Principal Component Analysis

In the previous lectures we have gained from the approach where, given the data and the scaffold of the model, we define the cost function and find the model parameters through its optimization. The approach worked for various kinds of generalized linear models and neural networks, that is, for regression and classification. We also benefitted from regularization, with which we could avoid overfitting (L2) or simplify the model by excluding some of its variables (L1), making it more prone to interpretation. One of the questions we tackle in this chapter is whether we can use this approach for cases where we do not have a target variable and where our principal aim is data representation.

In this and the following chapter, we thus venture into a different kind of modelling, referred to as unsupervised, where we are interested in representation of the data in lower dimensions for the purpose of visualization, pattern discovery, and finding clusters. We will deal with dimensionality reduction by data projection, data embedding, and finally, with clustering. In this chapter, we dive into data projection by means of principal component analysis, and present two approaches, a standard analytical approach and a non-standard one based on representation of the cost function through computation and optimization by gradient descent. We start with the latter, a more general approach. This will also be useful in the next chapter where, instead of projections, we deal with embedding, and will also accommodate regularization, which we cannot carry out analytically.

Let us start with some data and motivation.

Dispersion and Variance

Two examples of two-dimensional data are shown. In both cases, we standardized the data. In the first case, the data are dispersed throughout the entire data space, while in the second they exhibit structure, since the attributes are correlated, with covariance equal to -0.9.

For generating two-dimensional normally distributed data with a

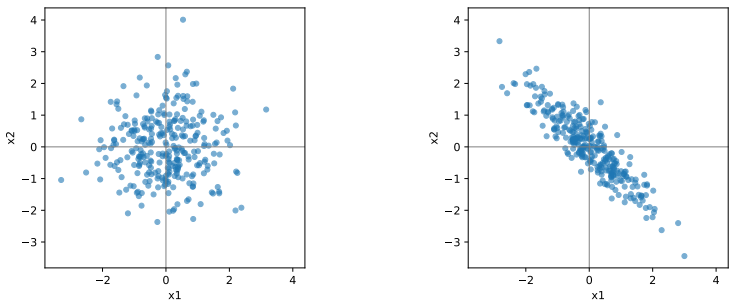

Figure 25: Two examples of standardized data, the first (left) with covariance 0 and the second with covariance -0.9.

given covariance, `numpy` was of help:

```
def generate_2d_normal(n, rho, rnd_seed=42):
    np.random.seed(rnd_seed)
    cov = np.array([[1.0, rho], [rho, 1.0]])
    data = np.random.multivariate_normal(mean=[0, 0], cov=cov, size=n)
    mean = data.mean(axis=0)
    std = data.std(axis=0)
    return (data - mean) / std
```

We would like to determine in which direction the data are stretched, or, equivalently, which is the direction where the data are most dispersed. For our example with covariance -0.9, two such directions are shown below, given by the two direction vectors a and b.

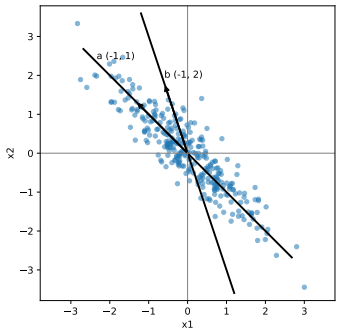

Figure 26: Two candidate directions along which we can observe the dispersion of the data.

Clearly, the data are more dispersed in the direction of vector a. But how do we show this mathematically? In statistics, dispersion is measured by variance. For one-dimensional data $z_1, \ldots, z_n$, the variance is defined as

$$\mathrm{Var}(z) = \frac{1}{n} \sum_{i=1}^{n} (z_i - \bar{z})^2.$$

If we wish to measure dispersion in a specific direction, we project the data onto a unit direction vector u, where $|u| = 1$. We choose vector u as the direction in space (e.g., candidate direction a or b)

that we want to analyze. For direction a, the direction vector will be $u = \frac{a}{|a|}$.

Let the data be written in a matrix $X \in \mathbb{R}^{n \times d}$, where each row represents one instance. Since our data are two-dimensional, matrix X has two columns. The projection of the data onto direction u is then given by

$$X_u = Xu,$$

where $X_u \in \mathbb{R}^n$ represents the (one-dimensional) projection of our data. The direction vector u also determines the weights of the input attributes and therefore their linear combination.

Once again, we are dealing with a linear combination of attributes! Here too we encounter weights again. Does this lead to an interpretation of the results?

For both of our direction vectors, we can now determine the variance obtained when the data are projected onto the directions they define:

```
X = generate_2d_normal(n, rho=rho)
a = np.array([-1.0, 1.0])
b = np.array([-1.0, 3.0])
for v in [a, b]:
    var = projected_variance(X, v)
    print(f"Projected variance along ({v[0]}, {v[1]}): {var:.3f}")
```

The result is as expected: the data projected onto vector a are more dispersed:

```
Projected variance along (-1.0, 1.0): 1.902
Projected variance along (-1.0, 3.0): 1.541
```

Can we algorithmically find the projection direction vector for which the data are most dispersed? We can. The procedure is called principal component analysis, and the reader is probably already familiar with it, but here we would (of course) like to carry it out using gradient descent and automatic differentiation.

