M, self.popular_label_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train a PopularClassifier on the breast cancer training data.
>>> pc = PopularClassifier().fit(X_train, y_train)
>>> pc.popular_label_
1
# Score the model on the testing data.
>>> pc.score(X_test, y_test)
0.6573426573426573                  # 65.73% of the testing data is labeled 1.

# Change the strategy to always guess 0 by changing the hyperparameters.
>>> pc.strategy = "constant"
>>> pc.constant = 0
>>> pc.score(X_test, y_test)
0.34265734265734266                  # 34.27% of the testing data is labeled 0.
```

This is a terrible classifier, but it is actually implemented as `sklearn.dummy.DummyClassifier` because any legitimate machine learning algorithm should be able to beat it, so it is useful as a baseline comparison.

Note that `score()` was inherited from `ClassifierMixin` (it isn't defined explicitly), so it returns a classification rate. In the next example, a slight simplification of the equally unintelligent `sklearn.dummy.DummyRegressor`, the `score()` method is inherited from `RegressorMixin`, so it returns an R^2 value.

```
>>> from sklearn.base import RegressorMixin

>>> class ConstRegressor(BaseEstimator, RegressorMixin):
...     """Regressor that always predicts a mean or median of training data."""
...     def __init__(self, strategy="mean", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         self.mean_, self.median_ = np.mean(y), np.median(y)
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always predict the middle of the training data."""
...         M = X.shape[0]
...         if self.strategy == "mean":
...             return np.full(M, self.mean_)
...         elif self.strategy == "median":
...             return np.full(M, self.median_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train on the breast cancer data (treating it as a regression problem).
>>> cr = ConstRegressor(strategy="mean").fit(X_train, y_train)
>>> print("mean:", cr.mean_, " median:", cr.median_)
mean: 0.6173708920187794  median: 1.0

# Get the R^2 score of the regression on the testing data.
>>> cr.score(X_train, y_train)
0                           # Unsurprisingly, no correlation.
```

Achtung!

Both `PopularClassifier` and `ConstRegressor` wait until `predict()` to validate the `strategy` hyperparameter. The check could easily be done in the constructor, but that goes against scikit-learn conventions: in order to cooperate with automated validation tools, the constructor of any class inheriting from `BaseEstimator` must store the arguments of `__init__()` as attributes—with the same names as the arguments—and do nothing else.

Note

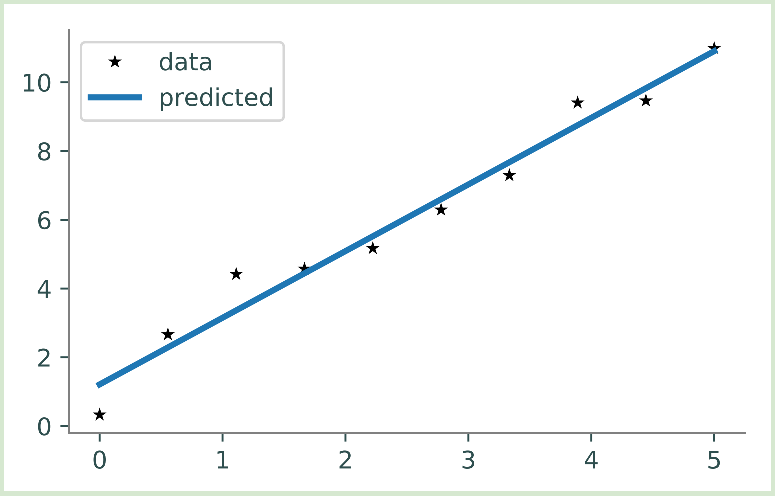
The first input to `fit()` and `predict()` are **always** two-dimensional $N \times D$ NumPy arrays, where N is the number of observations and D is the number of features. To fit or predict on one-dimensional data ($D = 1$), reshape the input array into a "column vector" before feeding it into the estimator. One-dimensional problems are somewhat rare in machine learning, but the following example shows how to do a simple one-dimensional linear regression.

