9 Selected unsupervised learning algorithms

This chapter covers

- Latent Dirichlet allocation for topic discovery
- Density estimators in computational biology and finance
- Structure learning for relational data
- Simulated annealing for energy minimization
- Genetic algorithm in evolutionary biology
- ML research: unsupervised learning

In the previous chapter, we looked at unsupervised ML algorithms to help learn patterns in our data; this chapter continues that discussion, focusing on selected algorithms. The algorithms presented in this chapter have been included to cover the breadth of unsupervised learning, and they are important to learn because they cover a range of applications, from computational biology to physics to finance. We'll start by looking at latent Dirichlet allocation (LDA) for learning topic models, followed by density estimators and structure learning algorithms, and concluding with simulated annealing (SA) and genetic algorithms (GAs).

9.1 Latent Dirichlet allocation

A topic model is a latent variable model for discrete data, such as text documents. Latent dirichlet allocation (LDA) is a topic model that represents each document as a finite mixture of topics, where a topic is a distribution over words. The objective is to learn the shared topic distribution and topic proportions for each document. LDA assumes a bag of words model in which the words are exchangeable and, as a result, sentence structure is not preserved (i.e., only the word counts matter). Thus, each document is reduced to a vector of counts over the vocabulary V and the entire corpus of D documents is summarized in a *term-document* matrix A_{VD}. LDA can be seen as a nonnegative matrix factorization problem that takes the term-document matrix and factorizes it into a product of topics W_{VK} and topic proportions H_{KD}: $A = W_H$.

Term frequency–inverse document frequency (tf-idf) is a common method for adjusting the word counts, as it logarithmically drives down to zero word counts that occur frequently across documents: $A \log(D/n_t)$, where D is the total number of documents in the corpus and n_t is the number of documents where term t appears. The tf-idf smoothing of word counts identifies the sets of words that are discriminative for documents and leads to better model performance. The term-document matrix generalizes from counts of individual words (unigrams) to larger structural units, such as n-grams. In the case of n-grams, different techniques for smoothing word counts (e.g., Laplace smoothing) are used to address the lack of observations in a very large feature space. Figure 9.1 shows the graphical model for the LDA.

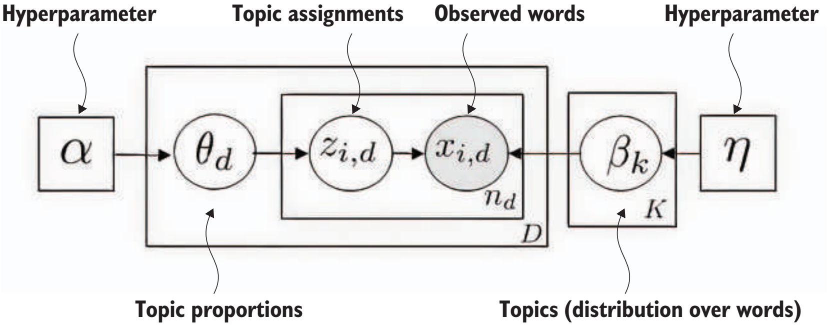

Figure 9.1 Latent Dirichlet allocation graphical model

The LDA topic model associates each word $x_{i,d}$ with a topic label $z_{i,d} \in \{1, 2, ..., K\}$. Each document is associated with topic proportions θ_d that could be used to measure document similarity. The topics β_k are shared across all documents. The hyperparameters α and η capture our prior knowledge of topic proportions and topics, respectively (e.g., from past online training of the model). The full generative model can be specified as follows.

$$\begin{aligned} \theta_d|\alpha &\sim \text{Dir}(\alpha) \\ z_{i,d}|\theta_d &\sim \text{Cat}(\theta_d) \\ \beta_k|\eta &\sim \text{Dir}(\eta) \\ x_{i,d}|z_{i,d}=k,\beta &\sim \text{Cat}(\beta_k) \end{aligned} \tag{9.1}$$

The joint distribution for a single document d can be written as follows (as discussed in David M. Blei, Andrew Y. Ng, and Michael I. Jordan's "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, 2003).

$$p(x,z,\theta,|\alpha,\beta) = p(\theta_d|\alpha)\prod_{i=1}^{n_d} p(z_{i,d}|\theta_d)p(x_{i,d}|z_{i,d},\beta) \tag{9.2}$$

The parameters α and β are corpus-level parameters, the variable θ_d is sampled once every document, and $z_{i,d}$ and $x_{i,d}$ are word-level variables sampled once for each word in each document. Unlike a multinomial clustering model, where each document is associated with a single topic, LDA represents each document as a mixture of topics.

9.1.1 Variational Bayes

The key inference problem we need to solve to use LDA is that of computing the posterior distribution of the latent variables for a given document: $p(\theta, z|x, \alpha, \beta)$. The posterior can be approximated with the following variational distribution.

$$q(\theta,z|\gamma,\phi) = q(\theta|\gamma)\prod_{i=1}^{n} q(z_i|\phi_i) \tag{9.3}$$

The variational parameters are optimized to maximize the evidence lower bound (ELBO). Recall from chapter 3 the definition of ELBO as a difference between the energy term and the entropy term.

$$\begin{aligned} \log p(x|\alpha,\eta) &\geq L(x,\phi,\gamma,\lambda) \\ &= E_q[\log p(x,z,\theta,\beta|\alpha,\eta)] - E_q[\log q(z,\theta,\beta)] \end{aligned} \tag{9.4}$$

We choose a fully factored distribution q of the form.

$$q(z_{id}=k) = \phi_{dwk}; \quad q(\theta_d) \sim \text{Dir}(\theta_d|\gamma_d); \quad q(\beta_k) \sim \text{Dir}(\beta_k|\lambda_k) \tag{9.5}$$

We can expand the lower bound by using the factorizations of p and q (following David M. Blei, Andrew Y. Ng, and Michael I. Jordan's "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, 2003) .

$$\begin{aligned} L(\gamma, \phi; \alpha, \beta) &= E_q[\log p(\theta|\alpha)] + E_q[\log p(z|\theta)] \\ &\quad + E_q[\log p(x|z, \beta)] \\ &\quad - E_q[\log q(\theta)] - E_q[\log q(z)] \end{aligned} \tag{9.6}$$

Each of the five terms in $L(\gamma, \phi; \alpha, \beta)$ can be expanded as follows.

$$\begin{aligned} L(\gamma, \phi; \alpha, \beta) &= \log\Gamma\left(\sum_{j=1}^{k}\alpha_j\right) - \sum_{i=1}^{k}\log\Gamma(\alpha_i) \\ &+ \sum_{i=1}^{k}(\alpha_i - 1)\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{k}\gamma_j\right)\right) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{k}\phi_{ni}\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{k}\gamma_j\right)\right) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{k}\sum_{j=1}^{V}\phi_{ni}x_n^j\log\beta_{ij} \\ &- \log\Gamma\left(\sum_{j=1}^{k}\gamma_j\right) + \sum_{i=1}^{k}\log\Gamma(\gamma_i) \\ &- \sum_{i=1}^{k}(\gamma_i - 1)\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{k}\gamma_j\right)\right) \\ &- \sum_{n=1}^{N}\sum_{i=1}^{k}\phi_{ni}\log\phi_{ni} \end{aligned} \tag{9.7}$$

Here, $\Psi(x) = d/dx \log \Gamma(x)$ is the digamma function. $L(\gamma, \phi; \alpha, \beta)$ can be maximized using coordinate ascent over the variational parameters ϕ, γ, α.

$$\begin{aligned} \phi_{dwk} &\propto \exp\{E_q[\log\theta_{dk}] + E_q[\log\beta_{kw}]\} \\ \gamma_{dk} &= \alpha + \sum_{w} n_{dw}\phi_{dwk} \\ \lambda_{kw} &= \eta + \sum_{d} n_{dw}\phi_{dwk} \end{aligned} \tag{9.8}$$

