10 Principles of Nonlinear Feature Engineering

10.1 Nonlinear regression

In this Section we introduce the general framework of nonlinear regression via *engineering* of nonlinear feature transformations based on our visual intuition, along with several examples. While this sort of nonlinear feature engineering is only feasible with low dimensional datasets (one, two, and three dimensional datesets to be precise), by walking through these examples we flush out a number important concepts in a relatively simple environment that will be omnipresent in our discussion of nonlinear learning going forward.

10.1.1 Modeling principles of nonlinear regression

In Chapter 5 we detailed the basic linear model for regression

$$\text{model}\left(\mathbf{x}, \mathbf{w}\right) = w_0 + x_1 w_1 + \cdots + x_N w_N \tag{10.1}$$

or more compactly

$$\text{model}\left(\mathbf{x}, \mathbf{w}\right) = \mathring{\mathbf{x}}^T \mathbf{w} \tag{10.2}$$

where

$$\mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{10.3}$$

To tune the parameters of our linear model over a generic dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ so that it represents the data (an example of which is shown in the left panel of Figure 10.1) as well as possible, or phrased algebraically so that we have[1]

[1] Here $\mathring{\mathbf{x}}_p^T = \begin{bmatrix} 1 & x_{1,p} & x_{2,p} & \cdots & x_{N,p} \end{bmatrix}$, following the compact notation in Equation (10.3).

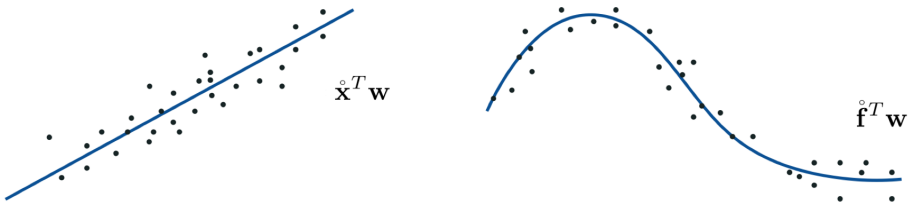

Figure 10.1 (left panel) Linear regression illustrated. Here the fit to the data is defined by the linear model $\mathring{\mathbf{x}}^T\mathbf{w}$. (right panel) Nonlinear regression is achieved by injecting nonlinear feature transformations into our model. Here the fit to the data is a nonlinear curve defined by $\mathring{\mathbf{f}}^T\mathbf{w}$. See text for further details.

$$\mathring{\mathbf{x}}_p^T\mathbf{w} \approx y_p \qquad p = 1, ..., P \tag{10.4}$$

we minimize a proper regression cost function, e.g., the Least Squares cost

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2. \tag{10.5}$$

We can move from *linear* to general *nonlinear* regression, in both its principles and implementation, simply by swapping out the linear model used in the construction of our linear regression with a nonlinear one. For example, instead of using a linear model we can use a nonlinear one involving a single nonlinear function f (e.g., a quadrtic, a sine wave, a logistic function, etc.) that can be parameterized or unparameterized. In the jargon of machine learning such a nonlinear function f is often called a nonlinear *feature transformation* (or just a *feature*) since it transforms our original input features $\mathbf{x}$. Our corresponding nonlinear model would then take the form

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f(\mathbf{x})\, w_1 \tag{10.6}$$

where the set Θ contains all model parameters including the linear combination weights (here w_0 and w_1) as well as potential internal parameters of the function f itself.

We can simply extend this idea to create nonlinear models that use more than just a single nonlinear feature transformation. In general we can form a nonlinear model as the weighted sum of B nonlinear functions of our input, as

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})\, w_1 + f_2(\mathbf{x})\, w_2 + \cdots + f_B(\mathbf{x})\, w_B \tag{10.7}$$

where f_1, f_2, ..., f_B are nonlinear (parameterized or unparameterized) feature transformations, and w_0 through w_B along with any additional weights internal to the nonlinear functions are represented in the weight set Θ.

Regardless of what nonlinear features we choose, the steps we take to formally resolve such a model for the purposes of regression are entirely similar to what we have seen for the simple case of linear regression. In analogy to the linear case, here too it is helpful to write the generic nonlinear model in Equation (10.7) more compactly as

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathring{\mathbf{f}}^T \mathbf{w} \tag{10.8}$$

denoting

$$\mathring{\mathbf{f}} = \begin{bmatrix} 1 \\ f_1\left(\mathbf{x}\right) \\ f_2\left(\mathbf{x}\right) \\ \vdots \\ f_B\left(\mathbf{x}\right) \end{bmatrix} \qquad \text{and} \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_B \end{bmatrix}. \tag{10.9}$$

Once again to tune the parameters of our generic nonlinear model over a dataset of P points so that it represents the data (an example of which is shown in the right panel of Figure 10.1) as well as possible, or phrased algebraically so that we have[2]

$$\mathring{\mathbf{f}}_p^T \mathbf{w} \approx y_p \qquad p = 1, ..., P \tag{10.10}$$

we minimize a proper regression cost function over Θ, e.g., the Least Squares cost

$$g\left(\Theta\right) = \frac{1}{P} \sum_{p=1}^{P} \left(\mathring{\mathbf{f}}_p^T \mathbf{w} - y_p\right)^2. \tag{10.11}$$

Despite all these structural similarities between the linear and nonlinear frameworks, one question still remains: how do we determine the appropriate nonlinear feature transformations for our model, and their number B for a generic dataset? This is indeed one of the most important challenges we face in machine learning, and is one which we will discuss extensively in the current Chapter as well as several of those to come.

10.1.2 Engineering features for nonlinear regression

Here we begin our investigation of nonlinear regression by discussing some simple instances when we can determine the sort and number of nonlinear features we need by *visualizing* the data, and by relying on our own pattern recognition abilities to determine the appropriate nonlinearities. This is an instance of what is more broadly referred to as *feature engineering* wherein the functional

[2] Here $\mathring{\mathbf{f}}_p^T = \begin{bmatrix} 1 & f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_B\left(\mathbf{x}_p\right) \end{bmatrix}$, following the compact notation in Equation (10.9).

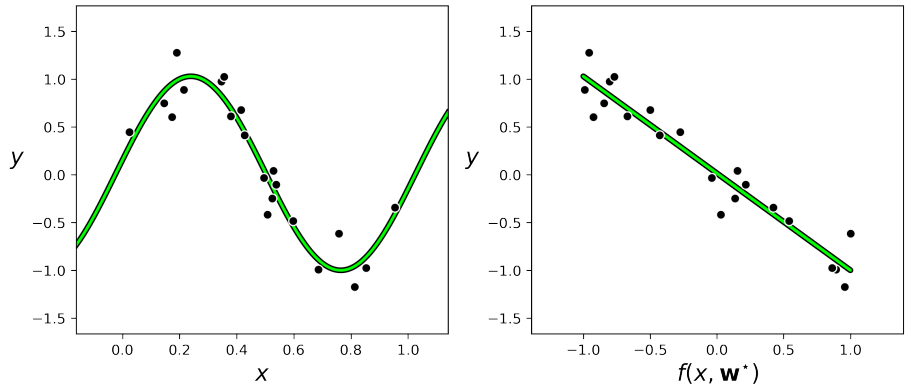

Figure 10.2 Figure associated with Example 10.1. (left panel) A nonlinear regression dataset and corresponding tuned model defined in Equations (10.12) and (10.13). (right panel) The same data and tuned model viewed in the *transformed feature space*. See text for further details.

form of nonlinearities to use in machine learning models is determined (or engineered) by humans through their expertise, domain knowledge, intuition about the problem at hand, etc.

Example 10.1 Modeling a wave

In the left panel of Figure 10.2 we show a nonlinear regression dataset. Because of the wavy appearance of this data we can defensibly propose a nonlinear model consisting of a sine function

$$f(x) = \sin(v_0 + xv_1) \tag{10.12}$$

parameterized by tunable weights v_0 and v_1, with our regression model given as

$$\text{model}(x, \Theta) = w_0 + f(x)\, w_1 \tag{10.13}$$

where $\Theta = \{w_0, w_1, v_0, v_1\}$. Intuitively it seems like this model could fit the data well if its parameters were all properly tuned. In the left panel of Figure 10.2 we show the resulting model fit to the data (in green) by minimizing the Least Squares cost via gradient descent.

With our weights fully tuned, notice that our model is defined *linearly* in terms of its feature transformation. This means that if we plot the transformed version of our dataset, i.e., $\left\{\left(f\left(x_p\right), y_p\right)\right\}_{p=1}^{P}$ wherein the internal feature weights v_0 and v_1 have been optimally tuned, our model fits this transformed data *linearly*, as shown in the right panel of Figure 10.2. In other words in this *transformed*

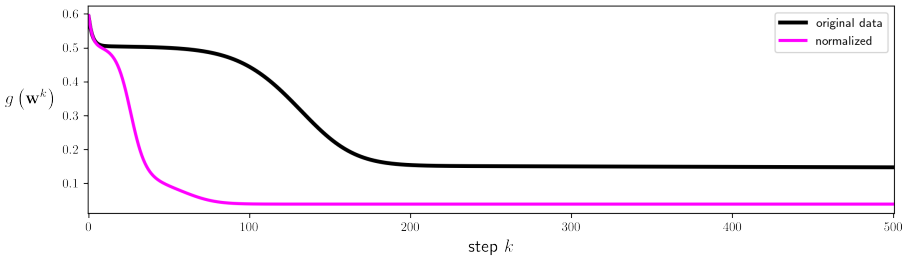

Figure 10.3 Figure associated with Example 10.1. See text for details.

feature space, whose input axis is given by $f(x)$ and whose output is y, our tuned nonlinear model becomes a linear one.

