6 Regression algorithms

This chapter covers

- Introducing regression
- Bayesian linear regression
- Hierarchical Bayesian regression
- KNN regression
- Gaussian process regression

In the previous chapter, we looked at supervised algorithms for classification. In this chapter, we will focus on supervised learning, in which we are trying to predict a continuous quantity. We will study four intriguing regression algorithms, which we will derive from the first principles: Bayesian linear regression, hierarchical Bayesian regression, KNN regression, and Gaussian process regression. These algorithms were selected because they span several important applications and illustrate diverse mathematical concepts. Regression algorithms are useful in a variety of applications. For example, they can be used to predict the price of financial assets or predict CO_2 levels in the atmosphere. Let's begin by reviewing the fundamentals of regression.

6.1 Introduction to regression

In supervised learning, we are given a dataset $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ consisting of tuples of data x and labels y. The goal of a regression algorithm is to learn a mapping from inputs x to outputs y, where y is a continuous quantity (i.e., $y \in R$).

A regressor f can be viewed as a mapping between a d-dimensional feature vector $\varphi(x)$ and a label y (i.e., f: $R^d \rightarrow R$). Regression problems are typically harder (to achieve higher accuracy) compared to classification problems because we are trying to predict a *continuous* quantity. Moreover, we are often interested in predicting *future* response variable y based on past training data.

One of the most widely used models for regression is *linear regression*, which models the response variable y as a linear combination of input feature vectors $\varphi(x)$.

$$y(x) = \underbrace{w^T}_{\text{Regression weights}} \overbrace{\phi(x)}^{\text{Features}} + \underbrace{\epsilon}_{\text{Gaussian noise}} = \sum_{d=1}^{D} w_d \phi(x_d) + \epsilon \tag{6.1}$$

Here, ϵ is the residual error between our linear predictions and the true response. We can characterize the quality of our regressor based on the mean squared error (MSE).

$$\text{MSE}(w) = E\left[(\underbrace{y}_{\text{Ground truth}} - \overbrace{f(x;w)}^{\text{Prediction}})^2\right] = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - w^T\phi(x_i)\right)^2 \tag{6.2}$$

In this chapter, we will look at several important regression models, starting with Bayesian and KNN regression, and their extensions to hierarchical regression models, and concluding with Gaussian process (GP) regression. We'll focus on both the theory and implementation of each model from scratch.

6.2 Bayesian linear regression

Recall that we can write the linear regression as $y(x) = w^{\mathrm{T}}x + \epsilon$. If we assume that $\epsilon \sim N(0, \sigma^2)$ is a zero-mean Gaussian RV with a variance σ^2, then we can formulate linear regression as follows.

$$p(y|x, \theta) = N\left(y|w^T x, \sigma^2\right) \tag{6.3}$$

Here, w are regression coefficients. To fit a linear regression model to data, we minimize the negative log likelihood.

$$
\begin{aligned}
\mathrm{NLL}(w, \sigma^2) &= -\log p(y|x, \theta) \leftarrow \text{Definition of NLL} \\
&= -\log \prod_{n=1}^{N} \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2} \left(y_n - w^T x \right)^2 \right] \right] \\
&= -\sum_{n=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) + \sum_{n=1}^{N} \frac{1}{2\sigma^2} \left(y_n - w^T x \right)^2 \\
&= \frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(y_n - w^T x \right)^2 + \frac{N}{2} \log\left(2\pi\sigma^2 \right) \qquad (6.4)
\end{aligned}
$$

Keeping σ^2 fixed and differentiating with respect to w, we get the following equation.

$$2X^T Xw - 2X^T y + 2\lambda w = 0 \qquad (6.5)$$

From equation 6.5, we can write the following.

$$\hat{w} = \left(X^T X + \lambda I \right)^{-1} X^T y \qquad (6.6)$$

One problem with the estimation in equation 6.6 is that it can result in overfitting. To make the Bayesian linear regression robust against overfitting, we can encourage the parameters to be small by placing a zero-mean Gaussian prior.

$$p(w) = \prod_{d} N\left(w_d | 0, \tau^2 \right) \qquad (6.7)$$

Thus, we can rewrite our regularized objective as follows.

$$\min_w \mathrm{NLL}\left(w, \sigma^2 \right) + \lambda \|w\|_2^2 \qquad (6.8)$$

Solving for w as before, we get the following coefficients.