Here, the expectations under q of $\log \theta$ and $\log \beta$ are as follows.

$$E_q[\log \theta_{dk}] = \Psi(\gamma_{dk}) - \Psi\left(\sum_{i=1}^{K} \gamma_{di}\right)$$
$$E_q[\log \beta_{kw}] = \Psi(\lambda_{kw}) - \Psi\left(\sum_{i=1}^{W} \lambda_{ki}\right) \tag{9.9}$$

The variational parameter updates can be used in an online setting that does not require a full pass through the entire corpus at each iteration. An online update of variational parameters enables topic analysis for very large datasets, including streaming data. Online variational Bayes (VB) for LDA is described in the algorithm in figure 9.2.

```
Define $\rho_t = (\tau_0 + t)^{-\kappa}$
Initialize $\lambda$ randomly
for $t = 1$ to $\infty$ do
    E step:
    Initialize $\gamma_{tk} = 1$
    repeat
        Set $\phi_{twk} \propto \exp\{E_q[\log \theta_{tk}] + E_q[\log \beta_{kw}]\}$ ← Topic assignments
        Set $\gamma_{tk} = \alpha + \sum_w \phi_{twk} n_{tw}$ ← Topic proportions
    until $\frac{1}{K} \sum_k |\Delta \gamma_{tk}| < \epsilon$
    M step:
    Compute $\tilde{\lambda}_{kw} = \eta + D n_{tw} \phi_{twk}$
    Set $\lambda = (1 - \rho_t)\lambda + \rho_t \tilde{\lambda}$
end for
```

Figure 9.2 LDA algorithm pseudo-code

As the t-th vector of word counts n_t is observed, we perform an E step to find locally optimal values of γ_t and ϕ_t, holding λ fixed. We then compute the $\tilde{\lambda}$ that would be optimal if our entire corpus consisted of the single document n repeated D times. We then update λ as a weighted average of its previous value and $\tilde{\lambda}$, where the weight is given by the learning parameter ρ_t for $\kappa \in (0.5, 1]$, controlling the rate at which old values of $\tilde{\lambda}$ are forgotten. We are now ready to implement variational Bayes for LDA from scratch in the following listing!

Listing 9.1 Variational Bayes for latent Dirichlet allocation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from wordcloud import WordCloud
from scipy.special import digamma, gammaln
```

```
np.random.seed(12)

class LDA:
    def __init__(self, A, K):
        self.N = A.shape[0]
        self.D = A.shape[1]
        self.K = num_topics

        self.A = A

        #init word distribution beta
        self.eta = np.ones(self.N)
        self.beta = np.zeros((self.N, self.K))
        for k in range(self.K):
            self.beta[:,k] = np.random.dirichlet(self.eta)
            self.beta[:,k] = self.beta[:,k] + 1e-6 #to avoid zero entries
            self.beta[:,k] = self.beta[:,k]/np.sum(self.beta[:,k])
        #end for

        #init topic proportions theta and cluster assignments z
        self.alpha = np.ones(self.K)
        self.z = np.zeros((self.N, self.D))
        for d in range(self.D):
            theta = np.random.dirichlet(self.alpha)
            wdn_idx = np.nonzero(self.A[:,d])[0]
            for i in range(len(wdn_idx)):
                z_idx = np.argmax(np.random.multinomial(1, theta))
                self.z[wdn_idx[i],d] = z_idx  #topic id
            #end for
        #end for

        #init variational parameters
        self.gamma = np.ones((self.D, self.K))
        for d in range(self.D):
            theta = np.random.dirichlet(self.alpha)
            self.gamma[d,:] = theta
        #end for

        self.lmbda = np.transpose(self.beta)
        ➥ #np.ones((self.K, self.N))/self.N

        self.phi = np.zeros((self.D, self.N, self.K))
        for d in range(self.D):
            for w in range(self.N):
                theta = np.random.dirichlet(self.alpha)
                self.phi[d,w,:] = np.random.multinomial(1, theta)
            #end for
        #end for

    def variational_inference(self, var_iter=10):

        llh = np.zeros(var_iter)
        llh_delta = np.zeros(var_iter)
```

Word dictionary size

Number of documents

Number of topics

Term-document matrix

Uniform Dirichlet prior on words

NxK topic matrix

Uniform Dirichlet prior on topics

Cluster assignments z

Topic proportions

Word frequencies

Assignments

```
        for iter in range(var_iter):
            print("VI iter: ", iter)
            J_old = self.elbo_objective()
            self.mean_field_update()
            J_new = self.elbo_objective()

            llh[iter] = J_old
            llh_delta[iter] = J_new - J_old
        #end for

        #update alpha and beta
        for k in range(self.K):
            self.alpha[k] = np.sum(self.gamma[:,k])
            self.beta[:,k] = self.lmbda[k,:] / np.sum(self.lmbda[k,:])
        #end for

        #update topic assignments
        for d in range(self.D):
            wdn_idx = np.nonzero(self.A[:,d])[0]
            for i in range(len(wdn_idx)):
                z_idx = np.argmax(self.phi[d,wdn_idx[i],:])
                self.z[wdn_idx[i],d] = z_idx  #topic id
            #end for
        #end for

        plt.figure()
        plt.plot(llh); plt.title('LDA VI');
        plt.xlabel('mean field iterations'); plt.ylabel("ELBO")
        plt.show()

        return llh

    def mean_field_update(self):

        ndw = np.zeros((self.D, self.N))
        for d in range(self.D):
            doc = self.A[:,d]
            wdn_idx = np.nonzero(doc)[0]

            for i in range(len(wdn_idx)):
                ndw[d,wdn_idx[i]] += 1
            #end for

            #update gamma
            for k in range(self.K):
                self.gamma[d,k] = self.alpha[k] + np.dot(ndw[d,:],
                ➥ self.phi[d,:,k])
            #end for

            #update phi
            for w in range(len(wdn_idx)):
                self.phi[d,wdn_idx[w],:] = np.exp(digamma(self.gamma[d,:])
➥ - digamma(np.sum(self.gamma[d,:])) + digamma(self.lmbda[:,wdn_idx[w]])
➥ - digamma(np.sum(self.lmbda, axis=1)))
                if (np.sum(self.phi[d,wdn_idx[w],:]) > 0): #to avoid 0/0
```

Word counts for each document

```
                    self.phi[d,wdn_idx[w],:] = self.phi[d,wdn_idx[w],:] /
                    ➥ np.sum(self.phi[d,wdn_idx[w],:]) #normalize phi
                #end if
            #end for

        #end for

        #update lambda given ndw for all docs
        for k in range(self.K):
            self.lmbda[k,:] = self.eta
            for d in range(self.D):
                self.lmbda[k,:] += np.multiply(ndw[d,:], self.phi[d,:,k])
            #end for
        #end for

    def elbo_objective(self):
        #see Blei 2003

        T1_A = gammaln(np.sum(self.alpha)) - np.sum(gammaln(self.alpha))
        T1_B = 0
        for k in range(self.K):
            T1_B +=  np.dot(self.alpha[k]-1, digamma(self.gamma[:,k]) -
            ➥ digamma(np.sum(self.gamma, axis=1)))
        T1 = T1_A + T1_B

        T2 = 0
        for n in range(self.N):
            for k in range(self.K):
                T2 += self.phi[:,n,k] * (digamma(self.gamma[:,k]) -
                ➥ digamma(np.sum(self.gamma, axis=1)))

        T3 = 0
        for n in range(self.N):
            for k in range(self.K):
                T3 += self.phi[:,n,k] * np.log(self.beta[n,k])