Finally note that as detailed in Section 9.3 for the case of linear regression, with nonlinear regression it is still highly advantageous to employ standard normalization to scale our input when employing gradient descent schemes to minimize a corresponding Least Squares cost. In Figure 10.3 we show the cost function history plots resulting from a run of gradient descent employing the original un-normalized (in black) and standard normalized (in magenta) versions of the input. Comparing the two histories we can see that a significantly lower cost function value found when using the standard normalized input. The convergence behavior displayed by the cost function history plot over the original data is actually somewhat deceiving as it appears to converge (but clearly not to a global minimum).

Example 10.2 Galileo and gravity

In 1638 Galileo Galilei, infamous for his expulsion from the Catholic church for daring to claim that the earth orbited the sun and not the converse (as was the prevailing belief at the time) published his final book: "Discourses and mathematical demonstrations relating to two new sciences." In this book, written as a discourse among three men in the tradition of Aristotle, he described his experimental and philosophical evidence for the notion of uniformly accelerated physical motion. Specifically, Galileo (and others) had intuition that the acceleration of an object due to (the force we now know as) gravity is uniform in time, or in other words that the distance an object falls is directly proportional (i.e., linearly related) to the amount of time it has been traveling, squared. This relationship was empirically solidified using the following ingeniously simple experiment performed by Galileo.

Repeatedly rolling a metal ball down a grooved 5.5 meter long piece of wood set at an incline as shown in Figure 10.4, Galileo timed how long the ball took to get $\frac{1}{4}$, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, and all the way down the wood ramp[3].

[3] Why did Galileo not simply drop the ball from some height and time how long it took to reach certain distances to the ground? Because no reliable way to measure time had yet existed. As a

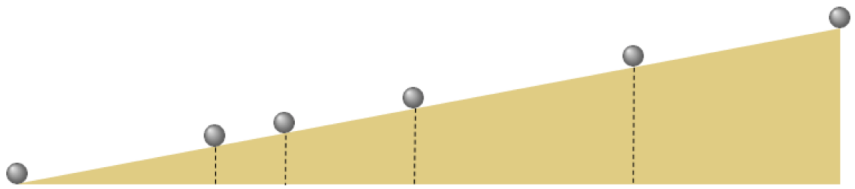

Figure 10.4 Figure associated with Example 10.2. Figurative illustration of Galileo's ramp experiment setup used for exploring the relationship between time and the distance an object falls due to gravity. To perform this experiment he repeatedly rolled a ball down a ramp and timed how long it took to get $\frac{1}{4}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}$, and all the way down the ramp. See text for further details.

Data from a modern reenactment of these experiments (averaged over 30 trials), results in the 6 data points shown in the left panel of Figure 10.5 where the input axis is time (in seconds) while the output is the portion of the ramp traveled by the ball during the experiments. The data here displays a nonlinear quadratic relationship between its input and output. This translates to using the quadratic model

$$\text{model}(x, \Theta) = w_0 + f_1(x)\, w_1 + f_2(x)\, w_2 \tag{10.14}$$

with two *unparameterized* feature transformations: the identity transformation $f_1(x) = x$ and the quadratic transformation $f_2(x) = x^2$. Replacing $f_1(x)$ and $f_2(x)$ in Equation (10.14) with x and x^2 gives the familiar quadratic form $w_0 + xw_1 + x^2 w_2$.

After standard normalizing the input of this dataset (see Section 9.3) we minimize the corresponding Least Squares cost via gradient descent, and plot the corresponding best nonlinear fit on the original data in the left panel of Figure 10.5. Since this model is a linear combination of its two feature transformations (plus a bias weight) we can also visualize its corresponding *linear fit* in the transformed feature space, as shown in the right panel of Figure 10.5. In this space our the input axes are given by $f_1(x)$ and $f_2(x)$ respectively, and our transformed points by the triple $\left(f_1\left(x_p\right), f_2\left(x_p\right), y\right)$.

10.1.3 `Python` implementation

Below we show a universal way to implement the generic nonlinear model shown in Equation 10.8, generalizing our original linear implementation from Section **??**.

result he had to use a *water clock* for his ramp experiments! Interestingly, Galileo was the one who set humanity on the route towards its first reliable time-piece in his studies of the pendulum.

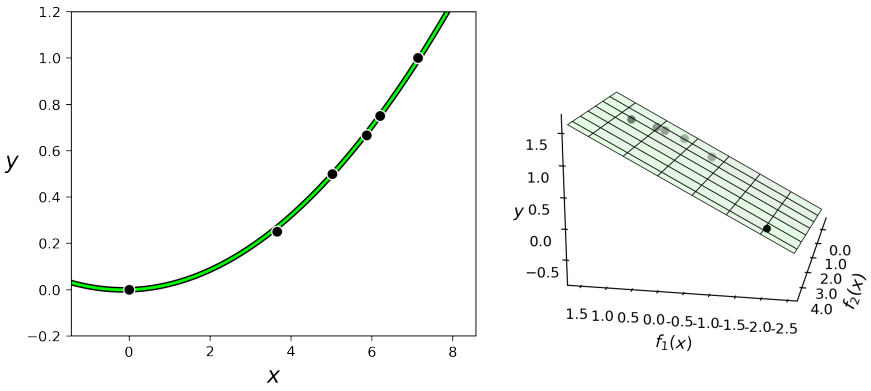

Figure 10.5 Figure associated with Example 10.2. (left panel) Data from a modern reenactment of a famous experiment performed by Galileo along with a well tuned quadratic model. (right panel) This datasest and model viewed in the feature transformed space. See text for further details.

```
# an implementation of our model employing a nonlinear feature
    transformation
def model(x,w):
    # feature transformation
    f = feature_transforms(x,w[0])

    # compute linear combination and return
    a = w[1][0] + np.dot(f.T,w[1][1:])
    return a.T
```

Here our generic set of engineered feature transformations are implemented in the `Python` function `feature_transforms`, and differ depending on how the features themselves are defined. Here we have implemented this function as generically as possible, to encompass the case where our desired feature transformations have internal parameters. That is, we package the model weights in the set Θ as `w`, which is a list containing the *internal weights* of `feature_transforms` in its first entry `w[0]`, and the weights in the final linear combination of the model stored in second etnry `w[1]`).

e.g., the `feature_transforms` function employed in Example 10.1 above can be implemented as follows

```
# the feature transformation from Example 2
def feature_transforms(x,w):
    # calculate feature transform
    f = np.sin(w[0] + np.dot(x.T,w[1:])).T
    return f
```

If our desired feature transformations do not have internal parameters we can

either leave the parameter input to this function empty, or implement the model above slightly differently by computing our set of feature transformations as

```
1 f = feature_transforms(x)
```

and computing the linear combination of these transformed features in line 7 as

```
1   a = w[0] + np.dot(f.T,w[1:])
```

In either case, in order to successfully perform nonlinear regression we can focus our attention solely on implementing the function `feature_transforms` - employing the `autograd`'-wrapped `numpy` library if we wish to employ automatic differentiation (see Section **??**). Nothing about how we implement our *regression cost functions* changes from the original context of linear regression detailed in Chapter 5. In other words, once we have implemented a given set of feature transformations correctly, employing the `model` above we can then tune the parameters of our nonlinear regression precisely as we have done in Chapter 5 employing any regression cost function and local optimization scheme. The only caveat one must keep in mind is that when employing *parameterized* models (like the one in Example 10.1) the corresponding cost functions are generally *non-convex*. Thus either zero or first order methods should be applied, or second order methods adjusted in the manner detailed in Section 4.5.

10.2 Nonlinear multi-output regression

In this Section we present a description of nonlinear feature engineering for multi-ouput regression first introduced Section 5.5. This mirrors what we have seen in the previous Section completely with one small but important difference: in the multi-output case we can choose to model each regression *separately*, employing one nonlinear model for output, or *jointly*, producing a single nonlinear model for all outputs simultaneously.

10.2.1 Modeling principles of nonlinear multi-output regression

With linear mulit-output regression we construct C linear models of the form $\mathring{\mathbf{x}}^T\mathbf{w}_c$ or equivalently one joint linear model including all C regressions by stacking the weight vectors $\mathbf{w}_c$ column-wise into an $(N+1)\times C$ matrix $\mathbf{W}$ (see Section 5.5.1) and forming the multi-output linear model

$$\text{model}\left(\mathbf{x},\mathbf{W}\right) = \mathring{\mathbf{x}}^T\mathbf{W}\,. \tag{10.15}$$

Given a dataset of P points $\left\{\left(\mathbf{x}_p, \mathbf{y}_p\right)\right\}_{p=1}^{P}$, where each paired input $\mathbf{x}_p$ and output $\mathbf{y}_p$ is N and C dimensional respectively, we aim to tune the parameters of $\mathbf{W}$ to learn a linear relationship between the input and output as

$$\mathring{\mathbf{x}}_p^T\mathbf{W} \approx \mathbf{y}_p \qquad p = 1, ..., P \tag{10.16}$$

by minimizing an appropriate cost function, e.g., the Least Squares cost

$$g\left(\mathbf{W}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathring{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_2^2. \tag{10.17}$$

With multi-output regression the move from linear to nonlinear modeling closely mirrors what we saw in the previous Section. That is, for the c^{th} regression problem we construct a model using (in general) B_c nonlinear feature transformations as

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_{c,1}\left(\mathbf{x}\right)w_{c,1} + f_{c,2}\left(\mathbf{x}\right)w_{c,2} + \cdots + f_{c,B_c}\left(\mathbf{x}\right)w_{c,B_c} \tag{10.18}$$

where $f_{c,1}$, $f_{c,2}$, ... f_{c,B_c} are nonlinear (potentially parameterized) functions and $w_{c,0}$ through w_{c,B_c} (along with any additional weights internal to the nonlinear functions) are represented in the weight set Θ_c.