$$\hat{w}_{ridge} = \left(X^T X + \lambda I \right)^{-1} X^T y \qquad (6.9)$$

We can learn the parameters w using gradient descent! Let's look at the pseudo-code in figure 6.1.

```
class ridge_reg:
function fit($X, y$)
for $i = 1, 2, \ldots,$ num_iter
    $\hat{y} = w^T X$
    $grad = -(y - \hat{y})^T X + \lambda w$
    $w = w - \eta_i \, grad$ ← Update regression weights
end for
return $w$
function predict($w, X$)
$\hat{y} = w^T X$ ← Make a prediction
return $\hat{y}$
```

Figure 6.1 Bayesian linear regression pseudo-code

The ridge_reg class consists of the fit and predict functions. In the fit function, we compute the gradient of the objective function wrt w and update the weight parameters. In the predict function, we use the learned regression weights to make a prediction on test data.

Listing 6.1 Bayesian linear regression

```
import math
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing

class ridge_reg():

    def __init__(self, n_iter=20, learning_rate=1e-3, lmbda=0.1):
     self.n_iter = n_iter
     self.learning_rate = learning_rate
     self.lmbda = lmbda

  def fit(self, X, y):
     X = np.insert(X, 0, 1, axis=1)

     self.loss = []
     self.w = np.random.rand(X.shape[1])

     for i in range(self.n_iter):
         y_pred = X.dot(self.w)
         mse = np.mean(0.5*(y - y_pred)**2 +
         ➥ 0.5*self.lmbda*self.w.T.dot(self.w))
         self.loss.append(mse)
         print(" %d iter, mse: %.4f" %(i, mse))
         grad_w = - (y - y_pred).dot(X) + self
         ➥ .lmbda*self.w
         self.w -= self.learning_rate * grad_w

 def predict(self, X):
```

Inserts const 1 for the bias term

Computes the gradient of NLL(w) wrt w

Updates the weights

```
        X = np.insert(X, 0, 1, axis=1)
        y_pred = X.dot(self.w)
        return y_pred
```

Inserts const 1 for the bias term

```
if __name__ == "__main__":

    X, y = fetch_california_housing(return_X_y=True)
    X_reg = X[:,2].reshape(-1,1)
    X_std = (X_reg - X_reg.mean())/X.std()
    y_std = (y - y.mean())/y.std()
```

Average number of rooms

Standard scaling

```
    X_std = X_std[:200,:]
    y_std = y_std[:200]

    rr = ridge_reg()
    rr.fit(X_std, y_std)
    y_pred = rr.predict(X_std)

    print(rr.w)

    plt.figure()
    plt.plot(rr.loss)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.tight_layout()
    plt.show()

    plt.figure()
    plt.scatter(X_std, y_std)
    plt.plot(np.linspace(-1,1), rr.w[1]*np.linspace(-1,1)+rr.w[0], c='red')
    plt.xlim([-0.01,0.01])
    plt.xlabel("scaled avg num of rooms")
    plt.ylabel("scaled house price")
    plt.show()
```

Figure 6.2 shows the output of Bayesian linear regression algorithm.

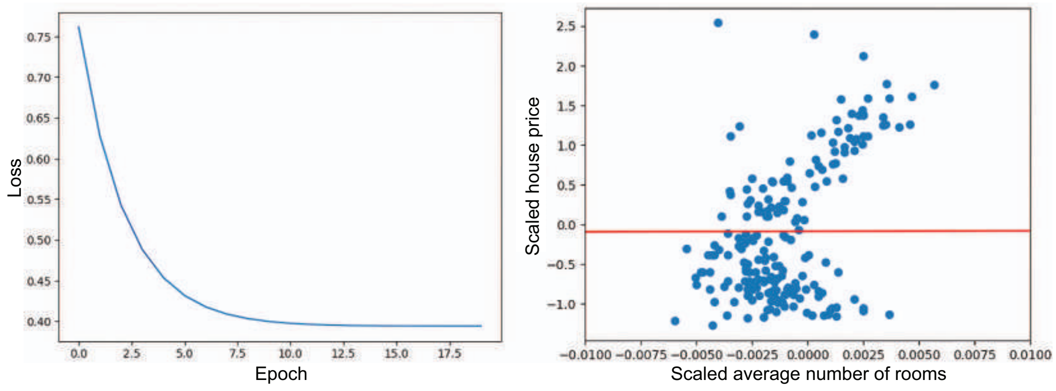

Figure 6.2 Bayesian linear regression loss function (left) and plot (right)

We can see the decrease in loss function over epochs on the left and a fit with the California house pricing dataset projection on the right. Note that both axes are standardized. The Bayesian linear regression successfully captures the trend of increasing house prices with the average number of rooms. In the next section, we will examine the benefits of a hierarchical model of linear regression.