        T4 = 0
        T4_A = -gammaln(np.sum(self.gamma, axis=1)) +
        ➥ np.sum(gammaln(self.gamma), axis=1)
        T4_B = 0
        for k in range(self.K):
            T4_B = -(self.gamma[:,k]-1) * (digamma(self.gamma[:,k]) -
            ➥ digamma(np.sum(self.gamma, axis=1)))
        T4 = T4_A + T4_B

        T5 = 0
        for n in range(self.N):
            for k in range(self.K):
                T5 += -np.multiply(self.phi[:,n,k], np.log(self.phi[:,n,k]
                ➥ + 1e-6))

        T15 = T1 + T2 + T3 + T4 + T5
        J = sum(T15)/self.D  #averaged over documents
        return J
```

```
if __name__ == "__main__":

    #LDA parameters
    num_features = 1000  #vocabulary size
    num_topics = 4      #fixed for LD

    #20 newsgroups dataset
    categories = ['sci.crypt', 'comp.graphics', 'sci.space',
    ➥ 'talk.religion.misc']

    newsgroups = fetch_20newsgroups(shuffle=True, random_state=42,
    ➥ subset='train',
                 remove=('headers', 'footers', 'quotes'),
                 ➥ categories=categories)

    vectorizer = TfidfVectorizer(max_features = num_features, max_df=0.95,
    ➥ min_df=2, stop_words = 'english')
    dataset = vectorizer.fit_transform(newsgroups.data)
    A = np.transpose(dataset.toarray())  #term-document matrix

    lda = LDA(A=A, K=num_topics)
    llh = lda.variational_inference(var_iter=10)
    id2word = {v:k for k,v in vectorizer.vocabulary_.items()}

    #display topics
    for k in range(num_topics):
        print("topic: ", k)
        print("----------")
        topic_words = ""
        top_words = np.argsort(lda.lmbda[k,:])[-10:]
        for i in range(len(top_words)):
            topic_words += id2word[top_words[i]] + " "
            print(id2word[top_words[i]])

        wordcloud = WordCloud(width = 800, height = 800,
                    background_color ='white',
                    min_font_size = 10).generate(topic_words)

        plt.figure()
        plt.imshow(wordcloud)
        plt.axis("off")
        plt.tight_layout(pad = 0)
        plt.show()
```

Figure 9.3 shows the increase in ELBO as the number of mean-field iterations increases. Figure 9.4 shows the inferred topic distributions visualized as word clouds.

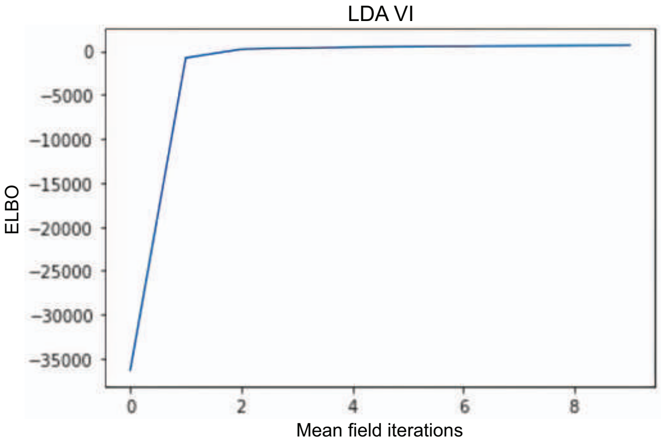

Figure 9.3 Increase in ELBO vs. the number of mean-field iterations

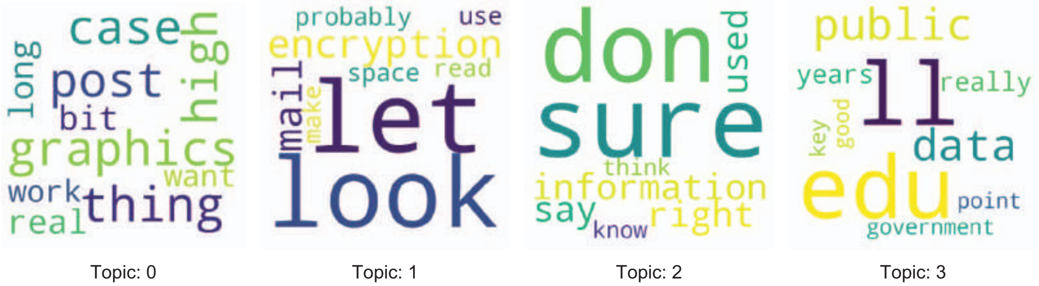

Figure 9.4 Inferred topic distributions via LDA mean-field variational inference

As we can see from the output, the top-K words for each topic match the categories in the 20 newsgroups dataset. In the following section, we will explore several methods to model the probability density of data for computational biology and finance.

9.2 Density estimators

The goal of *density estimation* is to model the probability density of data. In this section, we will discuss kernel density estimators applied to computational biology and look at how we can optimize a portfolio of stocks using tangent portfolio theory.

9.2.1 Kernel density estimator

An alternative approach to a K-component mixture model is a *kernel density estimator* (KDE) that allocates one cluster center per data point. KDEs are an application of kernel smoothing for probability density estimation that use kernels as weights. In the case of a Gaussian kernel, we have the following.

$$p(x|D) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{N}(x|x_i, \sigma^2 I) \tag{9.10}$$

Notice that we are averaging N Gaussians, with each Gaussian centered at the data point x_i. We can generalize the expression in equation 9.10 to any kernel $\kappa(x)$.

$$p(x|D) = \frac{1}{N}\sum_{i=1}^{N} \kappa_h(x - x_i) \tag{9.11}$$

The advantage of KDEs over parametric models, such as density mixtures, is that no model fitting is required (except for fine-tuning the bandwidth parameter h) and there is no need to pick the number of mixtures K. The disadvantage is that the model takes a lot of memory to store as well as time to evaluate. In other words, KDE is suitable when an accurate density estimate is required for a relatively small dataset (small number of points N). Let's look at an example that analyzes RNA-seq data to estimate the flux of a T7 promoter.

Listing 9.2 Kernel density estimate

```
import numpy as np
import matplotlib.pyplot as plt

class KDE():

    def __init__(self):
        #Histogram and Gaussian Kernel Estimator used to
        #analyze RNA-seq data for flux estimation of a T7 promoter
        self.G = 1e9
        self.C = 1e3
        self.L = 100
        self.N = 1e6
        self.M = 1e4
        self.LN = 1000
        self.FDR = 0.05

        #uniform sampling (poisson model)
        self.lmbda = (self.N * self.L) / self.G
        self.C_est = self.M/(1-np.exp(-self.lmbda))
        self.C_cvrg = self.G - self.G *
        ➥ np.exp(-self.lmbda)
        self.N_gaps = self.N * np.exp(-self.lmbda)

        #gamma prior sampling (negative binomial model)
        #X = "number of failures before rth success"
```

Length of genome in base pairs (bp)

Number of unique molecules

Length of a read, bp

Number of unique read sequences, bp

Number of reads, L bp long

Total length of assembled/mapped RNA-seq reads

False discovery rate

Expected number of bases covered

Library size estimate

Base coverage

Number of gaps (uncovered bases)

```
        self.k = 0.5                                     Dispersion parameter (fit to data)
        self.p = self.lmbda/(self.lmbda + 1/self.k)      Success probability
        self.r = 1/self.k                                Number of successes

        #RNAP binding data (RNA-seq)
        self.data = np.random.negative_binomial(self.r, self.p, size=self.LN)

    def histogram(self):
        self.bin_delta = 1                               Smoothing parameter
        self.bin_range = np.arange(1, np.max(self.data), self.bin_delta)
        self.bin_counts, _ = np.histogram(self.data, bins=self.bin_range)

