Exercise 4.1

The Python code for performing the fitting is given below in listing 1.

(a) See Fig. 1 and Fig. 2. The plots suggest that age is positively correlated with heart rate, weight is positively correlated with body surface area, and weight is positively correlated with the heart rate. The plots also show a positive correlation between blood pressure and age, weight, body surface area and heart rate.

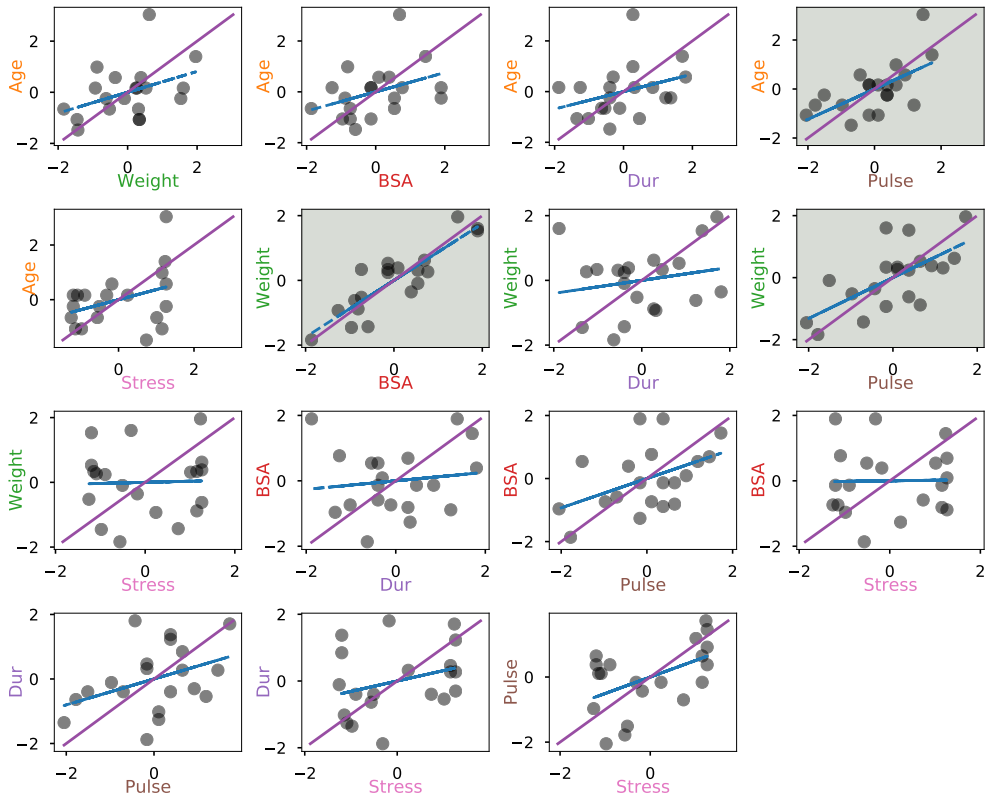

Figure 1: *Pairs of "x-variables". The plots highlighted in gray are pairs of variables with a Pearson correlation coefficient $|\rho| > 0.6$. The blue lines show linear fits to the data. The purple lines show where the data points would be if the x-values were equal to the y-values.*

Plotting all possible combinations of two variables is somewhat tedious when we have many variables. An alternative approach is to calculate the correlation coefficient and highlight it in a "heatmap" plot.[1]. This is shown in Fig. 3 and the conclusions from this figure are the same as we would draw from Fig. 1 and Fig. 2.