To simplify the chore of choosing nonlinear features for each regressor we can instead choose a single set of nonlinear feature transformations and *share* them among all C regression models. If we choose the same set of B nonlinear features for all C models the c^{th} model takes the form

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_1\left(\mathbf{x}\right)w_{c,1} + f_2\left(\mathbf{x}\right)w_{c,2} + \cdots + f_B\left(\mathbf{x}\right)w_{c,B} \tag{10.19}$$

where Θ_c now contains both the linear combination weights $w_{c,0}$, ...$w_{c,B}$ as well as any weights internal to the shared feature transformations. Note the only parameters unique to the c^{th} model are the linear combination weights since every model shares any weights internal to the feature transformations. Employing the same compact notation for our feature transformations as in Equation (10.9) we can express each of these models more compactly as

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = \mathring{\mathbf{f}}^T\mathbf{w}_c. \tag{10.20}$$

Figure 10.6 shows a prototypical multi-output regression using this notation.

We can then express all C models together by stacking all C weight vectors $\mathbf{w}_c$ column-wise into a $(B+1) \times C$ weight matrix $\mathbf{W}$, giving the joint model as

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathring{\mathbf{f}}^T\mathbf{W}. \tag{10.21}$$

This is a direct generalization of the original linear model shown in Equation

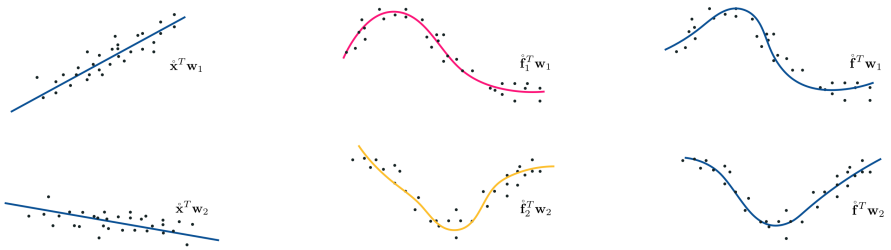

Figure 10.6 Figurative illustrations of multi-output regression with $C = 2$ outputs. (left panel) Linear multi-output regression. (middle panel) Nonlinear multi-output regression where each regressor uses its own distinct nonlinear feature transformation. (right panel) Nonlinear multi-output regression where both regressors share the same nonlinear feature transformations. See text for further details.

(10.15), and the set Θ contains the linear combination weights $\mathbf{W}$ as well as any parameters internal to our feature transformations themselves. To tune the weights of our joint model so that it represents our dataset as well as possible, or phrased algebraically so that we have

$$\mathring{\mathbf{f}}_p^T\mathbf{W} \approx \mathbf{y}_p \qquad p = 1, ..., P \tag{10.22}$$

we minimize an appropriate regression cost of this model over the parameters in Θ, e.g., the Least Squares cost[4]

$$g(\Theta) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathring{\mathbf{f}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_2^2. \tag{10.23}$$

10.2.2 Engineering features for nonlinear multi-output regression

With multi-output regression, determining appropriate features by visual inspection is more challenging than the basic instance of regression detailed in the previous Section. Here we provide one relatively simple example of this sort of feature engineering to give a flavor of this challenge and the nonlinear modeling involved.

[4] Note that if these feature transformations contain no internal parameters (e.g., polynomial functions) then each individual regression model can be tuned separately. However when employing *parameterized* features (e.g., neural networks) then the cost function does not decompose over each regressor and we must tune all of our model parameters *jointly*, that is we must learn all C regressions *simultaneously*. This differs from the linear case, where tuning the parameters of the linear model one regressor at-a-time or simultaneously returns the same result (see Section 5.5.2).

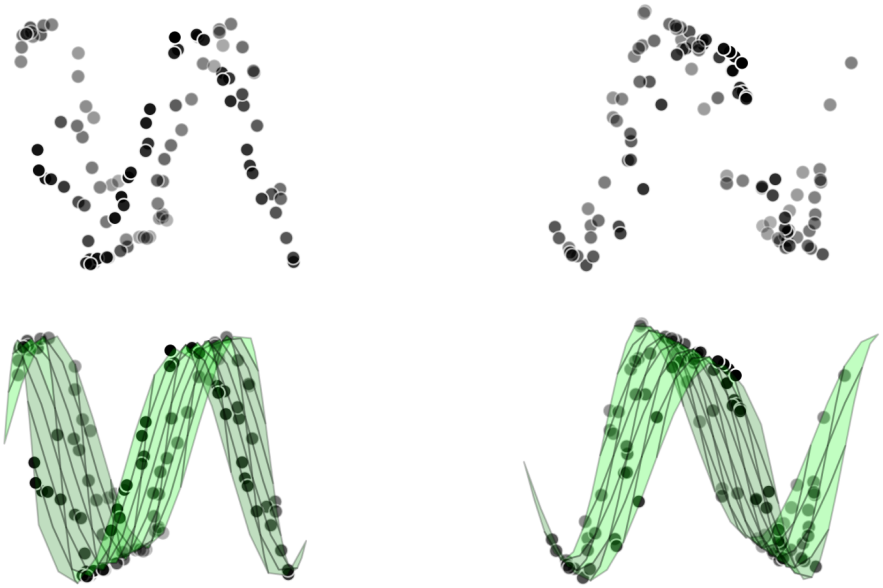

Figure 10.7 Figure associated with Example 7. See text for details.

Example 10.3 **Modeling multiple waves**

In the top panels of the Figure 10.7 we plot a multi-ouptput regression dataset with $N = 2$ dimensional inputs and $C = 2$ outputs, where the input paired with the first and second outputs are shown in the left and right panel, respectively. Both instances appear to be *sinusoidal* in nature, with each having its own unique shape.

From visual examination of the data we can reasonably choose to model both regressions simultaneously using $B = 2$ parameterized sinusoidal feature transformations

$$\begin{aligned} f_1(\mathbf{x}) &= \sin(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2) \\ f_2(\mathbf{x}) &= \sin(w_{2,0} + w_{2,1}x_1 + w_{2,2}x_2) \end{aligned} \tag{10.24}$$

Fitting this set of nonlinear features jointly by minimizing the Least Squares cost in Equation (10.23) using gradient descent results in the fits shown in the bottom panels of Figure 10.7.

10.2.3 `Python` implementation

As with the linear case, detailed in Section 5.5.3, here likewise we can piggy-back on our general `Pythonic` implementation of nonlinear regression introduced in Section 10.1.3 and employ precisely the same model and cost function implementation as used in the single-output case. The only difference here is in how

how we define our feature transformations and the dimension of our matrix of linear combination weights.

10.3 Nonlinear two-class classification

In this Section we introduce the general framework of nonlinear classification, along with a number of elementary examples. As in the Prior Sections, these examples are all low dimensional, allowing us to visually examine patterns in the data and propose appropriate nonlinearities, which we can inject into our linear supervised paradigm to produce nonlinear classifications. In doing this we are essentially performing nonlinear feature engineering for the two-class classification paradigm.

10.3.1 Modeling principles of nonlinear two-class classification

While we employed a *linear* model in deriving linear two-class classification in Chapter 6, this linearity was simply an *assumption* about the sort of boundary that (largely) separates the two classes of data. Employing by default label values $y_p \in \{-1, +1\}$ and expressing our linear model algebraically as

$$\text{model}\left(\mathbf{x}, \mathbf{w}\right) = \mathring{\mathbf{x}}^T\mathbf{w} \tag{10.25}$$

our linear decision boundary then consists of all input points $\mathbf{x}$ where $\mathring{\mathbf{x}}^T\mathbf{w} = 0$. Likewise label predictions are made (see Section 7.6) as

$$y = \text{sign}\left(\mathring{\mathbf{x}}^T\mathbf{w}\right). \tag{10.26}$$

To tune $\mathbf{w}$ we then minimize a proper two-class classification cost function, e.g., the two-class softmax (or cross-entropy) cost

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}}\right). \tag{10.27}$$

We can adjust this framework to jump from linear to nonlinear classification in an entirely similar way we did with regression in Section 10.1. That is, we can swap out our linear model with a general nonlinear one of the generic form

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right)w_1 + f_2\left(\mathbf{x}\right)w_2 + \cdots + f_B\left(\mathbf{x}\right)w_B \tag{10.28}$$

where f_1, f_2, ..., f_B are nonlinear parameterized or unparameterized functions and w_0 through w_B (along with any additional weights internal to the nonlinear functions) are represented in the weight set Θ. Just as with regression, here too we can express this more compactly (see Section 10.1.1) as

Figure 10.8 Figurative illustrations of two-class linear and nonlinear classification. (left panel) Linear two-class classification, where the separating boundary is defined by $\mathring{\mathbf{x}}^T\mathbf{w} = 0$. (right panel) Nonlinear two-class classification, where the separating boundary is defined as $\mathring{\mathbf{f}}^T\mathbf{w} = 0$. See text for further details.