        #histogram density estimation
        #P = integral_R p(x) dx, where X is in R^3
        #p(x) = K/(NxV), where K=number of points in region R
        #N=total number of points, V=volume of region R

        rnap_density_est = self.bin_counts/(sum(self.bin_counts) *
        ➥ self.bin_delta)
        return rnap_density_est

    def kernel(self):
        #Gaussian kernel density estimator with smoothing parameter h
        #sum N Guassians centered at each data point, parameterized by
        ➥ common std dev h

        x_dim = 1                                        Dimension of x
        h = 10                                           Standard deviation

        rnap_density_support = np.arange(np.max(self.data))
        rnap_density_est = 0
        for i in range(np.sum(self.bin_counts)):
            rnap_density_est += (1/(2*np.pi*h**2)**(x_dim/2.0))*np.exp(-
            ➥ (rnap_density_support - self.data[i])**2 / (2.0*h**2))
        #end for

        rnap_density_est = rnap_density_est / np.sum(rnap_density_est)
        return rnap_density_est

if __name__ == "__main__":

    kde = KDE()
    est1 = kde.histogram()
    est2 = kde.kernel()

    plt.figure()
    plt.plot(est1, c='b', label='histogram')
    plt.plot(est2, c='r', label='gaussian kernel')
    plt.title("RNA-seq density estimate based on negative binomial sampling
    ➥ model")
    plt.xlabel("read length, [base pairs]"); plt.ylabel("density");
    ➥ plt.legend()
    plt.show()
```

Figure 9.5 shows the RNA-seq density estimate based on the negative binomial model. It shows two density estimates—one based on a histogram and the other based on Gaussian Kernel Density Estimator. We can see that the Gaussian density is much smoother and that we can further fine-tune bin size smoothing parameter in the histogram estimator.

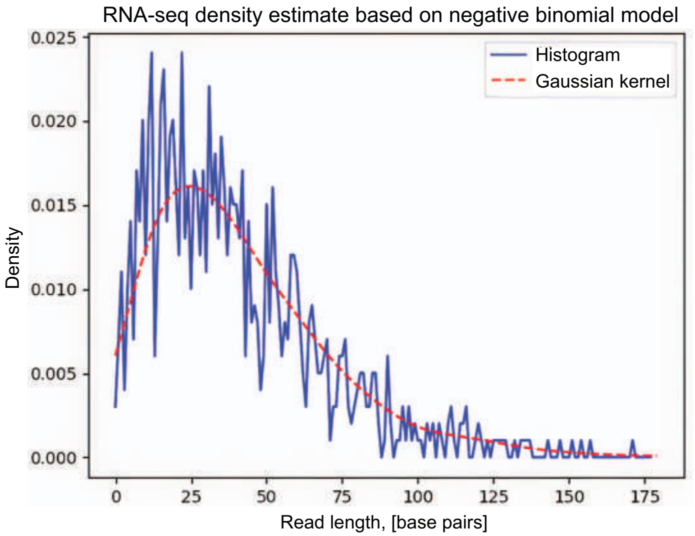

Figure 9.5 RNA-Seq density estimate via histogram and Gaussian KDE

9.2.2 Tangent portfolio optimization

The objective of mean-variance analysis is to maximize the expected return of a portfolio for a given level of risk, as measured by the standard deviation of past returns. By varying the mixing proportions of each asset, we can achieve different risk–return tradeoffs.

In listing 9.3, we first retrieve a data frame of closing prices for a list of stocks. We then examine stock price correlations via a scatter matrix plot. We then create a randomized portfolio and compute the portfolio risk. Next, we generate 1,000 randomly weighted portfolios and compute their value and risk. Finally, we choose the nearest neighbor portfolio weights in a way that minimizes standard deviation and maximizes portfolio value.

Listing 9.3 Tangent portfolio optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.neighbors import KDTree
from pandas.plotting import scatter_matrix
from scipy.spatial import ConvexHull
```

```
import pandas_datareader.data as web
from datetime import datetime
import pytz

STOCKS = ['SPY','LQD','TIP','GLD','MSFT']

np.random.seed(42)

if __name__ == "__main__":

    plt.close("all")

    #load data
    #year, month, day, hour, minute, second, microsecond
    start = datetime(2012, 1, 1, 0, 0, 0, 0, pytz.utc)
    end = datetime(2017, 1, 1, 0, 0, 0, 0, pytz.utc)

    data = pd.DataFrame()
    series = []
    for ticker in STOCKS:
        price = web.DataReader(ticker, 'stooq',
        ➥ start, end)
        series.append(price['Close'])                Loads data

    data = pd.concat(series, axis=1)
    data.columns = STOCKS
    data = data.dropna()

    scatter_matrix(data, alpha=0.2, diagonal='kde')     Plots data correlations
    plt.show()

    cash = 10000                                        Gets the current portfolio
    num_assets = np.size(STOCKS)
    cur_value = (1e4-5e3)*np.random.rand(num_assets,1) + 5e3
    tot_value = np.sum(cur_value)
    weights = cur_value.ravel()/float(tot_value)

    Sigma = data.cov().values
    Corr = data.corr().values
    volatility = np.sqrt(np.dot(weights.T,
    ➥ np.dot(Sigma, weights)))                  Computes portfolio risk

    plt.figure()
    plt.title('Correlation Matrix')
    plt.imshow(Corr, cmap='gray')
    plt.xticks(range(len(STOCKS)),data.columns)
    plt.yticks(range(len(STOCKS)),data.columns)
    plt.colorbar()
    plt.show()

                                                          Generates random
    num_trials = 1000                                     portfolio weights
    W = np.random.rand(num_trials, np.size(weights))
    W = W/np.sum(W,axis=1).reshape(num_trials,1)
                                                      Normalizes
```

```
pv = np.zeros(num_trials)     Portfolio value w'v
ps = np.zeros(num_trials)
                               Portfolio sigma: sqrt(w'Sw)
avg_price = data.mean().values
adj_price = avg_price

for i in range(num_trials):
    pv[i] = np.sum(adj_price * W[i,:])
    ps[i] = np.sqrt(np.dot(W[i,:].T, np.dot(Sigma, W[i,:])))

points = np.vstack((ps,pv)).T
hull = ConvexHull(points)

plt.figure()
plt.scatter(ps, pv, marker='o', color='b', linewidth = 3.0, label =
➥ 'tangent portfolio')
plt.scatter(volatility, np.sum(adj_price * weights), marker = 's',
➥ color = 'r', linewidth = 3.0, label = 'current')
plt.plot(points[hull.vertices,0], points[hull.vertices,1], linewidth =
➥ 2.0)
plt.title('expected return vs volatility')
plt.ylabel('expected price')
plt.xlabel('portfolio std dev')
plt.legend()
plt.grid(True)
plt.show()

#query for nearest neighbor portfolio
knn = 5
kdt = KDTree(points)
query_point = np.array([2, 115]).reshape(1,-1)
kdt_dist, kdt_idx = kdt.query(query_point,k=knn)
print("top-%d closest to query portfolios:" %knn)
print("values: ", pv[kdt_idx.ravel()])
print("sigmas: ", ps[kdt_idx.ravel()])
```

Figure 9.6 shows regression results between pairs of portfolio assets (left). Notice, for example, how SPY is uncorrelated with TIP and anticorrelated with GLD. Additionally, the diagonal densities are multimodal and show negative skewness for riskier assets (e.g., SPY versus LQD). Figure 9.6 also shows the expected return versus risk tradeoff for a set of randomly generated portfolios (right). The efficient frontier is defined by a set of portfolios at the top of the curve that corresponds to the maximum expected return for a given standard deviation. By adding a risk-free asset, we can choose a portfolio along a tangent line with the slope equal to the Sharpe ratio. In the following section, we will learn how to discover structure in relational data.