(b) See Fig. 1 and Fig. 2. The correlation coefficient is "large" and positive for the pairs

[1] https://matplotlib.org/gallery/images_contours_and_fields/image_annotated_heatmap.html

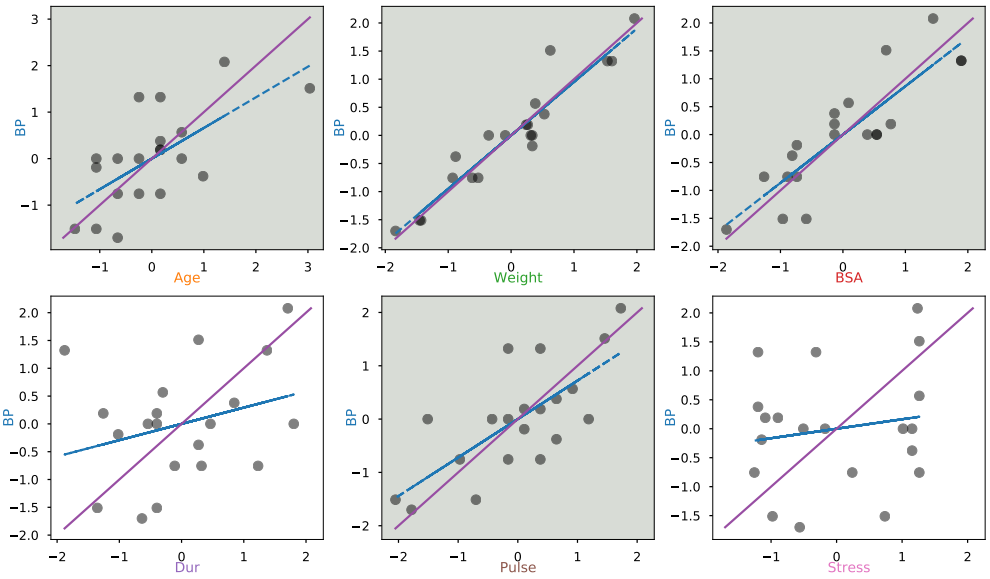

Figure 2: *The "y-variable"(blood pressure) plotted against the different "x-variables". The plots highlighted in gray are pairs of variables with a Pearson correlation coefficient $|\rho| > 0.6$. The blue lines show linear fits to the data. The purple lines show where the data points would be if the x-values were equal to the y-values.*

Figure 3: *The Pearson correlation coefficient for different combination of variables.*

of age and heart rate, weight and body surface area, and weight and heart rate. This is the same as the conclusions found from the visual inspection.

(c) See Fig. 4. In both cases (with or without scaling) the least-squares solution is found successfully. Results for the scaled case are given in table 1, and these results indicate that the age, weight and body surface area are the most important variables. However, since we have seen that the body surface area is correlated with the weight, we expect that this variable can be removed from the fitting (as we will do in the next step).

Variable	**Coefficient**	**P value**, $P > \|t\|$
Age	0.3239	0.000
Weight	0.7673	0.000
BSA	0.0940	0.026
Duration	0.0270	0.165
Pulse	−0.0592	0.111
Stress	0.0381	0.112

Table 1: *Coefficients when fitting with age, weight, body surface area, duration, heart rate and stress. The P value tests the null-hypothesis that the coefficient is zero. If this is less than the confidence level (default:* 0.05*) we expect the coefficient to be different from zero, and that there is a statistically significant relationship between the variable and the response.*

In the case without scaling the following warning is issued by statsmodels:

> The condition number is large, 2.23e+03. This might indicate that there are strong multicollinearity or other numerical problems.

Warnings of this kind typically indicate two things (as the warning says):

i) That some of the "x"-variables are correlated. One should then investigate what variables are correlated and reduce the number of variables so that our set of "x"-variables are uncorrelated.

ii) That there are numerical issues related to finding the least squares solutions. In general, it is hard to identify the root cause of this, but in many cases the situation can be improved by scaling the variables so that their numerical values are of the same order. This is important if our variables have very different units or meanings.

(d) We remove:

i) The body surface area as it is correlated with the weight.