6.3 Hierarchical Bayesian regression

Hierarchical models enable the sharing of features among groups. The parameters of the model are assumed to be sampled from a common distribution that models similarity between groups. Figure 6.3 shows three scenarios that illustrate the benefit of hierarchical modeling. In the figure on the left, we have a single set of parameters θ that model the entire sequence of observations, referred to as a *pooled model*. Here, any variation in data is not modeled explicitly, since we are assuming a common set of parameters that give rise to the data. On the other hand, we have an *unpooled model* scenario, in which we model a different set of parameters for each observation. In the unpooled case, we assume there is no sharing of parameters between groups of observations and each parameter is independent. The hierarchical model combines the best of both worlds: it assumes there is a common distribution from which individual parameters are sampled and, therefore, captures similarities between groups.

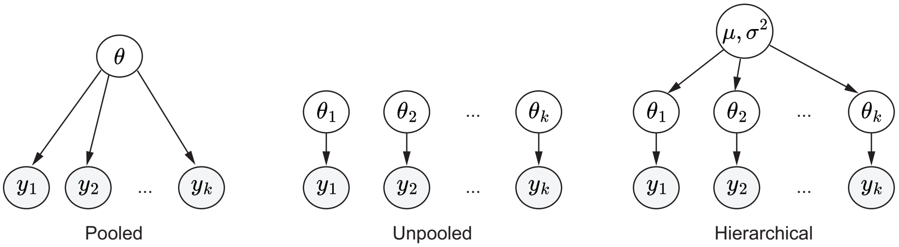

Figure 6.3 Pooled, unpooled, and hierarchical graphical models

In Bayesian hierarchical regression, we can assign priors on model parameters and use MCMC sampling to infer posterior distributions. We use the radon dataset to regress radon gas levels in houses of different counties, based on the floor number (in particular, whether or not there is a basement). Thus, our regression model looks like the following.

$$\begin{aligned}\alpha_c &\sim N\left(\mu_a, \sigma_a^2\right)\\ \beta_c &\sim N\left(\mu_\beta, \sigma_\beta^2\right)\\ \text{radon}_c &= \alpha_c + \beta_c \times \text{floor}_{i,c} + \epsilon_c\end{aligned} \tag{6.10}$$

Notice that subscript c indicates a county; thus, we are learning an intercept and a slope for each county sampled from a shared Gaussian distribution. Thus, we are assuming a hierarchical model in which our parameters (α_c and β_c) are sampled from common distributions. In code listing 6.2, we will define probability distributions over regression coefficients and model the data likelihood as the normal distribution with uniform standard deviation. Having specified the graphical model, we can run inference using the no-U-turn sampler (NUTS), with the help of the PyMC library. PyMC is a probabilistic programming library in Python that can be downloaded at https://docs.pymc.io/. It is an excellent tool for Bayesian modeling and can be considered from scratch, since we are defining the probabilistic graphical model from scratch and then using off-the-shelf tools for sampling the posterior distribution. If this is your first exposure to probabilistic programming languages, I highly recommend completing the PyMC online examples to learn more about its capabilities.

Let's examine the pseudo-code in figure 6.4.

```
function main($X$, $y$):
with pymc3.Model() as hierarchical_model:
    $\mu_a \sim N(0, 100^2)$            ─┐
    $\sigma_a \sim \text{Unif}[0, 100]$  │
    $\mu_b \sim N(0, 100^2)$             ├→ Hyperpriors
    $\sigma_b \sim \text{Unif}[0, 100]$ ─┘
    $a \sim N\left(\mu_a, \sigma_a^2\right)$ ← Intercept model
    $b \sim N\left(\mu_b, \sigma_b^2\right)$ ← Slope model
    $\epsilon \sim \text{Unif}[0, 100]$ ← Error model
    $y_{exp} = a + b \times X$ ← Expected value
    $y_{lh} \sim N\left(X; y_{exp}, \epsilon^2\right)$ ← Data likelihood
with hierarchical_model:
    mu, sds, elbo = pymc3.variational.advi(n = 100000)       ─┐
    step = pymc3.NUTS(scaling = sds$^2$, is_cov =True)        ├→ PyMC inference
    trace = pymc3.sample(5000, step, start = mu)             ─┘
return trace
```

Figure 6.4 Hierarchical Bayesian regression pseudo-code

The code consists of a single `main` function. In the first section of the code, we define the probabilistic model, and in the second section, we are using PyMC3 library for variational inference. First, we set the hyperpriors for the mean and variance of the regression intercept and slope models. Next, we define the intercept and slope model as the Gaussian RVs and define the error model as a uniform RV. Finally, we compute the regression expression and set it as a mean in the data likelihood model. We then proceed with NUTS inference implemented in PyMC.