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathring{\mathbf{f}}^T\mathbf{w}. \tag{10.29}$$

In complete analogy to the linear case, our decision boundary here consists of all inputs $\mathbf{x}$ where $\mathring{\mathbf{f}}^T\mathbf{w} = 0$, and likewise predictions are made as

$$y = \text{sign}\left(\mathring{\mathbf{f}}^T\mathbf{w}\right). \tag{10.30}$$

Figure 10.8 shows a prototypical two-class classification using this notation.

Finally, in order to tune the parameters in Θ we must minimize a proper cost function with respect to it, e.g., the two-class softmax cost (again in complete analogy to Equation (10.27))

$$g\left(\Theta\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\mathring{\mathbf{f}}_p^T\mathbf{w}}\right). \tag{10.31}$$

10.3.2 Engineering features for nonlinear two-class classification

With low dimensional datasets we can, in certain instances, fairly easily engineer a proper set of nonlinear features for two-class classification by examining the data visually. Below we explore two such examples.

Example 10.4 **When the decision boundary is just two single points!**

In discussing classification through the lens of logistic regression in Section 6.3 we saw how linear classification can be thought of as a specific instance of nonlinear regression. In particular we saw how from this perspective we aim at fitting a curve (or surface in higher dimensions) that consists of a linear combination of our input shoved through the `tanh` function. For $N = 1$ dimensional datasets, like the one shown in the left column of Figure 10.9, this results in learning a decision boundary that is defined by a *single* point.

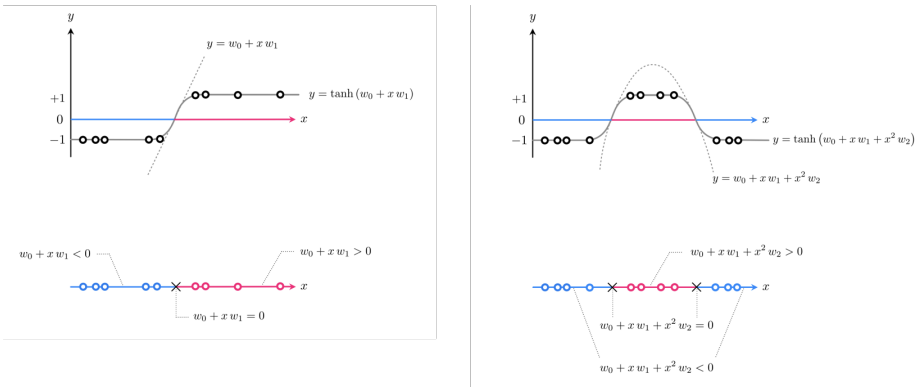

Figure 10.9 Figure associated with Example 10.4. (left column) A prototypical $N = 1$ dimensional linear classification dataset with fully tuned linear model shown from the regression perspective (top left panel), and from the perceptron perspective (bottom left panel) where label values are encoded as colors (red for +1 and blue for −1). (right column) A simple classification dataset that requires a decision boundary consisting of two points, something a linear model cannot provide. As can be seen here, a quadratic model can achieve this goal (provided its parameters are tuned appropriately).

However a linear decision boundary is quite inflexible in general, and fails to provide good separation even in the case of the simple example shown in the right column of Figure 10.9. For such a dataset we clearly need a model that is capable of crossing the input space (the x axis) *twice* at points separated by some distance, something a linear model can never do.

What sort of simple function crosses the horizontal axis twice? A quadratic function can. If adjusted to the right height a quadratic certainly can be made to cross the horizontal axis twice and, when passed through a `tanh` function, could indeed give us the sort of predictions we desire (as illustrated in the right column of Figure 10.9).

A quadratic model takes the form

$$\text{model}(x, \mathbf{w}) = w_0 + xw_1 + x^2 w_2 \tag{10.32}$$

which uses two feature transformations: the identity $f_1(x) = x$ and the quadratic transformation $f_2(x) = x^2$, with the weight set Θ only containing the weights w_0, w_1, and w_2 in $\mathbf{w}$.

In the left panel of Figure 10.10 we illustrate a toy dataset like the one shown in the right column of Figure 10.9. We also show in green, the result of fully tuning the quadratic model in Equation (10.32) by minimizing (via gradient descent) the corresponding two-class Softmax in Equation (10.31). In the right panel we show the same dataset only in the *transformed feature space* defined by our two features (as first detailed in Examples 10.1 and 10.2) wherein the nonlinear decision boundary becomes *linear*. This finding is true in general: a well

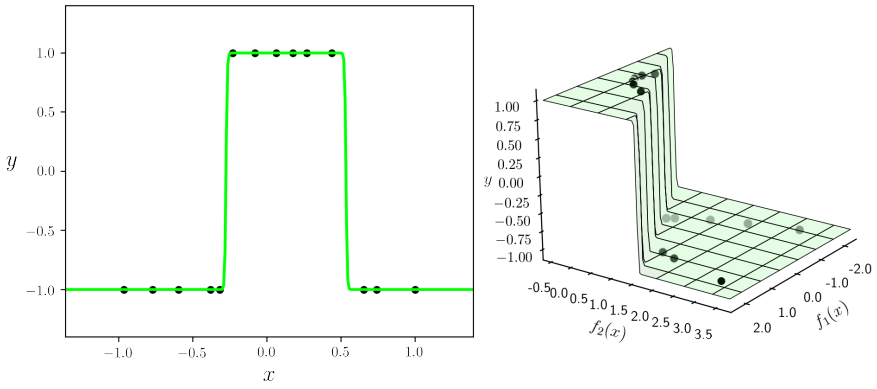

Figure 10.10 Figure associated with Example 10.4. See text for details.

separating *nonlinear* decision boundary in the original space of a dataset translates to a well-separating *linear* decision boundary in the transformed feature space. This is analogous to the case of regression, as detailed in Section 10.1.2, where a good *nonlinear* fit in an original space corresponds to a good *linear* fit in the transformed feature space.

Example 10.5 An elliptical decision boundary

In the left column of Figure 10.11 we show an $N = 2$ dimensional dataset, shown from the perceptron perspective in the top panel and from the regression perspective in the bottom panel.

Visually examining the dataset it appears that some sort of elliptical decision boundary centered at the origin, defined by

$$\text{model}(\mathbf{x}, \Theta) = w_0 + x_1^2 w_1 + x_2^2 w_2 \tag{10.33}$$

might do a fine job of classification. Parsing this formula we can see that we have used two feature transformations, i.e., $f_1(\mathbf{x}) = x_1^2$ and $f_2(\mathbf{x}) = x_2^2$, with the parameter set $\Theta = \{w_0, w_1, w_2\}$.

Minimizing the softmax cost in Equation (10.31) using this model (via gradient descent) we show the resulting nonlinear decision boundary layered over the dataset in the middle column of Figure 10.11, from the perceptron perspective in the top panel and from the regression perspective in the bottom panel. Finaly, in the right column we show the data in the *transformed feature space* along with the corresponding linear decision boundary.

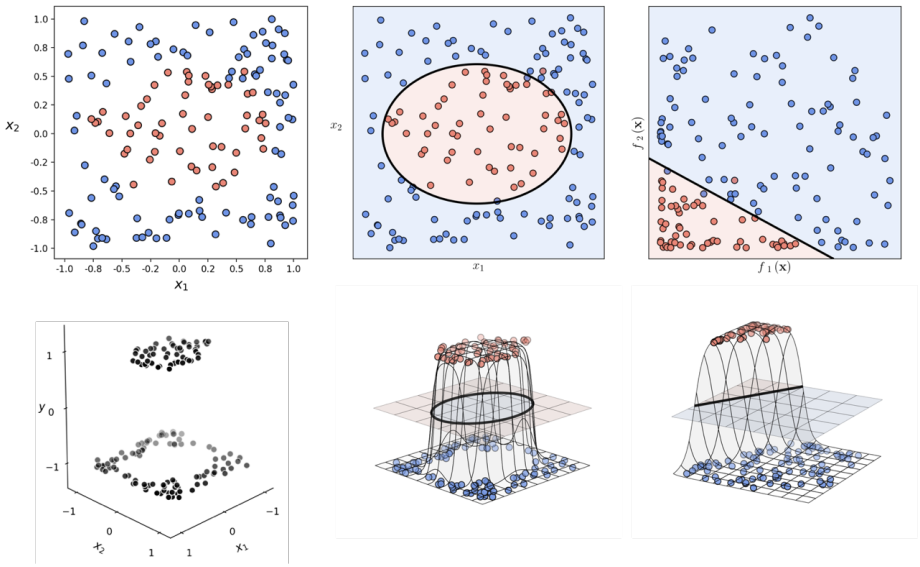

Figure 10.11 Figure associated with Example 10.5. See text for details.

10.3.3 Python implementation

The general nonlinear model in Equation 10.29 can - in general - be implemented precisely as described in Section 10.1.3 since indeed it is the same general nonlinear model we use with nonlinear regression. Therefore, just as with regression, the we need not alter the implementation of any two-class classification cost function introduced in Chapter 6 to perform nonlinear classification: all we need to do is properly define our nonlinear transformation(s) in `Python` (if we wish to use an automatic differentiator then these should be expressed using `autograd`'s `numpy` wrapper).

For example, one way to implement the feature transformation used in Example 10.5 is shown below.

```
# feature transformation shown in Example 3
def feature_transforms(x):
    # calculate feature transform
    f = x**2
    return f
```

10.4 Nonlinear multi-class classification

In this Section we present the general nonlinear extension of linear multi-class classification first introduced Chapter 7. This mirrors what we have seen in the

previous Sections very closely, and is in particular almost entirely similar to the discussion of nonlinear multi-output regression in Section 10.2.