```python
>>> from matplotlib import pyplot as plt
>>> from sklearn.linear_model import LinearRegression

# Generate data for a 1-dimensional regression problem.
>>> X = np.linspace(0, 5, 10)
>>> Y = 2*X + 1 + np.random.normal(size=10)

# Reshape the training data into a column vector.
>>> lr = LinearRegression().fit(X.reshape((-1,1)), Y)

# Define another set of points to do predictions on.
>>> x = np.linspace(0, 5, 20)
>>> y = lr.predict(x.reshape((-1,1)))   # Reshape before predicting.
>>> plt.plot(X, Y, 'k*', label="data")
>>> plt.plot(x, y, label="predicted")
>>> plt.legend(loc="upper left")
>>> plt.show()
```



Transformers

A scikit-learn *transformer* processes data to make it better suited for estimation. This may involve shifting and scaling data, dropping columns, replacing missing values, and so on.

Classes that inherit from the `TransformerMixin` base class have a `fit()` method that accepts an $N \times D$ matrix X (like an estimator) and an optional set of labels. The labels are not needed—in fact the `fit()` method should do nothing with them—but the parameter for the labels remains as a keyword argument to be consistent with the `fit(X,y)` syntax of estimators. Instead of a `predict()` method, the `transform()` method accepts data, modifies it (usually via a copy), and returns the result. The new data may or may not have the same number of columns as the original data.

One common transformation is shifting and scaling the features (columns) so that they each have a mean of 0 and a standard deviation of 1. The following example implements a basic version of this transformer.

```
>>> from sklearn.base import TransformerMixin

>>> class NormalizingTransformer(BaseEstimator, TransformerMixin):
...     def fit(self, X, y=None):
...         """Calculate the mean and standard deviation of each column."""
...         self.mu_ = np.mean(X, axis=0)
...         self.sig_ = np.std(X, axis=0)
...         return self
...
...     def transform(self, X):
...         """Center each column at zero and normalize it."""
...         return (X - self.mu_) / self.sig_
...
# Fit the transformer and transform the cancer data (both train and test).
>>> nt = NormalizingTransformer()
>>> Z_train = nt.fit_transform(X_train) # Or nt.fit(X_train).transform(X_train)
>>> Z_test = nt.transform(X_test)       # Transform test data (without fitting)

>>> np.mean(Z_train, axis=0)[:3]        # The columns of Z_train have mean 0...
array([-8.08951237e-16, -1.72006384e-17,  1.78678147e-15])
>>> np.std(Z_train, axis=0)[:3]         # ...and have unit variance.
array([1., 1., 1.])
>>> np.mean(Z_test, axis=0)[:3]         # The columns of Z_test each have mean
array([-0.02355067,  0.11665332, -0.03996177])                  # close to 0...
>>> np.std(Z_test, axis=0)[:3]          # ...and have close to unit deviation.
array([0.9263711 , 1.18461151, 0.91548103])