Listing 6.2 Hierarchical Bayesian regression

```
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

import pymc3 as pm

def main():

 data = pd.read_csv('./data/radon.txt')      Load data

 county_names = data.county.unique()
 county_idx = data['county_code'].values

 with pm.Model() as hierarchical_model:

     mu_a = pm.Normal('mu_alpha', mu=0., sd=100**2)
     sigma_a = pm.Uniform('sigma_alpha', lower=0,
     ➥ upper=100)                                       Hyperpriors
     mu_b = pm.Normal('mu_beta', mu=0., sd=100**2)
     sigma_b = pm.Uniform('sigma_beta', lower=0,
     ➥ upper=100)

     a = pm.Normal('alpha', mu=mu_a, sd=sigma_a,
     ➥ shape=len(data.county.unique()))      Intercept for each county
     b = pm.Normal('beta', mu=mu_b, sd=sigma_b,
     ➥ shape=len(data.county.unique()))      Slope for each county

     eps = pm.Uniform('eps', lower=0, upper=100)      Model error

     radon_est = a[county_idx] + b[county_idx] *
     ➥ data.floor.values
                                   Expected value
     y_like = pm.Normal('y_like', mu=radon_est, sd=eps,
     ➥ observed=data.log_radon)
                                   Data likelihood

 with hierarchical_model:
     # Use ADVI for initialization
     mu, sds, elbo = pm.variational.advi(n=100000)
     step = pm.NUTS(scaling=hierarchical_model.dict_to_array(sds)**2,
     ➥ is_cov=True)
     hierarchical_trace = pm.sample(5000, step, start=mu)


 pm.traceplot(hierarchical_trace[500:])
 plt.show()

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

From the trace plots in figure 6.5, we can see convergence in our posterior distributions for α_c and β_c, indicating different intercepts and slopes for different counties.

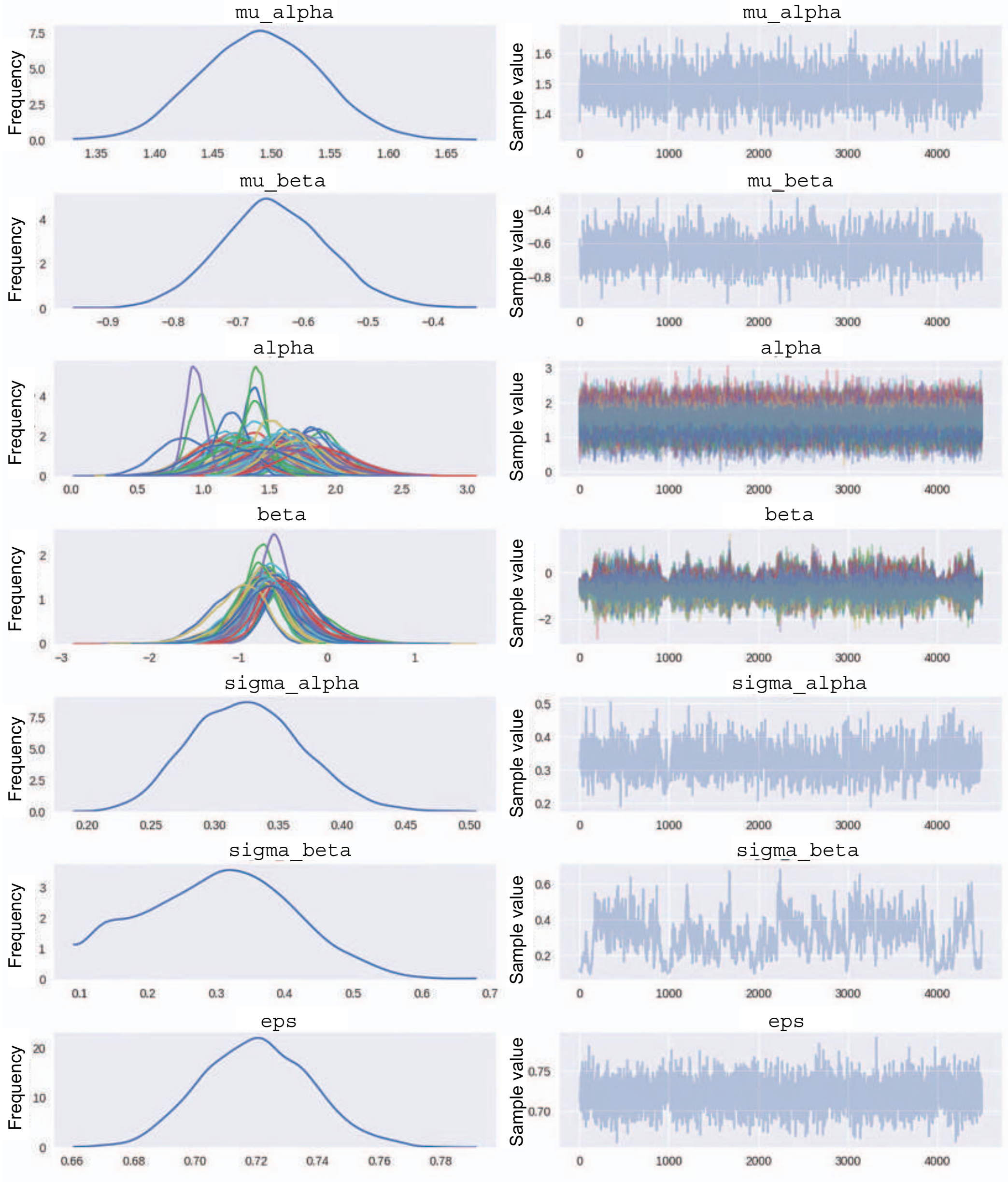

Figure 6.5 MCMC traceplots for hierarchical Bayesian regression

In addition, we also recover the posterior distribution of the shared parameters. μ_a means the group mean of log radon levels is close to 1.5, while μ_b means the slope is negative, with a mean of –0.65 and, therefore, having no basement decreases radon levels. In the next section, we will examine an algorithm suitable for nonlinear data.

6.4 KNN regression

K nearest neighbors (KNN) regression is an example of a nonparametric model, in which for a given query data point q, we find its k nearest neighbors in the training set and compute the average response variable y. In this section, we'll compute the average of KNN target labels for the iris dataset. The average is taken over the local neighborhood of K points that are closest to our query q.

$$y_q = \frac{1}{K} \sum_{i \in N_K(q,D)} y_i \tag{6.11}$$

Here, $N_K(q, D)$ denotes the local neighborhood of k nearest neighbors to query q from the training dataset D. To find the local neighborhood $N_K(q, D)$, we can compute a distance between the query point q and each of the training dataset points $x_i \in D$, sort these distances in ascending order, and take the top K data points. The runtime complexity of this approach is $O(n \log n)$, where n is the size of the training dataset due to the sort operation. We are now ready to implement a KNN regressor from scratch! In figure 6.6, KNN regression is computed by averaging the labels of K nearest neighbors based on Euclidean distance.