10.4.1 Modeling principles of nonlinear multi-class classification

As we saw in Chapter 7, with linear mulit-class classification we construct C linear models of the form $\mathring{\mathbf{x}}^T\mathbf{w}_c$, which we may represent jointly by stacking the weight vectors $\mathbf{w}_c$ column-wise into an $(N+1)\times C$ matrix $\mathbf{W}$ (see Section 7.3.9) and forming a single multi-output linear model of the form

$$\text{model}\left(\mathbf{x},\mathbf{W}\right) = \mathring{\mathbf{x}}^T\mathbf{W}\,. \tag{10.34}$$

Given a dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, where each input $\mathbf{x}_p$ is N dimensional and each y_p is a label value in the set $y_p \in \{0,...,C-1\}$, we aim to tune the parameters of $\mathbf{W}$ to satisfy the fusion rule

$$y_p = \text{argmax}\left[\,\text{model}\left(\mathbf{x}_p,\mathbf{W}\right)\right] \qquad p = 1,...,P \tag{10.35}$$

by either tuning each column of $\mathbf{W}$ one-at-a-time in a one-versus-all approach (see Section 7.2) or by minimizing an appropriate cost, e.g., multi-class softmax cost

$$g\left(\mathbf{W}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_c}\right) - \mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right]. \tag{10.36}$$

over the entire matrix $\mathbf{W}$ simultaneously (see Section 7.3).

With multi-class classification the move from linear to nonlinear modeling very closely mirrors what we saw in the case of multi-output regression in Section 10.2. That is, for the c^{th} classifier we can construct a model using (in general) B_c nonlinear feature transformations as

$$\text{model}_c\left(\mathbf{x},\Theta_c\right) = w_{c,0} + f_{c,1}\left(\mathbf{x}\right)w_{c,1} + f_{c,2}\left(\mathbf{x}\right)w_{c,2} + \cdots + f_{c,B_c}\left(\mathbf{x}\right)w_{c,B_c} \tag{10.37}$$

where $f_{c,1}$, $f_{c,2}$, ..., f_{c,B_c} are nonlinear parameterized or unparameterized functions and $w_{c,0}$ through w_{c,B_c} (along with any additional weights internal to the nonlinear functions) are represented in the weight set Θ_c.

To simplify the chore of choosing nonlinear features for each classifier we can instead choose a *single* set of nonlinear feature transformations and *share* them among all C two-class models. If we choose the same set of B nonlinear features for all C models the c^{th} model takes the form

$$\text{model}_c\left(\mathbf{x},\Theta_c\right) = w_{c,0} + f_1\left(\mathbf{x}\right)w_{c,1} + f_2\left(\mathbf{x}\right)w_{c,2} + \cdots + f_B\left(\mathbf{x}\right)w_{c,B} \tag{10.38}$$

where Θ_c now contains both the linear combination weights $w_{c,0}$, ...$w_{c,B}$ as

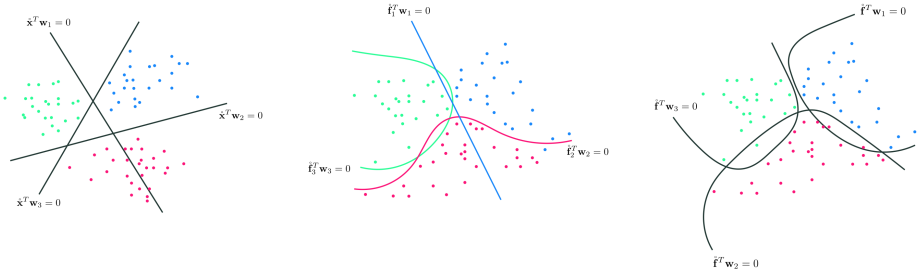

Figure 10.12 Figurative illustrations of multi-class classifiers on a generic dataset with $C = 3$ classes. (left panel) Linear multi-class classification. (middle panel) Nonlinear multi-class classification where each classifier uses its own distinct nonlinear feature transformation. (right panel) Nonlinear multi-class classification where all classifiers share the same nonlinear feature transformations. See text for further details.

well as any weights internal to the shared feature transformations. Note the only parameters unique to the c^{th} model are the linear combination weights since every model shares any weights internal to the feature transformations. Employing the same compact notation for our feature transformations as in Equation (10.9) we can express each of these models more compactly as

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = \mathring{\mathbf{f}}^T\mathbf{w}_c. \tag{10.39}$$

Figure 10.12 shows a prototypical multi-class classification using this notation.

We can then tune the parameters of each of these models individually, taking a one-versus-all approach, or simultaneously by minimizing a single joint cost function over all of them together. To perform the latter approach it is helpful to first re-express all C models together by stacking all C weight vectors $\mathbf{w}_c$ column-wise into a $(B+1) \times C$ weight matrix $\mathbf{W}$, giving the joint model as

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathring{\mathbf{f}}^T\mathbf{W}. \tag{10.40}$$

where the set Θ contains the linear combination weights in $\mathbf{W}$ as well as any parameters internal to our feature transformations themselves. To tune the weights of our joint model so that the fusion rule

$$y_p = \text{argmax}\left[\,\text{model}\left(\mathbf{f}_p, \mathbf{W}\right)\right] \qquad p = 1, ..., P \tag{10.41}$$

holds as well as possible we minimize an appropriate multi-class cost of this model over the parameters in Θ, e.g., the multi-class softmax cost

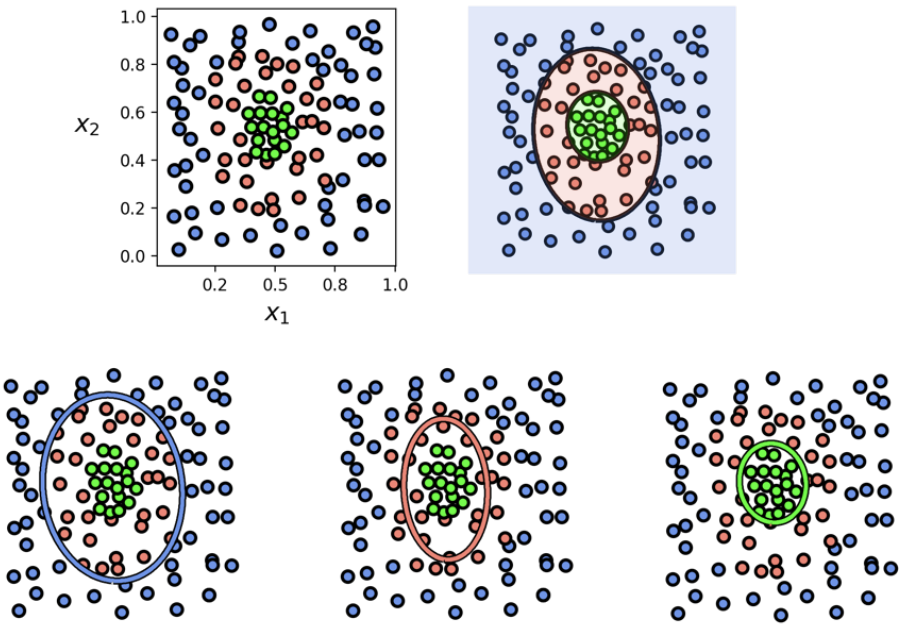

Figure 10.13 Figure associated with Example 10.6. See text for details.

$$g\left(\Theta\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\mathring{\mathbf{f}}_p^T\mathbf{w}_c}\right) - \mathring{\mathbf{f}}_p^T\mathbf{w}_{y_p}\right]. \tag{10.42}$$

10.4.2 Engineering features for nonlinear multi-class classification

Determining nonlinear features via visual analysis is even more challenging in the multi-class setting than in the instance of two-class classification detailed in the previous Section, as there are even more feature transformations to define ourselves. Here we provide a simple example of this sort of feature engineering, but in general we will want to *learn* such feature transformations automatically (as we begin to detail in the next Chapter).

Example 10.6 Elliptical boundaries

In this Example we engineer nonlinear features to perform multi-class classification on the dataset shown in the top left panel of Figure 10.13, which consists of $C = 3$ classes that appear to be (roughly) separable by elliptical boundaries. Here the points colored blue, red, and green have label values 0, 1, and 2, respectively.

Because the data is not centered at the origin we must use a full degree two

polynomial expansion of the input consisting of features of the form $x_1^i x_2^j$ where $i + j \leq 2$. This gives the degree two polynomial model

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + x_1 w_1 + x_2 w_2 + x_1 x_2 w_3 + x_1^2 w_4 + x_2^2 w_5. \tag{10.43}$$

Using this nonlinear model we minimize the mulit-class softmax in Equation (10.42) using gradient descent and plot the corresponding fused multi-class boundary in the top right panel of Figure 10.13 where each region is colored according to the prediction made by the final classifier.

We also plot the resulting two-class boundary produced by each individual classifier in the bottom row of Figure 10.13, coloring each boundary according to the one-versus-all classification performed in each instance. Here, as in the linear case outlined in Section 7.3, we can see that while each two-class sub-problem cannot be solved correctly, when fused via Equation (10.4.1) the resulting multi-class classification can still be very good.