Finding the Principal Component

Let u therefore be the direction onto which we project the data. We are looking for u where the data are most dispersed, that is, where the variance of the projection is largest. Let us first write the variance of the projected data:

$$\mathrm{Var}(X_u) = \frac{1}{n} \sum_{i=1}^{n} (x_i^\top u)^2,$$

where we have taken into account that the data are standardized and centered, so the mean of the projection is equal to 0. This can be written in matrix form as

In this notation, we made use of the fact that, if z is a scalar, then $z = z^\top$. How does this help us?

$$\mathrm{Var}(X_u) = \frac{1}{n} u^\top X^\top X u.$$

We define the criterion function

$$J(u) = \frac{1}{n} u^\top X^\top X u,$$

which measures the dispersion of the data in the direction u. We are looking for such a u that maximizes $J(u)$, under the constraint $|u| = 1$. We thus obtain the optimization problem

$$\max_u J(u) \quad \text{subject to} \quad |u| = 1,$$

which we can solve with gradient descent (or ascent), where after each step we will have to normalize the vector that updates the elements of u, to ensure that it remains unit-length.

Let us implement the described functionality in the class `PCA1D`. In doing so, we try to follow the structure of the implementation of linear regression, so that we do not change anything in our implementation of gradient descent or in the function `train`.

It is somewhat inconvenient that we must therefore include `ys` among the parameters of the loss function, but we leave to the reader the changes to the function `train` that would allow calls without `ys` and at the same time allow optimization for linear regression and finding the principal component.

```
class PCA1D:
    def __init__(self, n_inputs, rnd_seed=42):
        rng = random.Random(rnd_seed)
        self.u = [Value(rng.uniform(-1, 1), label=f"u{i}") \
                  for i in range(n_inputs)]
        self.normalize_u()

    def parameters(self):
        return self.u

    def _dot(self, a, b):
        return sum(ai * bi for ai, bi in zip(a, b))

    def normalize_u(self, eps=1e-8):
        nrm = (self._dot(self.u, self.u) + eps) ** 0.5
        self.u = [ui / nrm for ui in self.u]

    def loss(self, xs, ys=None):
        self.normalize_u()
        total = Value(0.0, label="proj_var_sum")
        for x in xs:
            z = self._dot(x, self.u)
            total += z**2
        variance = total / len(xs)
        return -variance

    def batch_loss(self, xs, ys=None, m=20):
        indices = random.sample(range(len(xs)), m)
        batch_xs = [xs[idx] for idx in indices]
```

```
        return self.loss(batch_xs, ys=None)

    def explained_variance(self, xs):
        return -self.loss(xs, ys=None).data

    def __repr__(self):
        self.normalize_u()
        return "PCA1D(u=[" + \
               ", ".join(f"{ui.data:.3f}" for ui in self.u) + "])"
```

In the code, we turned the criterion function into a loss by minimizing $-J(u)$ instead of maximizing $J(u)$. This allows us to use the same learning procedure as before without any changes. Since u must be a unit direction vector, we normalize it in the function `normalize_u` after each update. The method `loss` thus computes the negative variance of the projections of the data onto direction u, and gradient descent should then find the direction in which this variance is largest.

Let us now use our implementation above:

```
X = generate_2d_normal(n, rho=rho)
pca = PCA1D(n_inputs=X.shape[1], rnd_seed=42)
train(pca, X, None, learning_rate=0.05, n_epochs=50, report_every=5, batch_size=None)
pca.normalize_u()

var = -pca.loss(X, ys=None).data
print(f"u: [{', '.join(f'{ui.data:.3f}' for ui in pca.u)}]")
print(f"variance: {var:.3f}")
```

Our example data are evidently quite simple, so the procedure for finding the principal component converges quickly:

```
   5 Loss: -1.769 PCA1D(u=[0.523, -0.853])
  10 Loss: -1.875 PCA1D(u=[0.629, -0.777])
  15 Loss: -1.897 PCA1D(u=[0.674, -0.739])
  20 Loss: -1.901 PCA1D(u=[0.693, -0.721])
  25 Loss: -1.902 PCA1D(u=[0.701, -0.713])
  30 Loss: -1.902 PCA1D(u=[0.704, -0.710])
  35 Loss: -1.902 PCA1D(u=[0.706, -0.708])
  40 Loss: -1.902 PCA1D(u=[0.707, -0.708])
  45 Loss: -1.902 PCA1D(u=[0.707, -0.707])
  50 Loss: -1.902 PCA1D(u=[0.707, -0.707])
u: [0.707, -0.707]
variance: 1.902
```

The direction vector of the principal component coincides with the direction vector a in Figure 26, and the computed variance of the data is also the same.