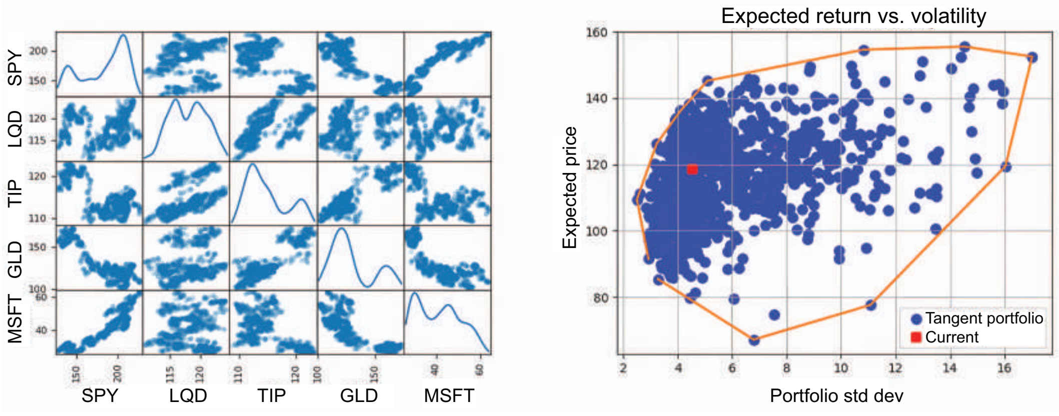

Figure 9.6 Pair plot (left) and tangent portfolio (right)

9.3 Structure learning

In this section, we will discover a graph's structure, given relational data. In other words, we would like to evaluate the probability of graph $G = (V, E)$, given observed data D. One challenge in inferring the graph structure is the exponential number of possible graphs. For example, in a directed graph G, we can have V choose 2 edges; since each edge has two possible directions, we have $O(2\verb|^|V(V-1)/2)$ possible graphs. Since the problem of structure learning for general graphs is *NP*-hard, we will focus on approximate methods. Namely, we'll look at the Chow-Liu algorithm for tree-based graphs as well as inverse covariance estimation for general graphs.

9.3.1 Chow-Liu algorithm

We can define the joint probability model for a tree T as follows.

$$p(x|T) = \prod_{t \in V} p(x_t) \prod_{(s,t) \in E} \frac{p(x_s, x_t)}{p(x_s)p(x_t)} \tag{9.12}$$

Here, $p(x_t)$ is a node marginal and $p(x_s, x_t)$ is an edge marginal. For example, for a $|V| = 3$ node V-shaped undirected tree, we have the following.

$$\begin{aligned} p(x_1, x_2, x_3|T) &= p(x_1)p(x_2)p(x_3)\frac{p(x_1, x_2)p(x_2, x_3)}{p(x_1)p(x_2)p(x_2)p(x_3)} \\ &= \frac{p(x1, x2)p(x_2, x_3)}{p(x_2)} = p(x_2)p(x_1|x_2)p(x_3|x_2) \end{aligned} \tag{9.13}$$

To derive the Chow-Liu algorithm, we can use the tree decomposition in equation 9.13 to write down the likelihood.

$$\begin{aligned}\log P(D|\theta, T) &= \sum_t \sum_k N_{tk} \log p(x_t = k|\theta) \\ &+ \sum_{s,t} \sum_{j,k} N_{stjk} \log \frac{p(x_s = j, x_t = k|\theta)}{p(x_s = j|\theta)p(x_t = k|\theta)}\end{aligned} \tag{9.14}$$

Here, N_{stjk} is the number of times node s is in state j and node t is in state k and N_{tk} is the number of times node t is in state k. We can rewrite N_{tk} as $N \times p(x_t = k)$ and, similarly, N_{stjk} as $N \times p(x_s = j, x_t = k)$. If we plug this in to our expression for log likelihood, we get the following.

$$\begin{aligned}\frac{1}{N} \log P(D|\theta, T) &= \sum_{t \in V} \sum_k \hat{p}(x_t = k) \log \hat{p}(x_t = k) \\ &+ \sum_{(s,t) \in E} I(x_s, x_t|\hat{\theta}_{st})\end{aligned} \tag{9.15}$$

Here, $I(x_s, x_t | \theta)$ is the mutual information between x_s and x_t. Therefore, the tree topology that maximizes the log likelihood can be computed via the maximum weight spanning tree, where the edge weights are pairwise mutual information $I(x_s, x_t | \theta)$. The algorithm in equation 9.15 is known as the *Chow-Liu algorithm*. Note that to compute the maximum spanning tree, we can use either Prim's algorithm or Kruskal's algorithms, which can be implemented in $O(E \log V)$ time. In the following section, we will examine how to infer the structure of a general graph based on inverse covariance estimation.

9.3.2 Inverse covariance estimation

Identifying stock clusters helps one discover similar companies, which can be useful for comparative analysis or a pairs trading strategy. We can find similar clusters by estimating the inverse covariance (precision) matrix that can be used to construct a graph network of dependencies, using the fact that zeros in the precision matrix correspond to the absence of edges in the constructed graph. Let's represent our unknown graph structure as a Gaussian graphical model. Let $\Lambda = \Sigma - 1$ represent the precision matrix of the multivariate normal. Then, the log likelihood of Λ can be derived as follows.

$$l(\Lambda) = \log \det \Lambda - \mathrm{tr}[S\Lambda] \tag{9.16}$$

Here, S is the empirical covariance matrix.

$$S = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)(x_i - \mu)^T \tag{9.17}$$

To encourage sparse structure, we can add a penalty term for nonzero entries in the precision matrix. Thus, our graph lasso negative log likelihood objective becomes as follows.

$$\mathrm{NLL}(\Lambda) = -\log\det\Lambda + \mathrm{tr}[S\Lambda] + c\|\Lambda\|_1 \tag{9.18}$$

In listing 9.4, we'll be using the difference between opening and closing daily prices to compute empirical covariance, which is used to fit the graph lasso algorithm to estimate the sparse precision matrix. Affinity propagation is used to compute the stock clusters and a linear embedding is used to display high dimensional data in 2D.

Listing 9.4 Inverse covariance estimation

```
import numpy as np
import pandas as pd
from scipy import linalg

from datetime import datetime
import pytz

from sklearn.datasets import make_sparse_spd_matrix
from sklearn.covariance import GraphicalLassoCV, ledoit_wolf
from sklearn.preprocessing import StandardScaler
from sklearn import cluster, manifold

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

import pandas_datareader.data as web

np.random.seed(42)

def main():

    #generate data (synthetic)
    #num_samples = 60
    #num_features = 20
    #prec = make_sparse_spd_matrix(num_features, alpha=0.95,
    ➥ smallest_coef=0.4, largest_coef=0.7)
    #cov = linalg.inv(prec)
    #X = np.random.multivariate_normal(np.zeros(num_features), cov,
    ➥ size=num_samples)
```

```
#X = StandardScaler().fit_transform(X)

#generate data (actual)
STOCKS = {
    'SPY': 'S&P500',
    'LQD': 'Bond_Corp',
    'TIP': 'Bond_Treas',
    'GLD': 'Gold',
    'MSFT': 'Microsoft',
    'XOM':  'Exxon',
    'AMZN': 'Amazon',
    'BAC':  'BofA',
    'NVS':  'Novartis'}

symbols, names = np.array(list(STOCKS.items())).T

#load data
#year, month, day, hour, minute, second, microsecond
start = datetime(2015, 1, 1, 0, 0, 0, 0, pytz.utc)
end = datetime(2017, 1, 1, 0, 0, 0, 0, pytz.utc)

qopen, qclose = [], []
data_close, data_open = pd.DataFrame(), pd.DataFrame()
for ticker in symbols:
    price = web.DataReader(ticker, 'stooq', start, end)
    qopen.append(price['Open'])
    qclose.append(price['Close'])

data_open = pd.concat(qopen, axis=1)
data_open.columns = symbols
data_close = pd.concat(qclose, axis=1)
data_close.columns = symbols

variation = data_close - data_open
variation = variation.dropna()
```