# Check to see if the classification improved.
>>> knn.fit(X_train, y_train).score(X_test, y_test)             # Old score.
0.8951048951048951
>>> knn.fit(Z_train, y_train).score(Z_test, y_test)             # New score.
0.958041958041958
```

This particular transformer is implemented as `sklearn.preprocessing.StandardScaler`. A close cousin is `sklearn.preprocessing.RobustScaler`, which ignores outliers when choosing the scaling and shifting factors.

Like estimators, transformers may have both hyperparameters (provided to the constructor) and model parameters (determined by `fit()`). Thus a transformer looks and acts like an estimator, with the exception of the `predict()` and `transform()` methods.

Achtung!

The `transform()` method should only rely on model parameters derived from the training data in `fit()`, **not** on the data that is worked on in `transform()`. For example, if the `NormalizingTransformer` is fit with the input $\widehat{X}$, then `transform()` should shift and scale any input X by the mean and standard deviation of $\widehat{X}$, not by the mean and standard deviation of X. Otherwise, the transformation is different for each input X.

Scikit-learn Module	**Classifier** Name	Notable Hyperparameters
`discriminant_analysis`	`LinearDiscriminantAnalysis`	`solver`, `shrinkage`, `n_components`
`discriminant_analysis`	`QuadraticDiscriminantAnalysis`	`reg_param`
`ensemble`	`AdaBoostClassifier`	`n_estimators`, `learning_rate`
`ensemble`	`RandomForestClassifier`	`n_estimators`, `max_depth`
`linear_model`	`LogisticRegression`	`penalty`, `C`
`linear_model`	`SGDClassifier`	`loss`, `penalty`, `alpha`
`naive_bayes`	`GaussianNB`	`priors`
`naive_bayes`	`MultinomialNB`	`alpha`
`neighbors`	`KNeighborsClassifier`	`n_neighbors`, `weights`
`neighbors`	`RadiusNeighborsClassifier`	`radius`, `weights`
`neural_network`	`MLPClassifier`	`hidden_layer_size`, `activation`
`svm`	`SVC`	`C`, `kernel`
`tree`	`DecisionTreeClassifier`	`max_depth`

Scikit-learn Module	**Regressor** Name	Notable Hyperparameters
`ensemble`	`AdaBoostRegressor`	`n_estimators`, `learning_rate`
`ensemble`	`ExtraTreesRegressor`	`n_estimators`, `max_depth`
`ensemble`	`GradientBoostingRegressor`	`n_estimators`, `max_depth`
`ensemble`	`RandomForestRegressor`	`n_estimators`, `max_depth`
`isotonic`	`IsotonicRegression`	`y_min`, `y_max`
`kernel_ridge`	`KernelRidge`	`alpha`, `kernel`
`linear_model`	`LinearRegression`	`fit_intercept`
`neural_network`	`MLPRegressor`	`hidden_layer_size`, `activation`
`svm`	`SVR`	`C`, `kernel`
`tree`	`DecisionTreeRegressor`	`max_depth`

Module	**Transformer** Name	Notable Hyperparameters
`decomposition`	`PCA`	`n_components`
`preprocessing`	`Imputer`	`missing_values`, `strategy`
`preprocessing`	`MinMaxScaler`	`feature_range`
`preprocessing`	`OneHotEncoder`	`categorical_features`
`preprocessing`	`QuantileTransformer`	`n_quantiles`, `output_distribution`
`preprocessing`	`RobustScaler`	`with_centering`, `with_scaling`
`preprocessing`	`StandardScaler`	`with_mean`, `with_std`

Table D.2: Common scikit-learn classifiers, regressors, and transformers. For full documentation on these classes, see `http://scikit-learn.org/stable/modules/classes.html`.

Validation Tools

Knowing how to determine whether or not an estimator performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the range of the desired model is $Y = \{0, 1\}$.

Evaluation Metrics

The `score()` method of a scikit-learn estimator representing the model $f : X \to \{0, 1\}$ returns the *accuracy* of the model, which is the percent of labels that are predicted correctly. However, accuracy isn't always the best measure of success. Consider the *confusion matrix* for a classifier, the matrix where the (i, j)th entry is the number of observations with actual label i but that are classified as label j. In binary classification, calling the class with label 0 the *negatives* and the class with label 1 the *positives*, this becomes the following.

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

With this terminology, we define the following metrics.

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

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

- F_β *Score*: $(1 + \beta^2)\dfrac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \dfrac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2 FN}$.

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

Scikit-learn implements these metrics in `sklearn.metrics`, as well as functions for evaluating regression, non-binary classification, and clustering models. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. For the complete list and further discussion, see `http://scikit-learn.org/stable/modules/model_evaluation.html`.

```
>>> from sklearn.metrics import (confusion_matrix, classification_report,
...                              accuracy_score, precision_score,
...                              recall_score, f1_score)

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

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

# Get accuracy (the "usual" score), precision, recall, and f1 scores.
>>> accuracy_score(y_test, knn_predicted)   # (CM[0,0] + CM[1,1]) / CM.sum()
0.8951048951048951
>>> precision_score(y_test, knn_predicted)  # CM[1,1] / CM[:,1].sum()
0.9438202247191011
>>> recall_score(y_test, knn_predicted)     # CM[1,1] / CM[1,:].sum()
0.8936170212765957
>>> f1_score(y_test, knn_predicted)
0.9180327868852459

# Get all of these scores at once with classification_report().
>>> print(classification_report(y_test, knn_predicted))
            precision    recall  f1-score   support

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

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

Cross Validation

The `sklearn.model_selection` module has utilities to streamline and improve model evaluation.

- `train_test_split()` randomly splits data into training and testing sets (we already used this).
- `cross_val_score()` randomly splits the data and trains and scores the model a set number of times. Each trial uses different training data and results in a different model. The function returns the score of each trial.
- `cross_validate()` does the same thing as `cross_val_score()`, but it also reports the time it took to fit, the time it took to score, and the scores for the test set as well as the training set.