We call vector u the *principal component*, because it determines

the direction in the data space along which the dispersion of the projections is greatest, that is, the direction that captures the most information from the data in terms of variance. The word "component" here emphasizes that this is a new axis or a new coordinate, constructed as a linear combination of the original attributes, while the word "principal" indicates that among all possible directions this one is the most important. The technique of the same name is built precisely on discovering such principal components, but we will get there shortly.

Use Case

Can such a simple method really be useful? Yes, it can. Who says that we must use it only on synthetic two-dimensional data? Let us illustrate its use on a somewhat more complex example. We collected data on hybrid cars and would like to arrange them somehow along a one-dimensional axis, perhaps for the sake of clarity. A sample of the data is shown in Table 4, while the training set itself included 15 cars.

Car	Type	kWh/ 100 km	HP	0–100 km/h (s)	Trunk (L)
Toyota Yaris Hybrid	0	18	116	9.7	286
Nissan Qashqai e-POWER	0	21	190	7.9	504
Lexus NX 450h+	1	24	309	6.3	520
Renault Clio E-Tech Hybrid	0	18	145	9.9	301
Volkswagen Golf eHybrid	1	21	204	7.4	273

Table 4: Data with a sample of modern hybrid vehicles and their performance and utility characteristics. Type 0 denotes hybrids, while 1 denotes plug-in hybrids.

We read the data, store the names of the cars, standardize the data, and compute the principal component.

```
df = pd.read_excel("cars.xlsx")
labels = df.iloc[:, 0].astype(str).to_list()
feat_names = df.columns[1:].to_list()
X = df.iloc[:, 1:].astype(float).to_numpy()
X = (X - X.mean(axis=0)) / X.std(axis=0)

pca = PCA1D(n_inputs=X.shape[1], rnd_seed=42)
train(pca, X, None, learning_rate=0.05, n_epochs=30, report_every=5, batch_size=None)
pca.normalize_u()
```

Convergence is fast here as well, with gradient descent finding the solution in a few dozen steps. At the end, we also compute the variance of the projections, the attribute weights (loadings), and the projections of individual cars onto the principal component (scores).

```
var = -pca.loss(X, ys=None).data
loadings = sorted(((n, u.data) for n, u in zip(feat_names, pca.u)), key=lambda t: abs(t[1]), reverse=True)
scores = X @ np.array([u.data for u in pca.u])
```

We can now plot the positions of the cars along the linear axis:

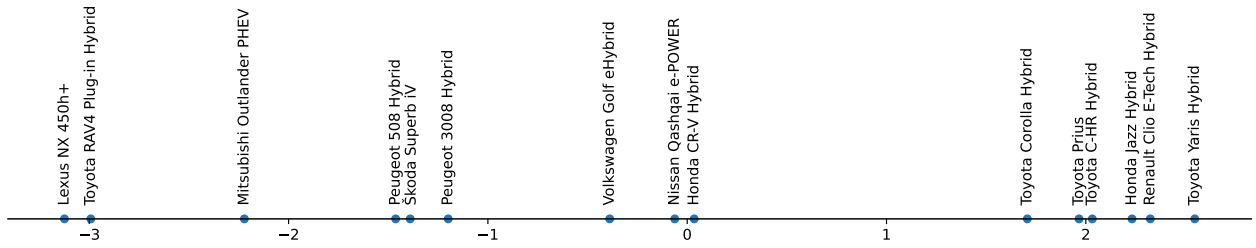

Figure 27: Ranking of cars using the principal component method.

The cars in our small dataset split into several groups. On the right are smaller, more economical cars, while on the left are more advanced, larger, and probably also considerably more expensive ones. Price was not included in the table, and we leave further interpretation of the results to the reader. Still, it is evident that even such a simple technique and the visualization of its results are useful and offer many possibilities for interpretation. Of course, in interpretation we would also be interested in the influence of individual attributes, or, equivalently, in their weights (loadings). Here they are:

```
-0.501: Horsepower
-0.476: kWh/100 km
 0.454: 0-100 km/h (s)
-0.449: Type
-0.339: Trunk Size (L)
```

The largest influence comes from engine power (`horsepower`), closely followed by electricity consumption (`kWh/100 km`), and then by acceleration, drivetrain type, and trunk size. The negative signs for power, consumption, and type indicate that larger, more powerful, and generally plug-in hybrids lie at one end of the axis, while the positive contribution of acceleration time means that slower, less powerful cars lie at the other. The principal component thus largely explains the transition from smaller, less powerful, and more economical vehicles to larger, more powerful, and more energy-demanding ones.