Per day variation in price for each symbol

```
X = variation.values
X /= X.std(axis=0)
```

Standardizes to use correlation rather than covariance

```
graph = GraphicalLassoCV()
graph.fit(X)
```

Estimates inverse covariance

```
gl_cov = graph.covariance_
gl_prec = graph.precision_
gl_alphas = graph.cv_alphas_
gl_scores = graph.cv_results_[‘mean_test_score’]

plt.figure()
sns.heatmap(gl_prec, xticklabels=names, yticklabels=names)
plt.xticks(rotation=45)
plt.yticks(rotation=45)
plt.tight_layout()
plt.show()

plt.figure()
```

```
    plt.plot(gl_alphas, gl_scores, marker='o', color='b', lw=2.0,
    ➥ label='GraphLassoCV')
    plt.title("Graph Lasso Alpha Selection")
    plt.xlabel("alpha")
    plt.ylabel("score")
    plt.legend()
    plt.show()

    _, labels = cluster.affinity_propagation(gl_cov)
    num_labels = np.max(labels)

    for i in range(num_labels+1):
        print("Cluster %i: %s" %((i+1), ', '.join(names[labels==i])))

    node_model = manifold.LocallyLinearEmbedding(
    ➥ n_components=2, n_neighbors=6, eigen_solver='dense')
    embedding = node_model.fit_transform(X.T).T

    #generate plots
    plt.figure()
    plt.clf()
    ax = plt.axes([0.,0.,1.,1.])
    plt.axis('off')

    partial_corr = gl_prec
    d = 1 / np.sqrt(np.diag(partial_corr))
    non_zero = (np.abs(np.triu(partial_corr, k=1)) >
    ➥ 0.02)

    #plot the nodes
    plt.scatter(embedding[0], embedding[1], s = 100*d**2, c = labels, cmap
    ➥ = plt.cm.Spectral)

    #plot the edges
    start_idx, end_idx = np.where(non_zero)
    segments = [[embedding[:,start], embedding[:,stop]] for start, stop in
    ➥ zip(start_idx, end_idx)]
    values = np.abs(partial_corr[non_zero])
    lc = LineCollection(segments, zorder=0, cmap=plt.cm.hot_r,
    ➥ norm=plt.Normalize(0,0.7*values.max()))
    lc.set_array(values)
    lc.set_linewidths(2*values)
    ax.add_collection(lc)

    #plot the labels
    for index, (name, label, (x,y)) in enumerate(zip(names, labels,
    ➥ embedding.T)):
        plt.text(x,y,name,size=12)

    plt.show()

if __name__ == "__main__":
    main()
```

Clusters using affinity propagation

Finds a low dimensional embedding for visualization

Connectivity matrix

Figure 9.7 shows the sparse precision matrix estimated by the graph lasso algorithm.

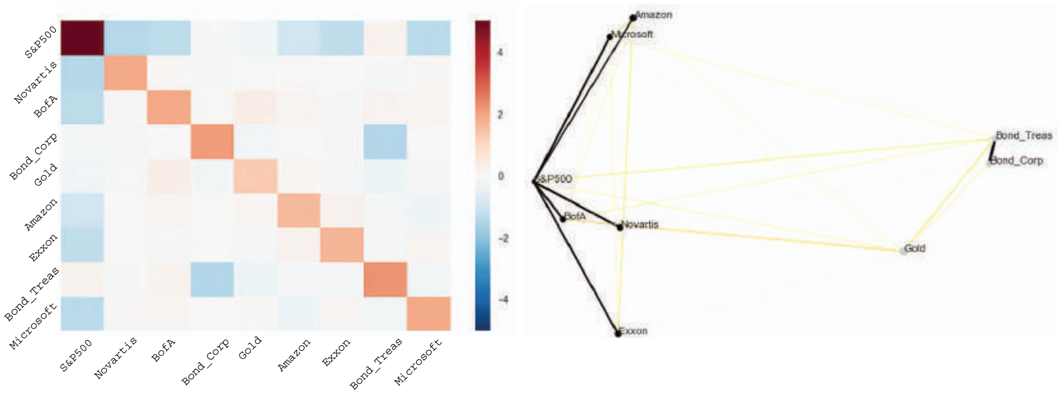

Figure 9.7 Graph lasso estimated precision matrix (left) and stock clusters (right)

The edge values in the precision matrix in figure 9.7 greater than a threshold correspond to connected components from which we can compute stock clusters, as visualized on the right-hand side of the figure. In the next section, we will learn about an energy minimization algorithm called simulated annealing.

9.4 Simulated annealing

Simulated annealing (SA) is a heuristic search method that allows for occasional transitions to less favorable states to escape local optima. We can formulate SA as an energy minimization problem with temperature parameter T as follows.

$$\begin{aligned} \alpha &= \exp\left\{\frac{E_{old} - E_{new}}{T}\right\} \\ p &= \min(1, \alpha) \end{aligned} \tag{9.19}$$

As we transition to a new state, we would like the energy in the new state to be lower—in which case, $\alpha > 1$ and $p = 1$—meaning we accept the state transition with the probability $p = 1$. On the other hand, if the energy of the new state is higher, this will cause $\alpha < 1$, and in that case, we accept the transition with probability $p = \alpha$. In other words, we accept unfavorable transitions with probability proportional to the difference in energies between states and inversely proportional to the temperature parameter T. Initially, the temperature T is high, allowing for many random transitions. As the temperature decreases (according to a cooling schedule), the difference in energy becomes more pronounced. The formulation in equation 9.19 allows simulated annealing to escape local optima. We are now ready to look at the pseudo-code in figure 9.8.

```
class simulated_annealing
function run(x_init, y_init):
converged = False
T = 1
x_old, y_old = x_init, y_init
energy_old = target(x_init, y_init)
while not converged:
    x_new, y_new = proposal(x_old, y_old) ← Evaluates the proposal
    energy_new = target(x_new, y_new) ← Computes the energy
    converged = check_convergence()
    alpha = exp((energy_old – energy_new)/T)
    r = min(1, alpha) ← Transition probability
    u = Unif[0,1]
    if u < r
        x_old, y_old = x_new, y_new  ┐
        energy_old = energy_new      ┘→ Accepts the proposed state
    end if
    T = temperature_schedule()
end while
x_opt, y_opt = x_old, y_old
return x_opt, y_opt
```

Figure 9.8 Simulated annealing pseudo-code

The `simulated_annealing` class contains the main `run` function. We begin by initializing the annealing temperature `T` and evaluating our `target` function at the initial location. Recall that we are interested in finding a minimum point in a complex energy landscape represented by the target function. We sample from our proposal distribution to obtain a new set of coordinates and evaluate the energy of the proposed coordinates. We check for convergence to decide whether to break out of the loop or continue. Next, we compute the simulated annealing transition probability alpha as a difference between old and new energy states divided by the temperature T. In the case of $r > 1$, we accept the low energy state with a probability of 1, and in the case of $0 < r < 1$ (i.e., the energy of the new state is higher), we accept the transition with probability $r = \alpha$. Finally, we adjust the temperature according to a cooling schedule, and upon convergence, we return the optimal coordinates (those that achieve the minimum energy found by simulated annealing). We are now ready to implement the simulated annealing algorithm from scratch in the following listing.