10.4.3 `Python` implementation

The general nonlinear model in Equation 10.40 above can be implemented as described in Section 10.2.3. since it is the same general nonlinear model we use with nonlinear multi-output regression. Therefore, just as with multi-output regression, we need not alter the implementation of a joint nonlinear multi-class classification cost functions introduced in Chapter 7: all we need do is properly define our nonlinear transformation(s) in `Python` (if we wish to use an automatic differentiator then these should be expressed using `autograd`'s `numpy` wrapper).

10.5 Nonlinear unsupervised learning

In this Section we discuss the general nonlinear extension of our fundamental unsupervised learning technique: the Autoencoder, introduced in Section 8.2.

10.5.1 Modeling principles of the nonlinear Autoencoder

In Section 8.2 we described linear Autoencoder, an elegant way to determine the best linear subspace to represent a set of mean-centered N dimensional input datapoints $\left\{\mathbf{x}_p\right\}_{p=1}^P$. To determine the projection of our data onto the K dimensional subspace spanned by the columns of an $N \times K$ matrix $\mathbf{C}$, we first *encode* our data on the subspace using the encoder model

$$\text{model}_e\left(\mathbf{x}, \mathbf{C}\right) = \mathbf{C}^T\mathbf{x} \tag{10.44}$$

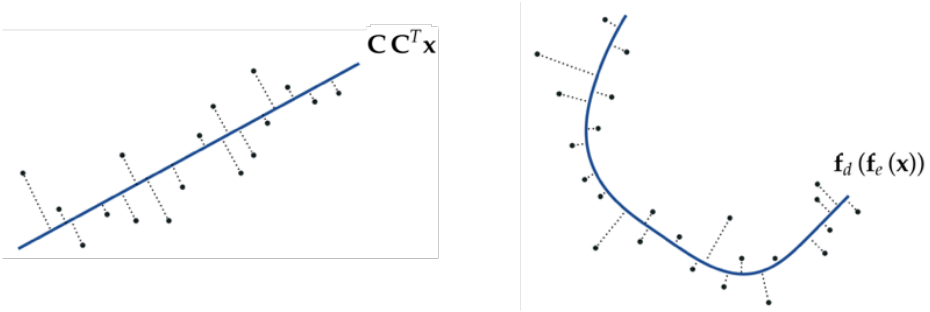

Figure 10.14 Figurative illustrations of a linear (left panel) and nonlinear (right panel) Autoencoder. See text for further details.

which takes in an N dimensional input and returns a K dimensional output. We then *decode* to produce the projection onto the subspace as

$$\text{model}_\text{d}\,(\mathbf{v}, \mathbf{C}) = \mathbf{C}\mathbf{v}. \tag{10.45}$$

The output of the decoder, being a projection onto a subspace lying in the original N dimensional space, is itself N dimensional.

The composition of these two steps gives the linear Autoencoder model

$$\text{model}\,(\mathbf{x}, \mathbf{C}) = \text{model}_\text{d}\,(\,\text{model}_\text{e}\,(\mathbf{x}, \mathbf{C})\,, \mathbf{C}) = \mathbf{C}\,\mathbf{C}^T\mathbf{x} \tag{10.46}$$

that, when $\mathbf{C}$ is tuned correctly, produces a linear subspace that represents the data extremely well

$$\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p \approx \mathbf{x}_p \qquad p = 1, ..., P \tag{10.47}$$

or equivalently, has an effect on input as close to the identity transformation as possible

$$\mathbf{C}\mathbf{C}^T \approx \mathbf{I}_{N\times N}. \tag{10.48}$$

In order to recover this ideal setting for $\mathbf{C}$ we can minimize, e.g., the Least Squares error measurement of the desired effect in Equation (10.47), as

$$g\,(\mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2. \tag{10.49}$$

To introduce nonlinearity here, i.e., to determine a nonlinear surface (also called a *manifold*) to project the data onto (as illustrated in Figure 10.14) we can simply replace the linear encoder/decoder models in Equations (10.44) and (10.45) with nonlinear versions of the generic form we have seen in previous Sections

$$\begin{aligned}\text{model}_{\text{e}}\left(\mathbf{x}, \Theta_e\right) &= w_0^e + f_1^e\left(\mathbf{x}\right) w_1^e + f_2^e\left(\mathbf{x}\right) w_2^e + \cdots + f_{B_e}^e\left(\mathbf{x}\right) w_{B_e}^e \\ \text{model}_{\text{d}}\left(\mathbf{v}, \Theta_d\right) &= w_0^d + f_1^d\left(\mathbf{v}\right) w_1^d + f_2^d\left(\mathbf{v}\right) w_2^d + \cdots + f_{B_d}^d\left(\mathbf{v}\right) w_{B_d}^d\end{aligned} \tag{10.50}$$

Here, $f_1^e, f_2^e, \ldots, f_{B_e}^e$ are the encoder feature transformations, with Θ_e denoting the parameter set for the encoding model. Similarly, $f_1^d, f_2^d, \ldots, f_{B_d}^d$ are the decoder feature transformations, with Θ_d denoting the parameter set for the decoding model.

For the sake of notational consistency, and since each model outputs (in general) a vector value, we also denote the encoder and decoder models as $\text{model}_{\text{e}}\left(\mathbf{x}, \Theta_e\right) = \mathbf{f}_e\left(\mathbf{x}\right)$ and $\text{model}_{\text{d}}\left(\mathbf{v}, \Theta_d\right) = \mathbf{f}_d\left(\mathbf{v}\right)$, respectively. With this notation we can write the general nonlinear Autoencoder model simply as

$$\text{model}\left(\mathbf{x}, \Theta\right) = \mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}\right)\right) \tag{10.51}$$

where the parameter set Θ now contains all parameters of both Θ_e and Θ_d.

As with the linear version, here our aim is to properly design the encoder/decoder pair and tune the parameters of Θ properly in order to determine the appropriate nonlinear manifold for the data, as

$$\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}_p\right)\right) \approx \mathbf{x}_p \qquad p = 1, ..., P \tag{10.52}$$

or equivalently, that the nonlinear Autoencoder model in Equation (10.51) produces the identity

$$\mathbf{f}_d\left(\mathbf{f}_e\right) \approx \mathbf{I}_{N\times N} \tag{10.53}$$

as well as possible. To tune the parameters in Θ we can minimize, e.g., the Least Squares error measurement of the desired effect in Equation (10.52), as

$$g\left(\Theta\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}_p\right)\right) - \mathbf{x}_p\right\|_2^2. \tag{10.54}$$

10.5.2 Engineering features for nonlinear unsupervised learning

Given that both encoder and decoder models contain nonlinear features that must be determined, and the compositional manner in which the model is formed in Equation (10.51), engineering features by visual analysis can be difficult even with an extremely simple example, which we provide here.

Example 10.7 A circular manifold

In this example we use a simulated dataset of $P = 20$ two-dimensional data

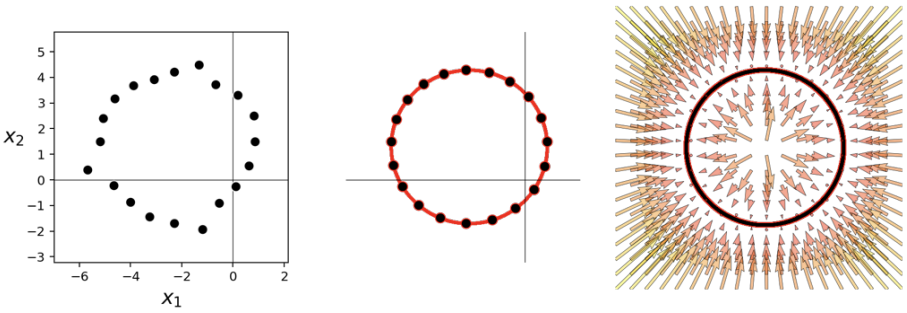

Figure 10.15 Figure associated with Example 10.7. (left panel) Original data distributed roughly on a circle. (middle panel) The final decoding of the original data onto the determined circular manifold. (right panel) A projection mapping showing how points in the nearby space are projected onto the final learned manifold.

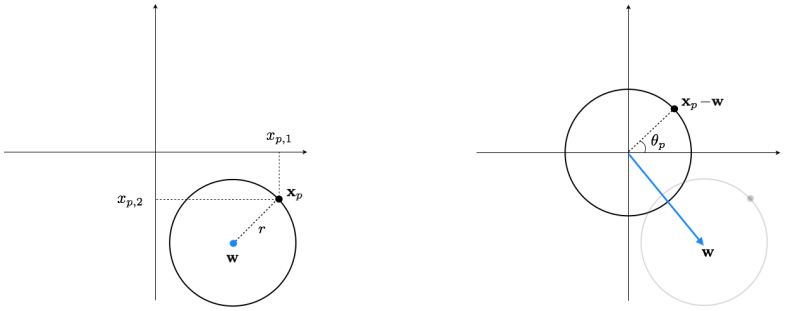

Figure 10.16 Figure associated with Example 10.7. (left panel) A circle in two dimensions is characterized by its center $\mathbf{w}$ and its radius r. (right panel) when centered at the origin, any point on the circle can be represented using the angle created between its connector to the origin and the horizontal axis.

points to learn a circular manifold via the nonlinear Autoencoder scheme detailed in the current Section. This dataset is displayed in the top left panel of Figure 10.15, where we can see it has an almost perfect circular shape.

To engineer a nonlinear Autoencoder model for this dataset, recall that a circle in two dimensions, as illustrated in the left panel of Figure 10.16, can be fully characterized using its center point $\mathbf{w}$ and radius r. Subtracting off $\mathbf{w}$ from any point $\mathbf{x}_p$ on the circle then centers the data at origin, as shown in the right panel of Figure 10.16.