```
class KNN:
function knn_search(K, X, y, Q)
for query in Q:
    idx = argsort(euclidian_dist(query, X))[:K]  ← KNN IDs
    knn_labels = [y[i] for i in idx]  ← KNN labels
    y_pred = mean(knn_labels)  ← KNN regression
end for
return y_pred
```

Figure 6.6 KNN regression pseudo-code

The code consists of the `knn_search` function, in which for every K nearest neighbor query Q, we compute the Euclidean distance between the query and all of the data points, sort the results, and pick out K lowest distance IDs. We then form the KNN region by collecting the labels with KNN IDs. Finally, we compute our result by averaging over KNN labels.

Listing 6.3 K nearest neighbors regression

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

from sklearn import datasets
from sklearn.model_selection import train_test_split

np.random.seed(42)
```

```
class KNN():

  def __init__(self, K):
     self.K = K

  def euclidean_distance(self, x1, x2):
     dist = 0
     for i in range(len(x1)):
         dist += np.power((x1[i] - x2[i]), 2)
     return np.sqrt(dist)

  def knn_search(self, X_train, y_train, Q):
     y_pred = np.empty(Q.shape[0])

     for i, query in enumerate(Q):
         idx = np.argsort([self.euclidean_distance(
         ➥ query, x) for x in X_train])[:self.K]
         knn_labels = y_train[idx]
         y_pred[i] = np.mean(knn_labels)

     return y_pred


if __name__ == "__main__":

    plt.close('all')

    #iris dataset
    iris = datasets.load_iris()
    X = iris.data[:,:2]
    y = iris.target

    X_train, X_test, y_train, y_test = train_test_split(X, y,
    ➥ test_size=0.2, random_state=42)

    K = 4
    knn = KNN(K)
    y_pred = knn.knn_search(X_train, y_train, X_test)

    plt.figure(1)
    plt.scatter(X_train[:,0], X_train[:,1], s = 100, marker = 'x', color =
    ➥ 'r', label = 'data')
    plt.scatter(X_test[:,0], X_test[:,1], s = 100, marker = 'o', color =
    ➥ 'b', label = 'query')
    plt.title('K Nearest Neighbors (K=%d)'% K)
    plt.legend()
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.grid(True)
    plt.show()
```