Listing 9.5 Simulated annealing

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
```

```
class simulated_annealing():
    def __init__(self):
        self.max_iter = 1000
        self.conv_thresh = 1e-4
        self.conv_window = 10

        self.samples = np.zeros((self.max_iter, 2))
        self.energies = np.zeros(self.max_iter)
        self.temperatures = np.zeros(self.max_iter)

    def target(self, x, y):          ◁ Energy landscape
        z = 3*(1-x)**2 * np.exp(-x**2 - (y+1)**2) \
            - 10*(x/5 -x**3 - y**5) * np.exp(-x**2 - y**2) \
            - (1/3)*np.exp(-(x+1)**2 - y**2)
        return z

    def proposal(self, x, y):
        mean = np.array([x, y])
        cov =  1.1 * np.eye(2)
        x_new, y_new = np.random.multivariate_normal(mean, cov)
        return x_new, y_new

    def temperature_schedule(self, T, iter):
        return 0.9 * T

    def run(self, x_init, y_init):

        converged = False
        T = 1
        self.temperatures[0] = T
        num_accepted = 0
        x_old, y_old = x_init, y_init
        energy_old = self.target(x_init, y_init)

        iter = 1
        while not converged:
            print("iter: {:4d}, temp: {:.4f}, energy = {:.6f}".format(iter,
            ➥ T, energy_old))
            x_new, y_new = self.proposal(x_old, y_old)
            energy_new = self.target(x_new, y_new)

            if iter > 2*self.conv_window:          ◁ Checks the convergence
                vals = self.energies[iter-self.conv_window : iter-1]
                if (np.std(vals) < self.conv_thresh):
                    converged = True
                #end if
            #end if

            alpha = np.exp((energy_old - energy_new)/T)
            r = np.minimum(1, alpha)
            u = np.random.uniform(0, 1)
            if u < r:
                x_old, y_old = x_new, y_new
                num_accepted += 1
                energy_old = energy_new
            #end if
            self.samples[iter, :] = np.array([x_old, y_old])
            self.energies[iter] = energy_old

            T = self.temperature_schedule(T, iter)
            self.temperatures[iter] = T

            iter = iter + 1

            if (iter > self.max_iter): converged = True
        #end while

        niter = iter - 1
        acceptance_rate = num_accepted / niter
        print("acceptance rate: ", acceptance_rate)

        x_opt, y_opt = x_old, y_old

        return x_opt, y_opt, self.samples[:niter,:], self.energies[:niter],
        ➥ self.temperatures[:niter]

if __name__ == "__main__":

    SA = simulated_annealing()

    nx, ny = (1000, 1000)
    x = np.linspace(-2, 2, nx)
    y = np.linspace(-2, 2, ny)
    xv, yv = np.meshgrid(x, y)

    z = SA.target(xv, yv)
    plt.figure()
    plt.contourf(x, y, z)
    plt.title("energy landscape")
    plt.show()

    #find global minimum by exhaustive search
    min_search = np.min(z)
    argmin_search = np.argwhere(z == min_search)
    xmin, ymin = argmin_search[0][0], argmin_search[0][1]
    print("global minimum (exhaustive search): ", min_search)
    print("located at (x, y): ", x[xmin], y[ymin])

    #find global minimum by simulated annealing
    x_init, y_init = 0, 0
    x_opt, y_opt, samples, energies, temperatures = SA.run(x_init, y_init)
    print("global minimum (simulated annealing): ", energies[-1])
    print("located at (x, y): ", x_opt, y_opt)

    plt.figure()
    plt.plot(energies)
    plt.title("SA sampled energies")
    plt.show()
plt.figure()
plt.plot(temperatures)
plt.title("Temperature Schedule")
plt.show()
```

Figure 9.9 shows the target energy landscape (left) and SA sampled energies (right).

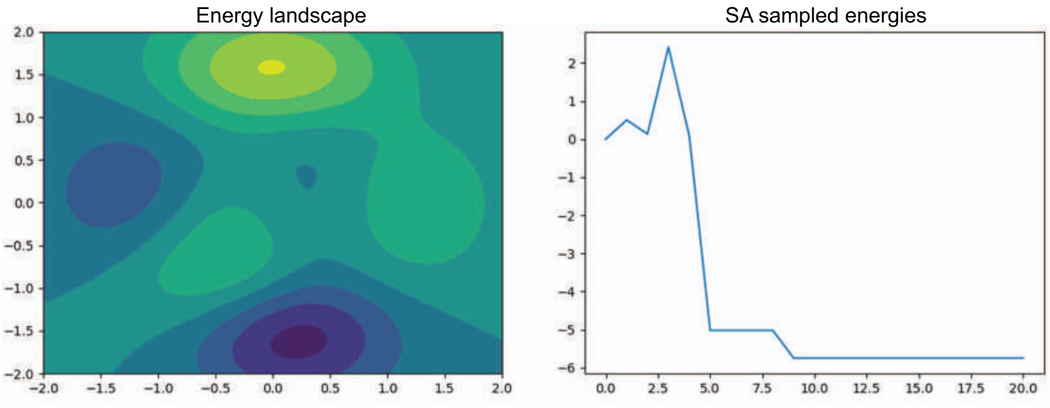

Figure 9.9 Energy landscape (left) and SA sampled energies (right)

In the example in figure 9.9, we were able to find the global minimum of an energy landscape using simulated annealing that matches the global minimum by exhaustive search. In the following section, we will study a genetic algorithm inspired by evolutionary biology.

9.5 Genetic algorithm

Genetic algorithms (GAs) are inspired by and modeled after evolutionary biology. They consists of a population of (randomly initialized) individual genomes that are evaluated for their fitness with respect to a target. Two individuals combine and cross over their genome to produce offspring. This crossover step is followed by a mutation, by which individual bases can change according to a mutation rate. These new individuals are added to the population and scored by the fitness function, which determines whether the individual survives in the population. Let's look at the pseudo-code in figure 9.10. In this example, we will evolve a random string to match a string target.

The `GeneticAlgorithm` class consists of the following functions: `calculate_fitness`, `mutate`, `crossover`, and `run`. In the `calculate_fitness` function, we compute the fitness score as 1/loss, where loss is defined as a distance between individual and target ASCII string characters. In the `mutate` function, we randomly change string characters of an individual according to a mutation rate. In the `crossover` function, we choose an index at which to cross the parents' genomes at random and then carry out the crossover of the parents' genomes at the chosen index producing two children. Finally, in the `run` function, we initialize the population and compute population fitness. After

```
class GeneticAlgorithm
function calculate_fitness(population, target):
for individual in population:
    compute loss as a distance between individual and target
    compute fitness = 1 / (loss + epsilon)
end for
return population_fitness
function mutate(individual, mutation_rate):
randomly change characters with probability equal to mutation_rate
return new_individual
function crossover(parent1, parent2):
cross_idx = random_integer(0, len(parent1))
child1 = parent1[:cross_idx] + parent2[cross_idx:]
child2 = parent2[:cross_idx] + parent1[cross_idx:]
return child1, child2
function run(target, population_size, mutation_rate):
init_population()
for epoch in num_iter:
    population_fitness = calculate_fitness(population, target)
    fittest_individual = population(argmax(population_fitness))
    if fittest_individual == target
        return fittest_individual
    end if
    parent_probabilities = fitness / sum(population_fitness)
    new_population = []
    for i in population_size:
        parent1, parent2 = random_choice(population, p= parent_probabilities)
        child1, child2 = crossover(parent1, parent2)
        new_population += [mutate(child1), mutate(child2)]
    end for
end for
```

Figure 9.10 Genetic algorithm pseudo-code

that, we find the fittest individual in the population and compare it with the target. If the two match, we exit the algorithm and return the fittest individual. Otherwise, we select two parents according to parent probabilities, ranked by fitness; crossover to produce offspring; mutate each offspring; and then add the offspring back to the new population. We repeat this process a fixed number of times or until the target is found. Let's now look at genetic algorithm implementation in detail.

Listing 9.6 Genetic algorithm implmentation