Doing multiple evaluations with different testing and training sets is extremely important. If the scores on a cross validation test vary wildly, the model is likely overfitting to the training data.

```python
>>> from sklearn.model_selection import cross_val_score, cross_validate

# Make (but do not train) a classifier to test.
>>> knn = KNeighborsClassifier(n_neighbors=3)

# Test the classifier on the training data 4 times.
>>> cross_val_score(knn, X_train, y_train, cv=4)
array([0.88811189, 0.92957746, 0.96478873, 0.92253521])

# Get more details on the train/test procedure.
>>> cross_validate(knn, X_train, y_train, cv=4,
...                return_train_score=False)
{'fit_time': array([0.00064683, 0.00042295, 0.00040913, 0.00040436]),
 'score_time': array([0.00115728, 0.00109601, 0.00105286, 0.00102782]),
 'test_score': array([0.88811189, 0.92957746, 0.96478873, 0.92253521])}

# Do the scoring with an alternative metric.
>>> cross_val_score(knn, X_train, y_train, scoring="f1", cv=4)
array([0.93048128, 0.95652174, 0.96629213, 0.93103448])
```

Note

Any estimator, even a user-defined class, can be evaluated with the scikit-learn tools presented in this section as long as that class conforms to the scikit-learn API discussed previously (i.e., inheriting from the correct base classes, having `fit()` and `predict()` methods, managing hyperparameters and parameters correctly, and so on). Any time you define a custom estimator, following the scikit-learn API gives you instant access to tools such as `cross_val_score()`.

Grid Search

Recall that the *hyperparameters* of a machine learning model are user-provided parameters that do not depend on the training data. Finding the optimal hyperparameters for a given model is a challenging and active area of research.[2] However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with an estimator, a dictionary of hyperparameters, and cross validation parameters (such as `cv` and `scoring`). When its `fit()` method is called, it does a cross validation test on the given estimator with every possible hyperparameter combination.

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

[2]Intelligent hyperparameter selection is sometimes called *metalearning*. See, for example, [SGCP+18].

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

>>> knn = KNeighborsClassifier()
# Specify the hyperparameters to vary and the possible values they should take.
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...               "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, cv=4, scoring="f1", verbose=1)
>>> knn_gs.fit(X_train, y_train)
Fitting 4 folds for each of 5 candidates, totalling 20 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent worker.
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.1s finished

# After fitting, the gridsearch object has data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_)
{'n_neighbors': 5, 'weights': 'uniform'} 0.9532526583188765
```

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

In some circumstances, the parameter grid can be also organized in a way that eliminates redundancy. Consider an `SVC` classifier from `sklearn.svm`, an estimator that works by lifting the data into a high-dimensional space, then constructing a hyperplane to separate the classes. The `SVC` has a hyperparameter, `kernel`, that determines how the lifting into higher dimensions is done, and for each choice of kernel there are additional corresponding hyperparameters. To search the total hyperparameter space without redundancies, enter the parameter grid as a list of dictionaries, each of which defines a different section of the hyperparameter space. In the following code, doing so reduces the number of trials from $3 \times 2 \times 3 \times 4 = 72$ to only $1 + (1 \times 1 \times 3) + (1 \times 4) = 11$.

```
>>> from sklearn.svm import SVC

>>> svc = SVC(C=0.01, max_iter=100)
>>> param_grid = [
...     {"kernel": ["linear"]},
...     {"kernel": ["poly"], "degree": [2,3], "coef0": [0,1,5]},
...     {"kernel": ["rbf"], "gamma": [.01, .1, 1, 100]}]
>>> svc_gs = GridSearchCV(svc, param_grid,
                          cv=4, scoring="f1",
                          verbose=1, n_jobs=-1).fit(X_train, y_train)
Fitting 4 folds for each of 11 candidates, totalling 44 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  44 out of  44 | elapsed:    2.4s finished