Gets K nearest neighbors to query the point

Extracts KNN training labels

Computes the average of KNN training labels

Finding the exact nearest neighbors in high-dimensional space is often computationally intractable, and therefore, there exist approximate methods. There are two classes of approximate methods: those that partition the space into regions, such as k-d tree implemented in fast library for approximate nearest neighbors (FLANN), and hashing-based methods, such as locality-sensitive hashing (LSH). In the next section, we will discuss a different type of regression over functions.

6.5 Gaussian process regression

Gaussian processes (GPs) define a prior over functions that can be updated to a posterior once we have observed data (see C. E. Rasmussen and C. K. I. Williams's, *Gaussian Processes for Machine Learning*, The MIT Press, 2006). In a supervised setting, the function gives a mapping between the data points x_i and the target value y_i: $y_i = f(x_i)$. Gaussian processes infer a distribution over functions given the data $p(f|x, y)$ and then use it to make predictions given new data. A GP assumes that the function is defined at a finite and arbitrary chosen set of points $x_1, ..., x_n$, such that $p(f(x_1), ..., f(x_n))$ is jointly Gaussian with mean $\mu(x)$ and covariance $\Sigma(x)$, where $\Sigma_{ij} = \kappa(x_i, x_j)$ and κ is a positive definite kernel function. One example that can be solved by Gaussian Process regression is predicting the CO_2 level based on observed measurements. In code listing 6.4, we will assume a sinusoidal model and a radial basis function kernel.

Consider a simple regression problem.

$$f(x) = x^T w \quad y = f(x) + \epsilon \quad \epsilon \sim N\left(0, \sigma_n^2\right) \tag{6.12}$$

Assuming independent and identically distributed noise, we can write down the following likelihood function.

$$\begin{aligned} p(y|X, w) &= \prod_{i=1}^{n} p(y_i|x_i, w) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\left\{-\frac{\left(y_i - x_i^T w\right)^2}{2\sigma_n^2}\right\} \\ &\sim N\left(Xw, \sigma_n^2 I\right) \end{aligned} \tag{6.13}$$

In the Bayesian framework, we need to specify a prior over the parameters: $w \sim N(0, \Sigma_p)$. Writing only the terms of the likelihood and the prior, which depend on the weights, we get the following equation.

$$\begin{aligned} p(w|X, y) &\propto \exp\left\{-\frac{1}{2\sigma_n^2}||y - Xw||^2\right\} \exp\left\{-\frac{1}{2} w^T \Sigma_p^{-1} w\right\} \\ &\propto \exp\left\{-\frac{1}{2}(w - \bar{w})\left(\frac{1}{\sigma_n^2} X X^T + \Sigma_p^{-1}\right)(w - \bar{w})\right\} \\ &\sim N\left(\frac{1}{\sigma_n^2} A^{-1} X y, A^{-1}\right) \end{aligned} \tag{6.14}$$

Here, we assume the following.

$$A = \sigma_n^{-2} X X^T + \Sigma_p^{-1} \tag{6.15}$$

Thus, we have a closed-form posterior distribution over the parameters w. To make predictions using this equation, we need to invert the matrix A of size $p \times p$.