```
import numpy as np
import string

class GeneticAlgorithm():

    def __init__(self, target_string, population_size, mutation_rate):
        self.target = target_string
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.letters = [" "] + list(string.ascii_letters)

    def initialize(self):
        self.population = []
        for _ in range(self.population_size):
            individual = "".join(np.random.choice(self.letters,
            ➥ size=len(self.target)))
            self.population.append(individual)

    def calculate_fitness(self):
        population_fitness = []
        for individual in self.population:
            loss = 0
            for i in range(len(individual)):
                letter_i1 = self.letters.index(individual[i])
                letter_i2 = self.letters.index(self.target[i])
                loss += abs(letter_i1 - letter_i2)
            fitness = 1 / (loss + 1e-6)
            population_fitness.append(fitness)
        return population_fitness

    def mutate(self, individual):
        individual = list(individual)
        for j in range(len(individual)):
            if np.random.random() < self.mutation_rate:
                individual[j] = np.random.choice(self.letters)
        return "".join(individual)

    def crossover(self, parent1, parent2):
        cross_i = np.random.randint(0, len(parent1))
        child1 = parent1[:cross_i] + parent2[cross_i:]
        child2 = parent2[:cross_i] + parent1[cross_i:]
        return child1, child2

    def run(self, iterations):
        self.initialize()

        for epoch in range(iterations):
            population_fitness = self.calculate_fitness()

            fittest_individual =
            ➥ self.population[np.argmax(population_fitness)]
            highest_fitness = max(population_fitness)
            if fittest_individual == self.target:
                break
            parent_probabilities = [fitness / sum(population_fitness) for
            ➥ fitness in population_fitness]
            new_population = []
            for i in np.arange(0, self.population_size, 2):
                parent1, parent2 = np.random.choice(self.population,
                ➥ size=2, p=parent_probabilities, replace=False)
                child1, child2 = self.crossover(parent1, parent2)
```

Init population with random strings

Calculates the fitness of each individual in the population

Calculates the loss as the distance between characters

Randomly changes characters with a probability equal to the mutation rate

Creates children from parents by crossover

Selects parents proportional to their fitness

Next generation

Selects two parents

Crossover to produce offspring

Keeps mutated offspring for the next generation

```
                new_population += [self.mutate(child1), self.mutate(child2)]
            print("iter %d, closest candidate: %s, fitness: %.4f" %(epoch,
            ➥ fittest_individual, highest_fitness))
            self.population = new_population

        print("iter %d, final candidate: %s" %(epoch, fittest_individual))

if __name__ == "__main__":

    target_string = "Genome"
    population_size = 50
    mutation_rate = 0.1

    ga = GeneticAlgorithm(target_string, population_size, mutation_rate)
    ga.run(iterations = 1000)
```

As we can see from the output, we were able to produce the target sequence `"Genome"` by evolving randomly initialized letter sequences. In the next section, we will expand on the topics we've learned by reviewing the research literature on unsupervised learning.

9.6 ML research: Unsupervised learning

In this section, we cover additional insights and research related to the topics presented in this chapter. We had our first encounter with a Bayesian nonparametric model in the form of Dirichlet process *K*-means. The number of parameters in such models increases with data, and therefore, Bayesian nonparametric models are better able to model real-world scenarios. *DP*-means can be seen as a small variance asymptotics (SVA) approximation of the Dirichlet process mixture model. One of the main advantages of Bayesian nonparametric models is that they can be used for modeling infinite mixtures and hierarchical extensions can be utilized for sharing clusters across multiple data groups.

We looked at the EM algorithm, which is a powerful optimization framework used widely in machine learning. There are several extensions to the EM algorithm, such as online EM, which deals with online or streaming datasets; annealed EM, which uses the temperature parameter to smooth the energy landscape during optimization to track the global optimum; variational EM, which replaces exact inference in the E step with variational inference; Monte Carlo EM, which draws samples in the E step from the intractable distribution; and several others.

We looked at two ways to reduce dimensionality for the purpose of feature selection or data visualization. There are several other ways to learn the underlying data manifold, such as Isomap, which is an extension of Kernel PCA that seeks to maintain geodesic distances between all points; locally linear embedding (LLE), which can be thought of as a series of local PCA globally compared to find the best nonlinear embedding; spectral embedding based on the decomposition of the graph Laplacian; and multidimensional scaling (MDS) in which the distances in the embedding reflect the distances in the original high dimensional space well. Note that some of these methods can be combined, such as PCA, which can be used to preprocess the data and initialize t-SNE to reduce the computational complexity.

We looked at a powerful way to discover topics in text documents using the variational Bayes algorithm for latent Dirichlet allocation, for which there are several extensions as well. For example, the correlated topic model captures correlations between topics, the dynamic topic model tracks the evolution of topics over time, and the supervised LDA model can be used to grade or assign scores to documents to evaluate their quality.

We looked at the problem of density estimation using kernels and discovered the effect of the smoothing parameter on the resulting estimate. We can implement KDE more efficiently by using ball tree or KD tree to reduce the time complexity required to query the data.

Regarding structure learning, we discovered the exponential number of possible graph topologies and touched on the topic of causality. We looked at how we could construct simpler tree graphs using mutual information between nodes as edge weights in the maximum weight spanning tree. We also saw how regularizing the inverse covariance matrix for general graphs led to more interpretable topologies with fewer edges in the inferred graph.

Finally, we looked at two unsupervised algorithms inspired by statistical physics (simulated annealing) and evolutionary biology (the genetic algorithm). We saw how by using the temperature parameter and a cooling schedule, we can modify the energy landscape and how selecting unfavorable in the short-term moves can lead to better long-term optima. There are many NP-hard problems that can be approximated with simulated annealing, including the traveling salesman problem (TSP). We can often use several restarts and different initialization points to arrive at better optima. Genetic algorithms, on the other hand, although satisfying in their parallelism with nature, can take a long time to converge. However, they can lead to several interesting applications, such as neural network architecture search.

9.7 Exercises

9.1 Explain how latent Dirichlet allocation can be interpreted as nonnegative matrix factorization.

9.2 Explain why sparsity is desirable in inferring general graph structure.

9.3 List several NP-hard problems that can be approximated with the simulated annealing algorithm.

9.4 Brainstorm problems that can be efficiently solved by applying a genetic algorithm.

Summary

- Latent Dirichlet allocation (LDA) is a topic model that represents each document as a finite mixture of topics, where a topic is a distribution over words. The objective is to learn the shared topic distribution and topic proportions for each document.
- A common method for adjusting the word counts is tf-idf that logarithmically drives down to zero word counts that occur frequently across documents: $A \log (D / n_t)$, where D is the total number of documents in the corpus and n_t is the number of documents in which the term t appears.
- The goal of density estimation is to model the probability density of data. Kernel density estimation (KDE) allocates one cluster center per data point.
- The objective of mean-variance analysis is to maximize the expected return of a portfolio for a given level of risk, as measured by the standard deviation of past returns.
- Since the problem of structure learning for general graphs is NP-hard, we focused on approximate methods. Namely, we looked at the Chow-Liu algorithm for tree-based graphs as well as inverse covariance estimation for general graphs.
- Simulated annealing (SA) is a heuristic search method that allows for occasional transitions to less favorable states to escape local optima.
- We can formulate simulated annealing as an energy minimization problem with temperature parameter T, with which we can modify the energy landscape and select moves that are unfavorable in the short term but can lead to better long-term optima.
- Genetic algorithms (GA) are inspired by and modeled after evolutionary biology. They consist of a population of (randomly initialized) individual genomes that are evaluated for their fitness with respect to a target.
- In genetic algorithms, two individuals combine and cross over their genome to produce an offspring. This crossover step is followed by a mutation by which individual bases can change according to a mutation rate. The resulting offspring are added to the population and scored according to their fitness level.