Variance in Multidimensional Spaces

So far, we have measured the dispersion of data in one dimension, where variance is defined as the average of the squared deviations from the center of the data. But how do we measure dispersion when the data are multidimensional? Let the data be $x_1, \ldots, x_n \in \mathbb{R}^d$. Assume that they are centered, that is, $\bar{x} = 0$. A natural generalization of variance in a multidimensional space is

$$\mathrm{Var}(X) = \frac{1}{n} \sum_{i=1}^{n} \|x_i\|^2,$$

where $\|x_i\|$ denotes the Euclidean norm of vector x_i. This definition makes sense only if we assume that the space $\mathbb{R}^d$ is Euclidean, that is, equipped with Euclidean geometry and with the standard inner product

$$x^\top y = \sum_{j=1}^{d} x_j y_j,$$

and the associated norm $\|x\|^2 = x^\top x$. In such a space, the familiar geometric laws hold, among them the Pythagorean theorem.

Proposition. If the data are written in coordinates as $x_i = (x_{i1}, \ldots, x_{id})$, then

$$\mathrm{Var}(X) = \sum_{j=1}^{d} \mathrm{Var}(X^{(j)}),$$

where $X^{(j)}$ denotes the j-th coordinate of the data.

Proof. Let us begin with the definition of multidimensional variance:

$$\mathrm{Var}(X) = \frac{1}{n} \sum_{i=1}^{n} \|x_i\|^2.$$

Since we are using the Euclidean norm, we can expand the squared norm by coordinates:

$$\|x_i\|^2 = \sum_{j=1}^{d} x_{ij}^2.$$

Substituting this gives

$$\mathrm{Var}(X) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d} x_{ij}^2.$$

We interchange the order of summation:

$$\mathrm{Var}(X) = \sum_{j=1}^{d} \left(\frac{1}{n} \sum_{i=1}^{n} x_{ij}^2 \right).$$

The expression in parentheses is precisely the variance of the j-th coordinate:

$$\mathrm{Var}(X^{(j)}) = \frac{1}{n}\sum_{i=1}^{n} x_{ij}^2.$$

We thus obtain

$$\mathrm{Var}(X) = \sum_{j=1}^{d} \mathrm{Var}(X^{(j)}).$$

□

The result tells us that in a Euclidean space the total dispersion of the data decomposes into contributions from the individual coordinates. This is a direct consequence of the Pythagorean theorem: the square of the length of a vector is the sum of the squares of its projections onto orthogonal axes.

Since the variances add up across axes, and since the data in Figure 25 were standardized, we can conclude that the total variance of both of these datasets is equal to 2. The projections onto direction a from Figure 26 therefore explain $1.902/2.0 = 95\%$ of the variance in the data, while those onto direction b explain only $1.541/2.0 = 77\%$. By *explain*, we mean what proportion of the total dispersion of the data we capture if we represent the data only through their projection onto the chosen direction.

The explained variance indicates what proportion of the total variance of the data is captured by the chosen projection. If the total variance of the data is $\mathrm{Var}(X)$ and the variance of the projection onto direction u is $\mathrm{Var}(Xu)$, then the explained variance ratio is equal to $\mathrm{Var}(Xu)/\mathrm{Var}(X)$. This quantity measures how much information (in the sense of dispersion) is preserved after projection.

Principal Component Analysis

The property above, namely that in a Euclidean space we can decompose the total variance into the sum of variances along individual coordinate axes, is also key to understanding principal component analysis. The goal is to find a new coordinate system in which the axes are mutually orthogonal, and where the total variance is again distributed as the sum of variances along these new axes. Principal component analysis is a technique that selects such a coordinate system in which most of the variance is concentrated in as few axes as possible, so that the first axis captures the largest possible share of the variance, the second axis captures the largest possible share of the remaining variance, and so on.

Principal component analysis (PCA) is often used to find a two-dimensional projection, since the data can then be displayed in a scatter plot. We can extend our class `PCA1D` into a class `PCA2D`, where we simultaneously search for two direction vectors u_1 and u_2, which are unit-length, mutually orthogonal, and determine the directions of the greatest joint variance of the projections. The procedure must ensure that u_1 captures the largest variance, while u_2 captures the largest variance among all directions orthogonal to u_1.

Principal component analysis (PCA) was introduced by Karl Pearson in 1901 as a continuation of his work on correlations and covariance structures of data, where he sought ways to describe multidimensional data using a smaller number of variables. Harold Hotelling later systematized the method in the 1930s, introduced its name, and formally connected it with the eigendecomposition of the covariance matrix and the sequential search for orthogonal components.

It should be noted that optimizing the sum of variances along two orthogonal directions determines the two-dimensional subspace with the greatest explained variance. However, the individual components within this subspace are not necessarily uniquely determined, unless we additionally require that the first direction is the one with the greatest variance, while the second is orthogonal to it and, among such directions, the most variable.