Assuming the observations are noiseless, we want to predict the function outputs $y^* = f(x^*)$, where x^* is our test data. Consider the following joint GP distribution.

$$\begin{pmatrix} f \\ f_* \end{pmatrix} \sim N\left(\begin{pmatrix} \mu \\ \mu_* \end{pmatrix}, \begin{pmatrix} K & K_* \\ K_*^T & K_{**} \end{pmatrix}\right) \tag{6.16}$$

Here, $K = \kappa(X, X)$, $K^* = \kappa(X, X^*)$ and $K^{**} = \kappa(X^*, X^*)$, where X is our training dataset, X^* is our test dataset, and κ is the kernel or covariance function. Using standard rules for conditioning Gaussians, the posterior has the following form.

$$\begin{aligned} p(f_*|X_*, X, f) &\sim N(f_*|\mu_*, \Sigma_*) \\ \mu_* &= \mu(X_*) + K_*^T K^{-1}(f - \mu(X)) \\ \Sigma_* &= K_{**} - K_*^T K^{-1} K_* \end{aligned} \tag{6.17}$$

In code listing 6.4, we use the radial basis function kernel as a measure of similarity defined in equation 6.18. We are now ready to implement GP regression from scratch (see figure 6.7)!

```
class GPreg:
    function kernel_func(x, z)
        Kfn = exp $\left\{-\frac{1}{2\sigma^2}\|x - x\|^2\right\}$   <- Radial basis function kernel
        return Kfn
    function compute_posterior(X)
        K = kernel_func(X_train, X_train)
        Ks = kernel_func(X_train, X_text)
        Kss = kernel_func(X_test, X_test)
        $\mu_{post} - \mu(X_{text}) + K_s^T K^{-1} (f - \mu(X_{train}))$   -> Gaussian process posterior
        $\Sigma_{post} = K_{ss} - K_s^T K^{-1} K_s$
        return $\mu_{post}, \Sigma_{post}$
```

Figure 6.7 Gaussian process regression pseudo-code

The code consists of `kernel_func` and `compute_posterior` functions. The `kernel_func` returns a radial basis function (RBF) kernel, which measures the similarity between two inputs x and z. In the `computer_posterior` function, we first compute the ingredients required for posterior mean and covariance equations and then compute and return the posterior mean and covariance, as derived in the text.

Listing 6.4 Gaussian process regression

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist

np.random.seed(42)

class GPreg:

 def __init__(self, X_train, y_train, X_test):

     self.L = 1.0
     self.keps = 1e-8

     self.muFn = self.mean_func(X_test)
     self.Kfn = self.kernel_func(X_test, X_test) + 1e-15*np.eye(
     ➥  np.size(X_test))

     self.X_train = X_train
     self.y_train = y_train
     self.X_test = X_test

 def mean_func(self, x):
     muFn = np.zeros(len(x)).reshape(-1,1)
     return muFn

 def kernel_func(self, x, z):
     sq_dist = cdist(x/self.L, z/self.L, 'euclidean')**2
     Kfn = 1.0 * np.exp(-sq_dist/2)
     return Kfn

 def compute_posterior(self):
     K = self.kernel_func(self.X_train, self.X_train)
     Ks = self.kernel_func(self.X_train, self.X_test)
     Kss = self.kernel_func(self.X_test, self.X_test) + self
     ➥  .keps*np.eye(np.size(self.X_test))
     Ki = np.linalg.inv(K)              ◁—— O(N_train ^ 3)

     postMu = self.mean_func(self.X_test) + np.dot(np.transpose(Ks),
     ➥  np.dot(Ki, (self.y_train - self.mean_func(self.X_train))))
     postCov = Kss - np.dot(np.transpose(Ks), np.dot(Ki, Ks))

     self.muFn = postMu
     self.Kfn = postCov

     return None
```

```
    def generate_plots(self, X, num_samples=3):
        plt.figure()
        for i in range(num_samples):
            fs = self.gauss_sample(1)
            plt.plot(X, fs, '-k')
            #plt.plot(self.X_train, self.y_train, 'xk')

        mu = self.muFn.ravel()
        S2 = np.diag(self.Kfn)
        plt.fill(np.concatenate([X, X[::-1]]), np.concatenate([mu -
        ➥ 2*np.sqrt(S2), (mu + 2*np.sqrt(S2))[::-1]]), alpha=0.2, fc='b')
        plt.show()

    def gauss_sample(self, n):
        A = np.linalg.cholesky(self.Kfn)
        Z = np.random.normal(loc=0, scale=1, size=(len(self.muFn),n))
        S = np.dot(A,Z) + self.muFn
        return S

def main():

    # generate noise-less training data
    X_train = np.array([-4, -3, -2, -1, 1])
    X_train = X_train.reshape(-1,1)
    y_train = np.sin(X_train)