>>> print(svc_gs.best_params_, svc_gs.best_score_)
{'gamma': 0.01, 'kernel': 'rbf'} 0.8909310239174055
```

See `https://scikit-learn.org/stable/modules/grid_search.html` for more details about `GridSearchCV` and its relatives.

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* (`sklearn.pipeline.Pipeline`) chains together one or more transformers and one estimator into a single object, complete with `fit()` and `predict()` methods. For example, it is often a good idea to shift and scale data before feeding it into a classifier. The `StandardScaler` transformer can be combined with a classifier with a pipeline. Calling `fit()` on the resulting object calls `fit_transform()` on each successive transformer, then `fit()` on the estimator at the end. Likewise, calling `predict()` on the `Pipeline` object calls `transform()` on each transformer, then `predict()` on the estimator.

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

# Chain together a scaler transformer and a KNN estimator.
>>> pipe = Pipeline([("scaler", StandardScaler()),      # "scaler" is a label.
                     ("knn", KNeighborsClassifier())])  # "knn" is a label.
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972                       # Already an improvement!
```

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

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

# Pass the Pipeline object to the GridSearchCV and fit it to the data.
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("knn", KNeighborsClassifier())])
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=4, n_jobs=-1, verbose=1).fit(X_train, y_train)
Fitting 4 folds for each of 40 candidates, totalling 160 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 160 out of 160 | elapsed:    0.3s finished

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

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

```
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("classifier", KNeighborsClassifier())])
>>> pipe_param_grid = [
...     {"classifier": [KNeighborsClassifier()],    # Try a KNN classifier...
...      "classifier__n_neighbors": [2,3,4,5],
...      "classifier__weights": ["uniform", "distance"]},
...     {"classifier": [SVC(kernel="rbf")],         # ...and an SVM classifier.
...      "classifier__C": [.001, .01, .1, 1, 10, 100],
...      "classifier__gamma": [.001, .01, .1, 1, 10, 100]}]
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=5, scoring="f1",
...                        verbose = 1, n_jobs=-1).fit(X_train, y_train)
Fitting 5 folds for each of 44 candidates, totalling 220 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 220 out of 220 | elapsed:    0.6s finished

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

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

Additional Material

Exercises

Problem 1. Writing custom scikit-learn transformers is a convenient way to organize the data cleaning process. Consider the data in `titanic.csv`, which contains information about passengers on the maiden voyage of the *RMS Titanic* in 1912. Write a custom transformer class to clean this data, implementing the `transform()` method as follows:

1. Extract a copy of data frame with just the `"Pclass"`, `"Sex"`, and `"Age"` columns.
2. Replace `NaN` values in the `"Age"` column (of the copied data frame) with the mean age. The mean age of the training data should be calculated in `fit()` and used in `transform()` (compare this step to using `sklearn.preprocessing.Imputer`).
3. Convert the `"Pclass"` column datatype to pandas categoricals (`pd.CategoricalIndex`).
4. Use `pd.get_dummies()` to convert the categorical columns to multiple binary columns (compare this step to using `sklearn.preprocessing.OneHotEncoder`).
5. Cast the result as a NumPy array and return it.

Ensure that your transformer matches scikit-learn conventions (it inherits from the correct base classes, `fit()` returns `self`, etc.).

Problem 2. Read the data from `titanic.csv` with `pd.read_csv()`. The `"Survived"` column indicates which passengers survived, so the entries of the column are the labels that we would like to predict. Drop any rows in the raw data that have `NaN` values in the `"Survived"` column, then separate the column from the rest of the data. Split the data and labels into training and testing sets. Use the training data to fit a transformer from Problem 1, then use that transformer to clean the training set, then the testing set. Finally, train a `LogisticRegressionClassifier` and a `RandomForestClassifier` on the cleaned training data, and score them using the cleaned test set.

Problem 3. Use `classification_report()` to score your classifiers from Problem 2. Next, do a grid search for each classifier (using only the cleaned training data), varying at least two hyperparameters for each kind of model. Use `classification_report()` to score the resulting best estimators with the cleaned test data. Try changing the hyperparameter spaces or scoring metrics so that each grid search yields a better estimator.

Problem 4. Make a pipeline with at least two transformers to further process the Titanic dataset. Do a gridsearch on the pipeline and report the hyperparameters of the best estimator.

Bibliography

[ADH$^+$01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[BLB$^+$13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[PVG$^+$11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[SGCP$^+$18] Brandon Schoenfeld, Christophe Giraud-Carrier, Mason Poggemann, Jarom Christensen, and Kevin Seppi. Preprocessor selection for machine learning pipelines. *arXiv preprint arXiv:1810.09942*, 2018.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.