To preserve orthogonality between the vectors during optimization, we use Gram–Schmidt orthogonalization. Let us first assume that we have two vectors v_1 and v_2 that we wish to transform into an orthonormal pair u_1, u_2. The procedure proceeds in two steps, namely, by normalizing the first vector:

$$u_1 = \frac{v_1}{\|v_1\|}.$$

and subtracting the projection and normalizing the second vector, where we first remove from v_2 the component in the direction of u_1,

$$\tilde{v}_2 = v_2 - (v_2^\top u_1)\, u_1,$$

$$u_2 = \frac{\tilde{v}_2}{\|\tilde{v}_2\|}.$$

This guarantees that $u_1^\top u_2 = 0$ and $\|u_1\| = \|u_2\| = 1$. Intuitively, we "clean" the second vector of the component in the direction of the first, which is a direct consequence of the Pythagorean theorem: the remainder lies in the orthogonal direction.

Our implementation follows the one we used for the class `PCA1D`:

```
class PCA2D:
    def __init__(self, n_inputs, rnd_seed=42):
        rng = random.Random(rnd_seed)
        self.v1 = [Value(rng.uniform(-1, 1), label=f"v1_{i}") for i in range(n_inputs)]
        self.v2 = [Value(rng.uniform(-1, 1), label=f"v2_{i}") for i in range(n_inputs)]

    def parameters(self):
        return self.v1 + self.v2

    def _dot(self, a, b):
        return sum(ai * bi for ai, bi in zip(a, b))

    def _norm(self, v, eps=1e-8):
        return (self._dot(v, v) + eps) ** 0.5

    def orthonormal_basis(self):
        # Gram-Schmidt orthonormalization
        nrm1 = self._norm(self.v1)
        u1 = [vi / nrm1 for vi in self.v1]
```

```
        proj_v2_on_u1 = self._dot(self.v2, u1)
        v2_orth = [v2i - proj_v2_on_u1 * u1i for v2i, u1i in zip(self.v2, u1)]
        nrm2 = self._norm(v2_orth)
        u2 = [v / nrm2 for v in v2_orth]
        return u1, u2

    def loss(self, xs, ys=None):
        # Maximize projected variance on two orthonormal axes.
        u1, u2 = self.orthonormal_basis()
        total = Value(0.0, label="proj_var_sum_2d")
        for x in xs:
            z1 = self._dot(x, u1)
            z2 = self._dot(x, u2)
            total += z1**2 + z2**2
        explained_var = total / len(xs)
        return -explained_var

    def batch_loss(self, xs, ys=None, m=20):
        indices = random.sample(range(len(xs)), m)
        batch_xs = [xs[idx] for idx in indices]
        return self.loss(batch_xs, ys=None)
```

Compared to the class `PCA1D`, the key difference here is that we optimize two direction vectors simultaneously. Instead of a single vector u, we have two candidates v_1 and v_2, which we transform during each computation into an orthonormal basis (u_1, u_2) using the Gram–Schmidt procedure. The loss function therefore no longer measures the variance of the projection onto a single direction, but rather the sum of the variances of the projections onto both directions. In this way, we capture the total dispersion of the data in a two-dimensional subspace. It is also important that we perform orthogonalization on the fly (within the method `orthonormal_basis`), so there is no need to constrain the parameters explicitly — the conditions of orthogonality and unit length are ensured by construction. The extended method thus effectively searches for the first two principal components simultaneously.

The program converges quickly here as well. For the purposes of analysis, we print the explained variance ratios

```
Total variance: 5.000
Explained variance (PC1): 2.413 (48.3%)
Explained variance (PC2): 2.102 (42.0%)
Total explained variance (PC1+PC2): 4.515 (90.3%)
```

and explain the composition of the individual components,

```
Loadings, PC1:
 -0.630: Type
  0.444: 0-100 km/h (s)
```

```
 -0.392: Horsepower
 -0.370: kWh/100 km
  0.339: Trunk Size (L)

Loadings, PC2:
 -0.881: Trunk Size (L)
 -0.313: Horsepower
 -0.299: kWh/100 km
  0.187: 0-100 km/h (s)
  0.028: Type
```

and use these in the interpretation of the obtained projection.

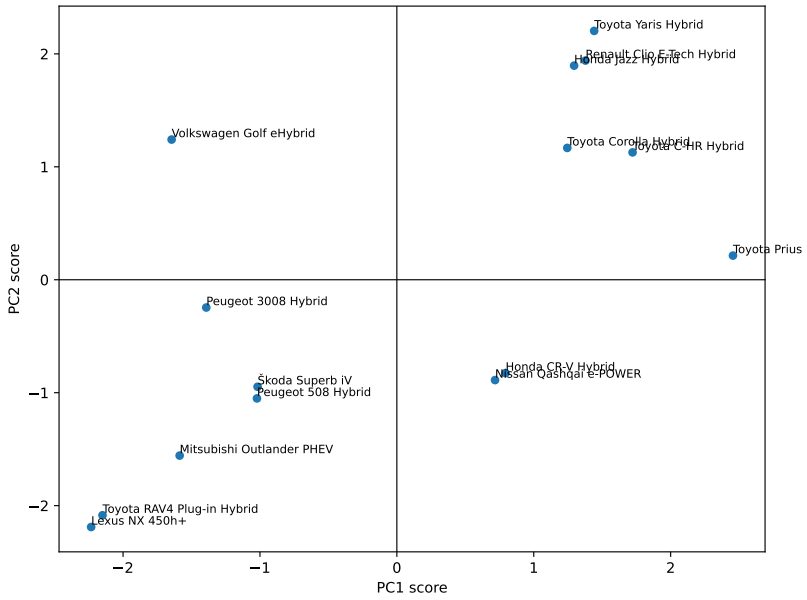

Figure 28: Two-dimensional projection of the car data.