    # generate  test data
    X_test = np.linspace(-5, 5, 50)
    X_test = X_test.reshape(-1,1)

    gp = GPreg(X_train, y_train, X_test)
    gp.generate_plots(X_test,3)
    gp.compute_posterior()
    gp.generate_plots(X_test,3)


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

Returns n samples from the multivariate Gaussian distribution

S = AZ + mu

Samples from the GP prior

Samples from the GP posterior

Figure 6.8 shows three functions drawn at random from a GP prior (left) and GP posterior (right) after observing five data points in the case of noise-free observations. The shaded area corresponds to two times the standard deviation around the mean (95% confidence region). We can see that the model perfectly interpolates the training data and that the predictive uncertainty increases as we move further away from the observed data.

Since our algorithm is defined in terms of inner products in the input space, it can be lifted into feature space by replacing the inner products with $k(x, x')$; this is often referred to as the *kernel trick*.

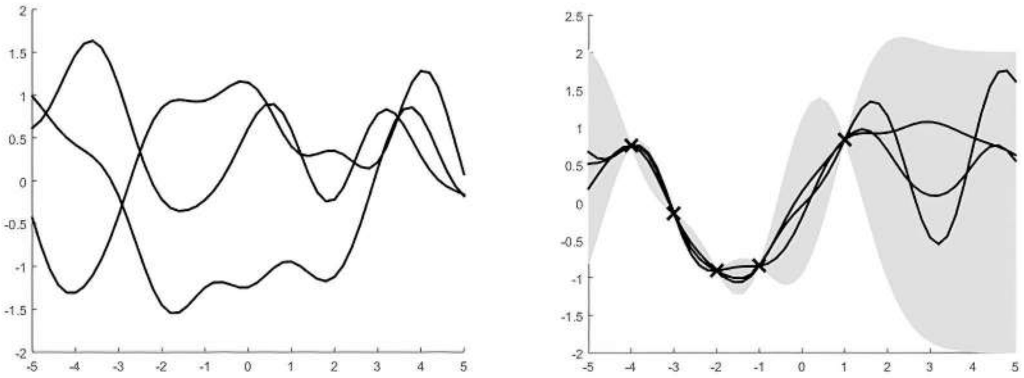

Figure 6.8 Gaussian process regression: Samples from the prior (left) and posterior (right)

The kernel measures similarity between objects, and it doesn't require pre-processing them into feature vector format. For example, a common kernel function is a *radial basis function.*

$$k(x, x') = \exp\left(-\frac{||x - x'||^2}{2\sigma^2}\right) \tag{6.18}$$

In the case of a Gaussian kernel, the feature map lives in an infinite dimensional space. In this case, it is clearly infeasible to explicitly represent the feature vectors.

Regression algorithms help us predict continuous quantities. Based on the nature of data (e.g., linear versus nonlinear relationship between the variables), we can choose either a linear algorithm, such as a Bayesian linear regression or nonlinear KNN regression. We may benefit from a hierarchical model in which certain features are shared among the population. Also, in the case of predicting functional relationships between variables, Gaussian process regression provides the answer to do so. In the following chapter, we will discuss more advanced supervised learning algorithms.

6.6 Exercises

6.1 Compute the runtime and memory complexity of a KNN regressor.

6.2 Derive the Gaussian process (GP) update equations based on the rules for conditioning of multivariate Gaussian random variables.

Summary

- The goal of a regression algorithm is to learn a mapping from inputs x to outputs y, where y is a continuous quantity.
- In Bayesian linear regression defined by $y(x) = w^{\mathrm{T}}x + \epsilon$, we assume the noise term is a zero-mean Gaussian random variable.

- Hierarchical models enable the sharing of features among groups. A hierarchical model assumes there is a common distribution from which individual parameters are sampled and, therefore, captures similarities between groups.
- K nearest neighbors (KNNs) regression is a nonparametric model, in which for a given query data point q, we find its KNN in the training set and compute the average response variable y.
- Gaussian processes (GPs) define a prior over functions that can be updated to a posterior once we have observed data. A GP assumes the function is defined at a finite and arbitrary chosen set of points $x_1, \ldots, x_n$, such that $p(f(x_1), \ldots, f(x_n))$ is jointly Gaussian with a mean of $\mu(x)$ and covariance of $\Sigma(x)$, where $\Sigma_{ij} = \kappa(x_i, x_j)$ and κ is a positive definite kernel function.