Once centered, any two dimensional point $\mathbf{x}_p - \mathbf{w}$ on the circle can be encoded as the (scalar) angle θ_p between the line segment connecting it to the origin and the horizontal axis. Mathematically speaking, we have

$$f_e\left(\mathbf{x}_p\right) = \theta_p = \arctan\left(\frac{x_{p,2} - w_2}{x_{p,1} - w_1}\right) \tag{10.55}$$

To design the decoder, beginning with θ_p, we can reconstruct $\mathbf{x}_p$ as

$$f_d\left(\theta_p\right) = \begin{bmatrix} r_1\cos(\theta_p) + d_1 \\ r_2\sin(\theta_p) + d_2 \end{bmatrix} \tag{10.56}$$

Taken together this encoder/decoder pair defines an appropriate nonlinear Autoencder model of the general form given in Equation (10.51), with a set of parameters $\Theta = \{\mathbf{w}, r_1, r_2, d_1, d_2\}$.

In the middle panel of Figure 10.15 we show the final learned manifold along with the decoded data, found by minimizing the cost function in Equation (10.54) via gradient descent. In the right panel of the Figure we show the manifold recovered as a black circle (with red outline for ease of visualization), and illustrate how points in the space are attracted (or *projected*) to the recovered manifold as a vector field with arrows colored according to their distance to the linear subspace.

10.6 Exercises

Section 10.1 exercises

1. **Modeling a wave**

   Repeat the experiment described in Example 10.1, including making the panels shown in Figure 10.2.

2. **Modeling population growth**

   In Figure 10.17 we show a population growth dataset - which shows the population of Yeast cells growing in a constrained chamber. This is a common shape found with population growth data, where the creature under study starts off with only a few members and is limited in growth by how fast it can reproduce and the resources available in its environment. In the beginning such a population grows exponentially, but the growth halts rapidly when the population reaches the maximum carrying capacity of its environment.

   Propose a *single* nonlinear feature transformation for this dataset and fit a corresponding model to it using the Least Squares cost function and gradient descent. Do not forget to *standard normalize* the input of the data (see Section

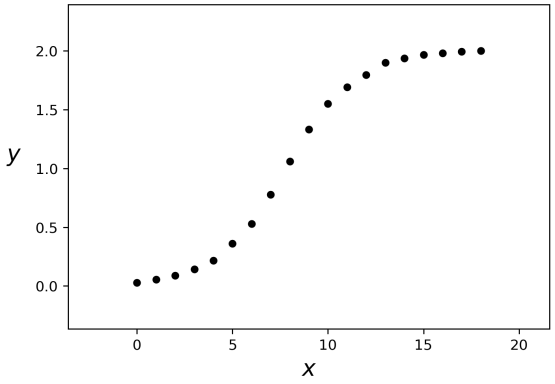

Figure 10.17 Figure associated with Exercise 2. See text for details.

9.3). Plot the data along with the final fit provided your model together in a single panel.

3. **Galileo's experiment**

Repeat the experiment described in Example 10.2, including making the panels shown in Figure 10.5.

4. **Moore's law**

Gordon Moore, co-founder of Intel corporation, predicted in a 1965 paper[5] that the number of transistors on an integrated circuit would double approximately every two years. This conjecture, referred to nowadays as the Moore's law, has proven to be sufficiently accurate over the past 5 decades. Since the processing power of computers is directly related to the number of transistors in their CPUs, Moore's law provides a trend model to predict the computing power of future microprocessors. Figure **??** plots the transistor counts of several microprocessors versus the year they were released, starting from Intel 4004 in 1971 with only 2300 transistors, to Intel's Xeon E7 introduced in 2014 with more than 4.3 billion transistors.

a) Propose an single feature transformation for the Moore's law dataset shown in Figure **??** so that the transformed input/output data is related linearly. *Hint: to produce a linear relationship you will end up having to transform the output, not the input.*

b) Formulate and minimize a Least Squares cost function for appropriate weights, and fit your model to the data in the original data space.

[5] One can find a modern reprinting of this paper in e.g., (**?**).

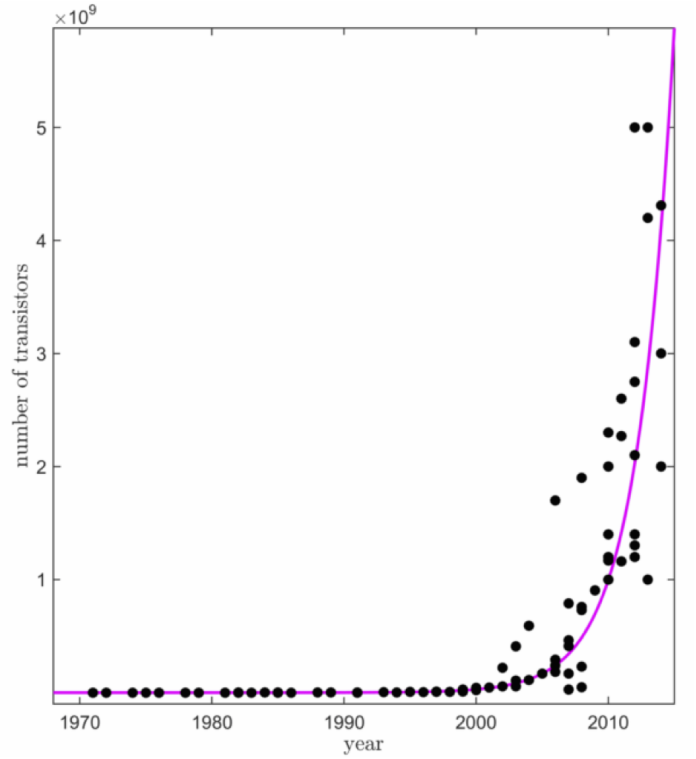

Figure 10.18 As Moore proposed fifty years ago, the number of transistors in microprocessors versus the year they were invented follows an exponential pattern.

5. Ohm's law and linear regression

Ohm's law, proposed by the German physicist Georg Simon Ohm following a series of experiments made by him in the 1820s, connects the magnitude of the current in a galvanic circuit to the sum of all the exciting forces in the circuit, as well as the length of the circuit. Although he did not publish any account of his experimental results, it is easy to verify his law using a simple experimental setup, shown in the left panel of Figure 10.19, that is very similar to what he then utilized (the data in this Figure is taken from (**?**)). The spirit lamp heats up the circuit, generating an electromotive force which creates a current in the coil deflecting the needle of the compass. The tangent of the deflection angle is directly proportional to the magnitude of the current passing through the circuit. The magnitude of this current, denoted by I, varies depending on the length of the wire used to close the circuit (dashed curve). In the right panel of Figure 10.19 we plot the readings of the current I (in terms of the tangent of the deflection angle) when the circuit is closed with a wire of length x (in cm), for 5 different values of x.

a) Suggest a single nonlinear transformation of the original data to fit so that the transformed input/output data is related linearly. *Hint: to produce a*

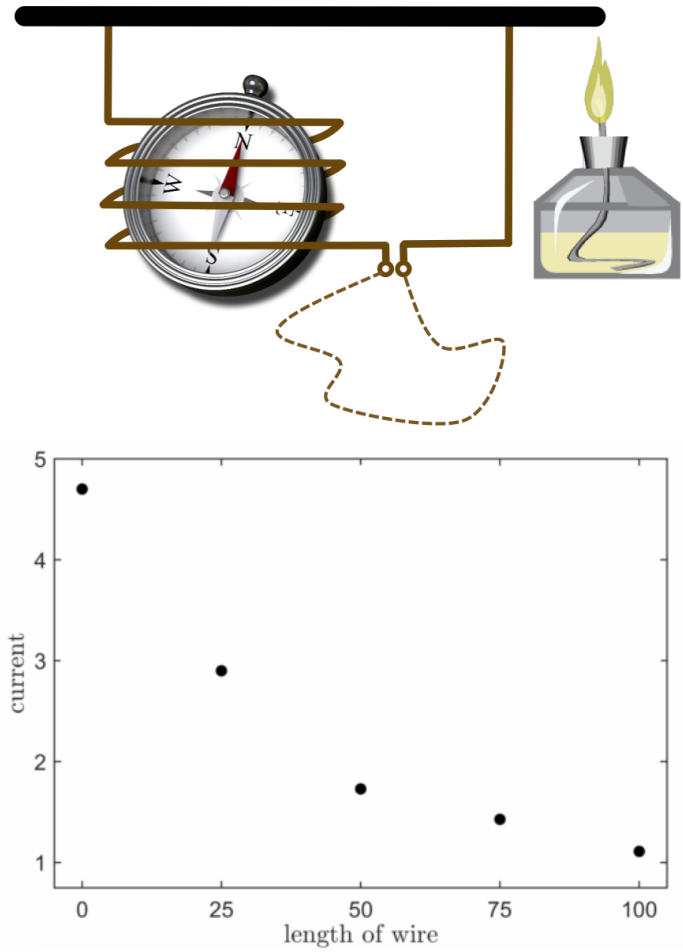

Figure 10.19 (left panel) Experimental setup for verification of Ohm's law. Black and brown wires are made up of constantan and copper, respectively. (right panel) Current measurements for 5 different lengths of closing wire.

linear relationship you will likely end up having to transform the output.

b) Formulate a proper Least Squares cost function using your transformed data and minimize it to recover ideal parameters for your model.

c) Fit your proposed model to the data and display it in the original data space.