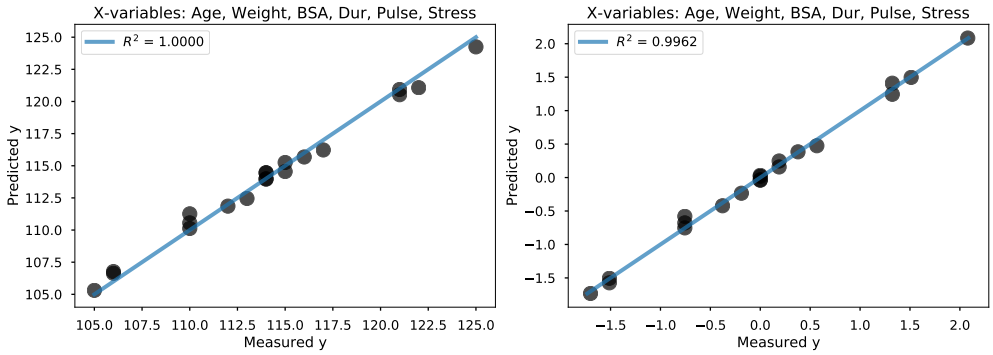

Figure 4: *Results of the ordinary least-squares fitting. The blue line shows where data points would be if measured values were equal to predicted values.*

ii) The heart rate as it is also correlated with the weight.

This leaves four variables: age, weight, duration and stress, and the results of the fitting can be found in Fig. 5. The results from the fitting (the coefficients, see table 2)

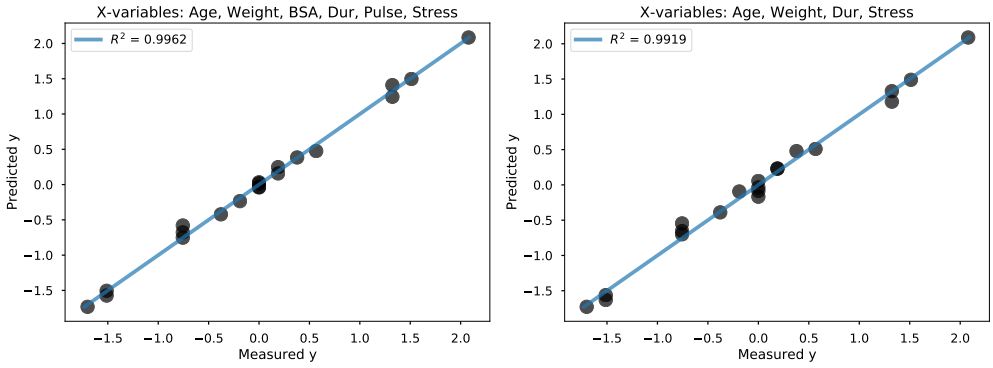

Figure 5: *Results of the ordinary least-squares fitting, comparing the case where we use all variables and the case with just four of the variables. The blue lines show where data points would be if measured values were equal to predicted values.*

indicates that age and weight are more important than the other variables. Thus, we can try to make a new model where we use just two variables: age and weight. The results are shown in Fig. 6. In this figure, two additional models are also shown, using just the age or just the weight.

These results show that we can largely predict the blood pressure using the age and weight, or just the weight alone.

(e) The error from leaving one out is 0.008 and 0.01, respectively, for the two cases. We see that the error is not increased a lot when we reduce the number of variables significantly. Using the alternative formula, the errors are still 0.008 and 0.01.

Variable	**Coefficient**	**P value**, $P > \|t\|$
Age	0.3149	0.000
Weight	0.8181	0.000
Duration	0.0158	0.532
Stress	0.0149	0.560

Table 2: *Coefficients when fitting with age, weight, duration and stress. The P value tests the null-hypothesis that the coefficient is zero. If this is less than the confidence level (default:* 0.05*) we expect the coefficient to be different from zero, and that there is a statistically significant relationship between the variable and the response.*