The first two principal components therefore capture more than 90% of the total variance. The two-dimensional projection of the data thus preserves the structure of the data well. The first component (PC1) primarily separates vehicles according to performance and drivetrain type, which is reflected in the larger absolute loadings for the attributes `Type`, `Horsepower`, and `kWh/100 km`. The second component (PC2), on the other hand, is strongly associated with vehicle size (especially `Trunk Size`), and therefore arranges the cars vertically according to spaciousness. In Figure 28, we can thus clearly observe groups of vehicles: smaller, more economical models appear in the upper part of the space, while larger and more powerful vehicles (often plug-in hybrids) appear in the lower part, confirming that PCA successfully reveals the main patterns in the data.

Some Properties Related to Attribute Correlations

Let us briefly recap: in a Euclidean space, variances along orthogonal axes add up. If the attributes are standardized, the variance along each axis is equal to 1, and therefore the total variance of the data is equal to the number of attributes, d. The first principal component will therefore explain more than $1/d$ of the total variance only when some structure exists in the data, that is, when the data are not merely randomly scattered throughout the space.

This is already intuitive in two dimensions. For the data in Figure 25 on the left, where the attributes are uncorrelated and the data are approximately uniformly dispersed in all directions, we can rotate the first component arbitrarily and it will always capture approximately half of the variance. No direction is particularly distinguished. To explain the entire variance, we therefore necessarily need a second, orthogonal component. A similar situation also holds in higher-dimensional spaces: if the data are distributed approximately spherically, no component will be significantly more important than the others.

PCA is therefore most useful when correlations exist among the attributes or, more generally, when the data lie close to some lower-dimensional subspace, for example, some hyperplane. In such a case, the principal components point in the directions of the greatest dispersion of the data, that is, in the directions where the data carry the most information in terms of variance. The proportion of variance explained by an individual component will be greater than $1/d$ to the extent that this structure is more pronounced.

If we wish to explain all of the variance of the original space, we must use all principal components. This, however, is usually not meaningful, since in that case we do not reduce dimensionality at all. The purpose of PCA is not to model absolutely everything, nor all of the noise present in the data. We are often interested in a compromise: to retain only enough components to explain, say, 80% or 90% of the total variance, while neglecting the remainder.

Some caution is nevertheless required here. Directions with small variance often do correspond to noise, but not necessarily always; in certain problems they may also contain important information. PCA is therefore primarily a useful method for data compression, visualization, and structure discovery, rather than necessarily an automatic criterion for distinguishing between signal and noise.

An important property of PCA is also that the principal components are mutually orthogonal, and the projections of the data onto them are uncorrelated. In this way, we obtain a new coordinate system in which the information along individual axes is separated or

non-redundant.

This brings us to the main uses of PCA: (1) visualization of data in one or two dimensions, (2) dimensionality reduction and often also partial noise removal, and (3) transformation into a new, decorrelated attribute space. What a simple linear technique for data analysis, and with such broad applicability! And we are still not at the end. :)

Regularization

Especially for data in high-dimensional spaces, we may be bothered by the fact that the attribute weights for each principal component are nonzero. Ideally, we would like to explain each axis using only a smaller subset of attributes. The solution: L1 regularization! We already know this, and since we are also dealing with a cost function when discovering principal components, we can add regularization here as well. The addition is straightforward: to the loss, which we expressed as the proportion of explained variance (with a negative sign), we add either the sum of squared values (L2) or the sum of absolute values (L1) of the parameters. Here is the addition:

```
params = self.parameters()
if self.reg == "l1":
    loss += self.reg_strength * sum([abs(p) for p in params]) / len(params)
elif self.reg == "l2":
    loss += self.reg_strength * sum([p**2 for p in params]) / len(params)
return loss
```

When setting up the model, we must now add the regularization parameters, for example with

```
pca = PCA2D(n_inputs=X.shape[1], reg="l1", reg_strength=3)
```

while everything else remains the same. Of course, in this way the model, that is, our principal axes and the resulting projection, will differ from the non-regularized one. Specifically, with the settings above, the attribute weights, as shown in the output below,

```
Loadings for PC1:
 -0.541: Horsepower
 -0.533: Type
 -0.532: kWh/100 km
 -0.323: Trunk Size (L)
 -0.188: 0-100 km/h (s)

Loadings for PC2:
  0.865: 0-100 km/h (s)
 -0.502: Trunk Size (L)
```

```
0.000: Type
0.000: kWh/100 km
0.000: Horsepower
```

and the data projection will be somewhat different:

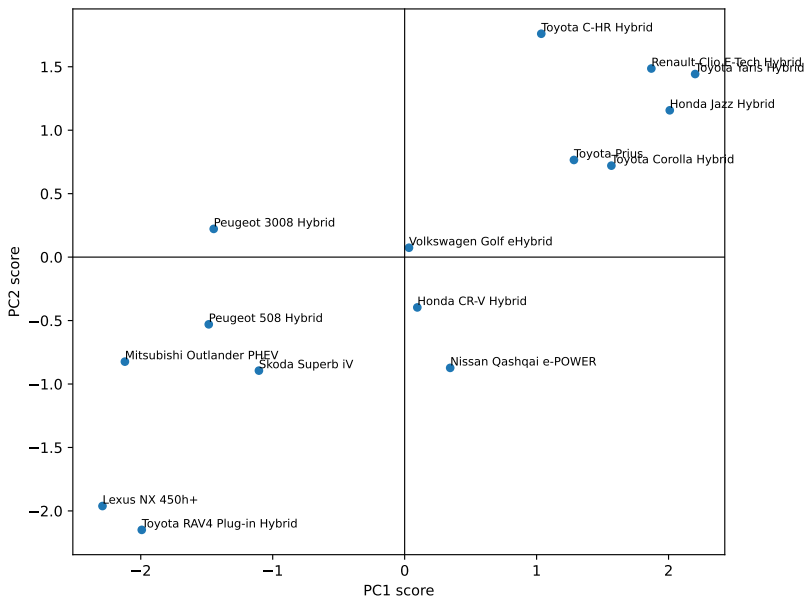

Figure 29: Two-dimensional projection of the car data obtained with L1 regularization ($alpha = 3$).

At this point, one might be surprised by the ease with which we introduced regularization into the principal component analysis framework. Classical PCA does indeed have an elegant analytical solution, but such an approach is also rather rigid, since it is tied to a precisely defined criterion function and orthogonality of the components. Once we wish to include additional requirements in the model, such as L1 or L2 regularization, the analytical solution is generally no longer available, and we must return to numerical optimization.

Classical PCA is based on the eigendecomposition of the covariance matrix $\Sigma = \frac{1}{n} X^\top X$: the principal components are the eigenvectors, while the corresponding eigenvalues give the explained variance along the individual directions.

We can understand principal component analysis not only as a closed mathematical construction, but also as an optimization problem. In doing so, we lose the path to the classical solution, but we also gain considerably more freedom: we can add terms to the criterion function that encourage smoothness, sparsity, or other desired properties of the components. Gradient descent is therefore not merely a didactic approximation of the analytical solution here, but rather a more general framework in which classical PCA appears only as its simplest case.

Above, we used L1 regularization as an example of attribute selection, complexity reduction, and assistance in easier interpretation

of the meaning of the components (loadings). But how can L2 regularization be useful? Let us not forget: the data from which we construct new principal components may be noisy . Even this very simple model of ours can overfit the noise. L2 regularization can mitigate this adaptation and produce a more robust model. Unlike L1 regularization, which encourages sparsity and leads to zero weights, L2 regularization does not eliminate the weights but instead shrinks them uniformly. In this way, it prevents individual attributes from dominating too strongly in determining the directions of the principal components. As a result, we obtain more stable components that are less sensitive to small changes in the data.

An example of such data may come from molecular biology, where modern technology allows us to measure several thousand or tens of thousands of variables simultaneously (e.g., gene expression levels), but because of cost and the limited number of samples (e.g., measured tissues), the amount of data is small, while the possibility of overfitting the model to the data is enormous.

Intuitively, we can say that L2 regularization slightly "smooths" the solution space: instead of the model selecting the direction with the maximum possible variance, which may partly be the result of noise, it rather selects a more balanced direction that better reflects the general structure of the data. In this sense, L2 regularization introduces a compromise between maximum explained variance and model robustness.

Accuracy and Learning Hyperparameters

Above, we also introduced regularization into principal component analysis and thereby introduced the hyperparameter of regularization strength, a parameter whose value we must estimate. But how? PCA does not make predictions, so how can we then evaluate the quality of the model and on this basis choose an appropriate degree of regularization?

We already discussed the evaluation of accuracy and the selection of hyperparameters at the end of the previous chapter. The quality of a model must always be evaluated on a separate test set. In PCA, quality is not tied to prediction, but during the development of PCA we nevertheless defined a criterion function — the explained variance ratio — which can serve us as a measure of how good our model is.