6. **Determining the orbit of celestial bodies**
One of the first recorded uses of regression via the Least Squares approach was made by Carl Frederich Gauss, a German mathematician, physicist, and all around polymath, who was interested in calculating the orbit of the asteroid Pallas by leveraging a dataset of recorded observations. Although Gauss solved the problem using ascension and declination data observed from the earth (**?**), here we modify the problem so that the simulated data

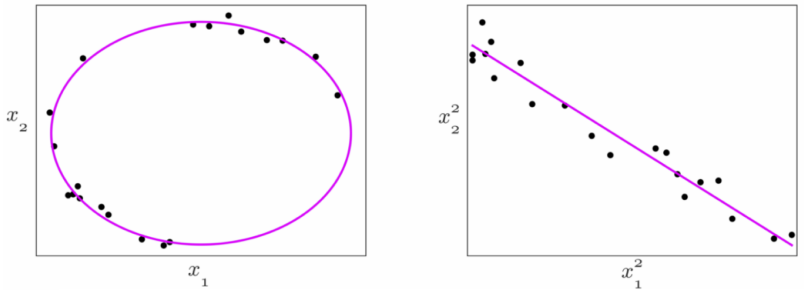

Figure 10.20 Simulated observation data for the location of the asteroid Pallas on its orbital plane. The ellipsoidal curve fit to the data approximates the true orbit of Pallas. (right panel) Fitting an ellipsoid to the data in the original data space is equivalent to fitting a line to the data in a new space where both dimensions are squared.

shown in the left panel of Figure 10.20 simulates Cartesian coordinates of the location of the asteroid on its orbital plane. With this assumption, and according to Kepler's laws of planetary motion, we need to fit an ellipse to a series of observation points in order to recover the true orbit.

In this instance the data comes in the form of $P = 20$ noisy coordinates $\left\{\left(x_{1,p},\, x_{2,p}\right)\right\}_{p=1}^{P}$ taken from an ellipsoid with the standard form of

$$\left(\frac{x_{1,p}}{v_1}\right)^2 + \left(\frac{x_{2,p}}{v_2}\right)^2 \approx 1 \quad \text{for all } p = 1...P, \tag{10.57}$$

where v_1 and v_2 are tunable parameters. By making the substitutions $w_1 = \left(\frac{1}{v_1}\right)^2$ and $w_2 = \left(\frac{1}{v_2}\right)^2$ this can be phrased equivalently as a set of approximate linear equations

$$x_{1,p}^2 w_1 + x_{2,p}^2 w_2 \approx 1 \quad \text{for all } p = 1...P. \tag{10.58}$$

a) Reformulate the equations shown above using vector notation as

$$\mathbf{f}_p^T \mathbf{w} \approx y_p \quad \text{for all } p = 1...P \tag{10.59}$$

by determining the appropriate $\mathbf{f}_p$ and y_p where $\mathbf{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^T$.

b) Formulate and solve the associated Least Squares cost function to recover the proper weights $\mathbf{w}$ and plot the ellipse with the data shown in the left panel of Figure **??** located in the file *asteroid_data.csv*.

Section 10.2 exercises

7. **A wavy multi-output dataset**

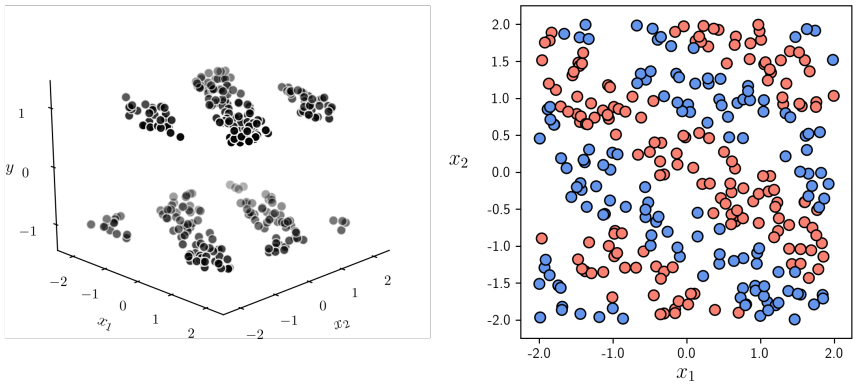

Figure 10.21 Figure associated with Exercise 9. See text for further details.

Repeat the experiment outlined in Example . You need not reproduce the illustrations shown in Figure 10.7, but use a cost function history plot to ensure that you are able to learn an accurate fit to the data.

Section 10.3 exercises

8. **An elliptical boundary**

Repeat the experiment outlined in Example 10.5. You need not reproduce the illustrations shown in Figure **??**, but use a cost function history plot to ensure that you are able to learn an accurate fit to the data.

9. **Nonlinear two-class classification**

Propose a nonlinear model for the dataset shown in Figure 10.21 and perform nonlinear two-class classification. You should be able to achieve perfect classification on this dataset.

Section 10.4 exercises

10. **An elliptical boundaries**

Repeat the experiment outlined in Example 10.6. You need not reproduce the illustrations shown in Figure 10.13, but use a cost function history plot to ensure that you are able to learn an accurate fit to the data.

Section 10.5 exercises

11. **A circular manifold**

Repeat the experiment outlined in Example 10.7. You need not reproduce the illustrations shown in Figure **??**, but use a cost function history plot to ensure that you are able to learn an accurate fit to the data.

12. **Another nonlinear extension of PCA**

As we saw in Section 8.2 in minimizing the PCA Least Squares cost

$$g\left(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\|_2^2. \tag{10.60}$$

over the $N \times K$ spanning set $\mathbf{C}$ and corresponding $K \times 1$ weight vectors $\mathbf{w}_1, ..., \mathbf{w}_P$ determines a proper K dimensional linear subspace for the set of input points $\mathbf{x}_1, .., \mathbf{x}_P$.

As an alternative to extending autoencoder to allow PCA to capture nonlinear subspaces (as described at length in Section 10.5), we can instead extend this PCA Least Squares cost. This is a more restricted version of nonlinear PCA, often termed *Kernel PCA*, that is the basis for similar nonlinear extensions of other unsupervised problems like e.g., K-means as well. In this exercise you will investigate fundamental principles involved in this nonlinear extension.

To do this we begin by choosing a set of B nonlinear features f_1, f_2, ..., f_B *that have no internal parameters* and *transform the input directly*, and denote the transformation of $\mathbf{x}_p$ using this entire set of feature transformations as

$$\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) \\ f_2\left(\mathbf{x}_p\right) \\ \vdots \\ f_B\left(\mathbf{x}_p\right) \end{bmatrix}. \tag{10.61}$$

Then, instead of learning a linear subspace for our input data, we learn one for these transformed input as

$$g\left(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{w}_p - \mathbf{f}_p\right\|_2^2. \tag{10.62}$$

Note here that since each $\mathbf{f}_p$ has size $B \times 1$, our spanning set $\mathbf{C}$ must necessarily consist of equal length vectors and be of size $B \times K$.

a) Assuming that the set of B feature transformations chosen *have no internal parameters*, describe the *classic PCA solution* to the problem of minimizing the cost function above (hint: see Section 8.4). This means that the spanning set $\mathbf{C}$ is assumed to be *orthogonal*.

b) Suppose that we have a dataset of input points distributed roughly on the unit circle in $N = 2$ dimensional space, and that we use the two feature transformations $f_1(\mathbf{x}) = x_1^2$ and $f_2(\mathbf{x}) = x_2^2$. What kind of subspace will we find in both the original and feature transformed space if we use use just the first principle component from the classical PCA solution to represent our data? Draw a picture of what this looks like in both spaces.

c) What can you conclude in general about what is going on simultaneously - in terms of the kind of subspace on which input and its feature transformed version are being projected onto - in both the *original* input space and the *feature transformed* space of this problem in general? hint: it is entirely analogous to what we saw in the cases of nonlinear regression, two-class, and multi-class classification in this Chapter (see e.g., Example 2 in Section 10.1 and 10.3).

13. **Nonlinear extension of K-Means**

The same idea introduced in the previous exercise can also be used to extend K-means clustering (see Section 8.4) to a nonlinear setting as well - and is the basis for so-called *kernel K-means*. This can be done by first noting that both PCA and K-means *have the same Least Squares cost function*. With K-means however the minimization of this cost function is *constrained* so that each weight vector $\mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^K$ is a standard basis vector (see Section 8.7). Here you will explore the same extension for K-means used with PCA in the previous Exercise.

a) Extend the K-means problem precisely as shown with PCA in the previous exercise. Compare this to the same sort of clustering performed on the original input - in words what is being clustered in each instance?

b) Suppose that we have a dataset of $N = 2$ input points distributed roughly in two clusters: one cluster consists of points distributed roughly on the unit circle, and another consists of points distributed roughly on the circle of radius two centered at the origin. Using the two $f_1(\mathbf{x}) = x_1^2$ and $f_2(\mathbf{x}) = x_2^2$ and $K = 2$ what kind of clusters will we find upon proper execution of K-means, both the original and feature transformed space? Draw a picture of what this looks like in both spaces.

c) What can you conclude in general about what is going on simultaneously

- in terms of the kind of clustering of both the input and its feature transformed version - in both the *original* input space and the *feature transformed* space of this problem in general? hint: it is entirely analogous to what we saw in the cases of nonlinear regression, two-class, and multi-class classification in this Chapter (see e.g., Example 2 in Section 10.1 and 10.3).