The procedure is therefore very similar, or rather fundamentally the same as in linear regression: we divide the data into a training and a test set. On the training set, we build the model (determine the principal components), and then project the test data onto these components and compute the explained variance on this projection. In this way, we evaluate how well the learned components capture the structure of new data. We must take care that all steps, such as data standardization, are performed on the training set, and that the same transformations are then applied to the test set. Otherwise, we might inadvertently use information from the test data. In doing so, we always compute the explained variance on the test set with respect to

the standardization learned on the training set, since otherwise we would inadvertently include information from the test data in the evaluation.

To estimate an appropriate value of the regularization strength, we can use the same trick as in linear regression: we place the entire method (standardization, selection of the regularization strength on a validation set, and construction of the final model with the thus estimated regularization strength) into a box, and for the purpose of independent evaluation of performance validate this box on the test set. For more robust validation, we can also use cross-validation here. And for the final model, we run the entire box containing our method on all of the data.

Principal Components Are Eigenvectors of the Covariance Matrix

Above, we approached PCA in a non-classical way, differently, through optimization of a criterion function, that is, in an unconventional manner, also because we will tackle many other techniques, where we will be interested in modeling and interpreting data, precisely in this way. PCA, however, also has a beautiful mathematical and statistical derivation and foundation. It proceeds through the following steps.

1. **Centering the Data** Let X be data of dimensions $n \times d$, where n is the number of instances and d is the number of features. We assume that the data are centered, which means that for each feature:
$$\frac{1}{n}\sum_{i=1}^{n} X_{i,j} = 0, \quad \text{for every } j = 1, 2, \ldots, d$$

2. **Finding the First Component $\mathbf{u}_1$.** We seek a vector $\mathbf{u}_1$ (of dimension $d \times 1$) that determines the first principal component — the direction in feature space onto which, if we project the data, the variance of the projections will be maximal. The projection of the data matrix X onto $\mathbf{u}_1$ is:
$$z = X\mathbf{u}_1$$
The variance of the projection z is:
$$\text{Var}(z) = \frac{1}{n}\|X\mathbf{u}_1\|^2 = \frac{1}{n}\mathbf{u}_1^T X^T X \mathbf{u}_1$$
Since we wish to find the direction $\mathbf{u}_1$ with maximal variance, our criterion function is:
$$\max_{\mathbf{u}_1} \mathbf{u}_1^T S \mathbf{u}_1$$
where a somewhat closer look at the structure above reveals that $S = \frac{1}{n}X^T X$ is the covariance matrix of the data. At the same time,

we constrain the length of vector $\mathbf{u}_1$ to 1. This is also the condition that the criterion function must satisfy:

$$\mathbf{u}_1^T \mathbf{u}_1 = 1$$

3. **Optimization.** We find the solution to the problem using Lagrange multipliers. We define the Lagrangian:

$$L(\mathbf{u}_1, \lambda) = \mathbf{u}_1^T S \mathbf{u}_1 - \lambda(\mathbf{u}_1^T \mathbf{u}_1 - 1)$$

and look for stationary points:

$$\frac{\partial L}{\partial \mathbf{u}_1} = 2S\mathbf{u}_1 - 2\lambda \mathbf{u}_1 = 0$$

$$S\mathbf{u}_1 = \lambda \mathbf{u}_1$$

The equation above is the eigenvalue equation of the covariance matrix. Thus, $\mathbf{u}_1$ is an eigenvector of the covariance matrix S, and the corresponding eigenvalue λ is:

$$S\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$$

4. **Solution**. The variance of the projection is therefore given by:

$$\text{Var}(z) = \frac{1}{n} \|X\mathbf{u}_1\|^2 = \mathbf{u}_1^T S \mathbf{u}_1$$

But from the eigenvalue equation,

$$S\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$$

it follows that:

$$\mathbf{u}_1^T S \mathbf{u}_1 = \mathbf{u}_1^T \lambda_1 \mathbf{u}_1 = \lambda_1 (\mathbf{u}_1^T \mathbf{u}_1) = \lambda_1$$

since we assumed that $\mathbf{u}_1^T \mathbf{u}_1 = 1$. Therefore:

$$\text{Var}(z) = \lambda_1$$

which means that the eigenvalue λ_1 is equal to the variance of the data in the direction of the first principal component $\mathbf{u}_1$.

Since we wish to maximize the variance of the projection, among all possible eigenvectors of the covariance matrix we choose for the first component the eigenvector corresponding to the largest eigenvalue λ_1. The variance in the direction $\mathbf{u}_1$ is equal to this eigenvalue:

$$\text{Var}(z) = \lambda_1$$

All remaining components are therefore eigenvectors ordered according to decreasing eigenvalues, which determine the explained variances of the individual components.

The derivation above is elegant and establishes the connection between principal components and the space of eigenvectors of the covariance matrix. It also becomes clear here why principal components are related to correlations among attributes. Yet despite the elegance of the solution, it does not allow extensions of the technique through regularization, where the approach using gradient descent, as developed in this chapter, becomes an appropriate solution.