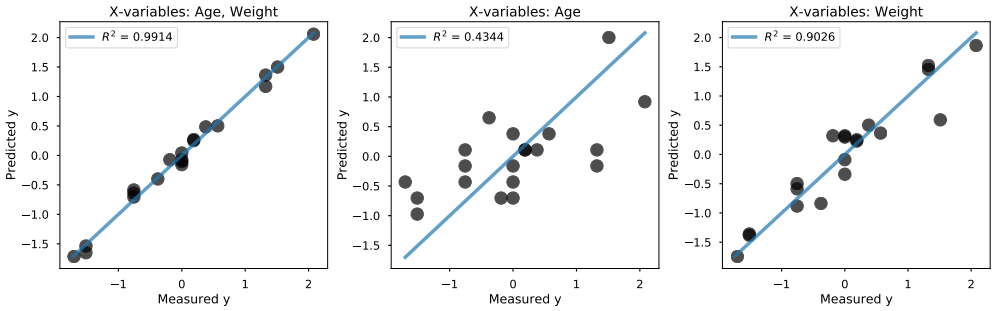

Figure 6: *Results of the ordinary least-squares fitting, comparing cases with fewer variables. The blue lines show where data points would be if measured values were equal to predicted values.*

Exercise 4.2

(a) The labels should **not** be included. If we include them, we are creating principal components that are using the label as a variable. This means that we assume that the label is known for new objects. I.e. there is then no need to predict the class as we already know it!

(b) The raw data is shown in Fig. 7.

(c) In this case, we scale the data. Here, the units for x and y is not given, and to be on the safe side, we scale the data.

(d) After performing the principal component analysis we find:

(i) That we have two principal components.

(ii) That the explained variance is as given in Fig. 8.

(e) We find that:

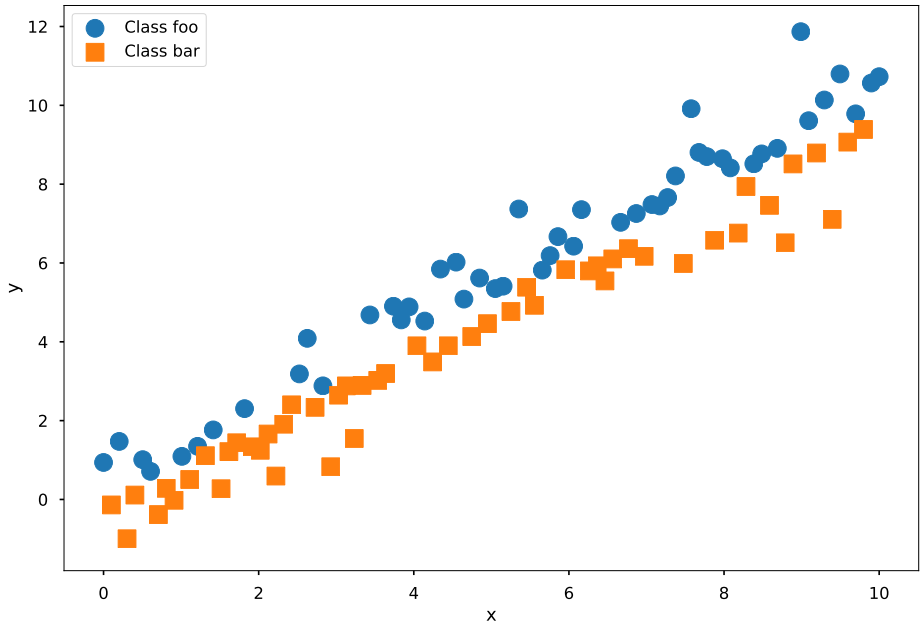

Figure 7: *The raw data.*

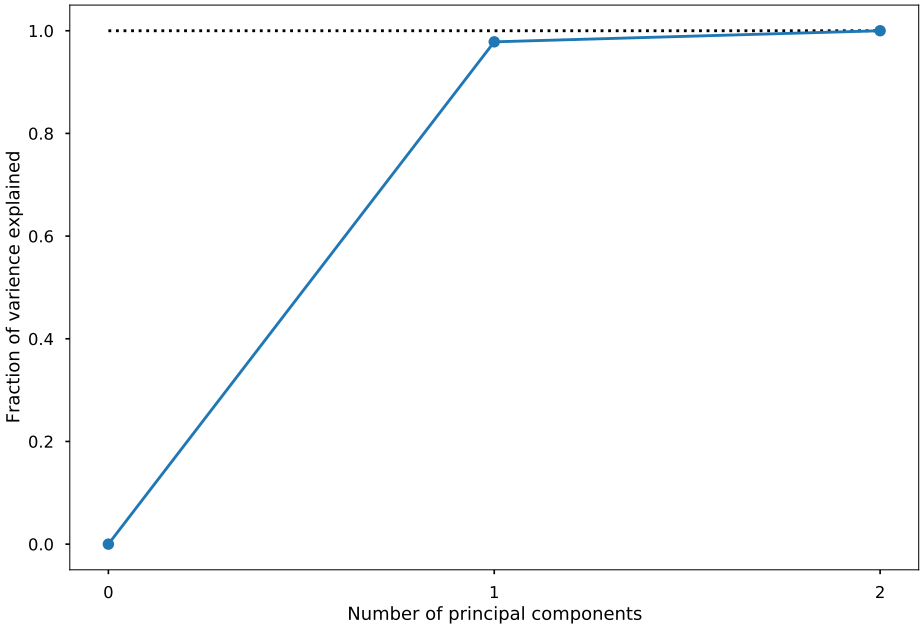

Figure 8: *The explained variance as a function of the number of principal components.*

(i) The two principal components are: $[0.70710678, 0.70710678]$ and $[0.70710678, -0.70710678]$.

(ii) They are normalized.

(iii) The dot product $[0.70710678, 0.70710678] \cdot [0.70710678, -0.70710678] = 0$, so they are orthogonal.

(iv) That they point in the directions we expect (directions of largest variance), see Fig. 9.

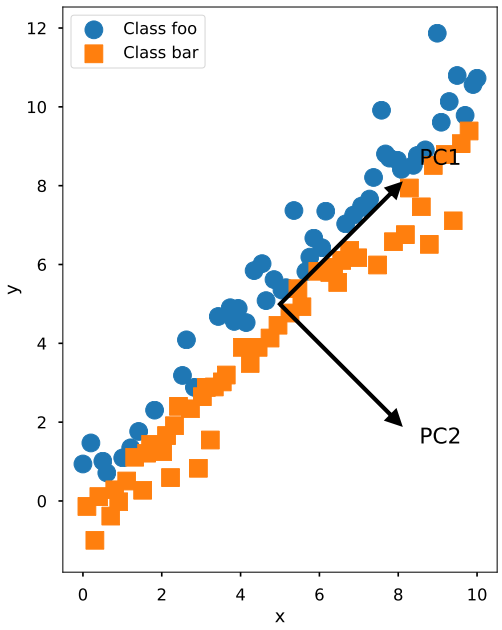

Figure 9: *The raw data and the two principal components (PC1 and PC2).*

(f) The contributions from the original variables to the principal components are shown in Fig. 10.

(g) The scores are shown in Fig. 11

(i) Principal component 2 seems to be more important for separating the two classes.

(ii) The two points are shown in Fig. 11. For point 1 we predict class "foo" and for point 2 we predict class "bar".

(iii) Based on this plot it seems like objects with a score along PC2 > 0 belongs to the class "bar" and objects with a score along PC2 < 0 belongs to the class "foo". A simple rule would then be:

$$\text{class} = \begin{cases} \text{foo} & \text{if score along PC2} < 0 \\ \text{bar} & \text{if score along PC2} > 0 \end{cases}$$

Note that this does not tell us the class if PC2 $= 0$, and that the rule is not perfect (i.e. some points in this plot will be classified incorrectly).

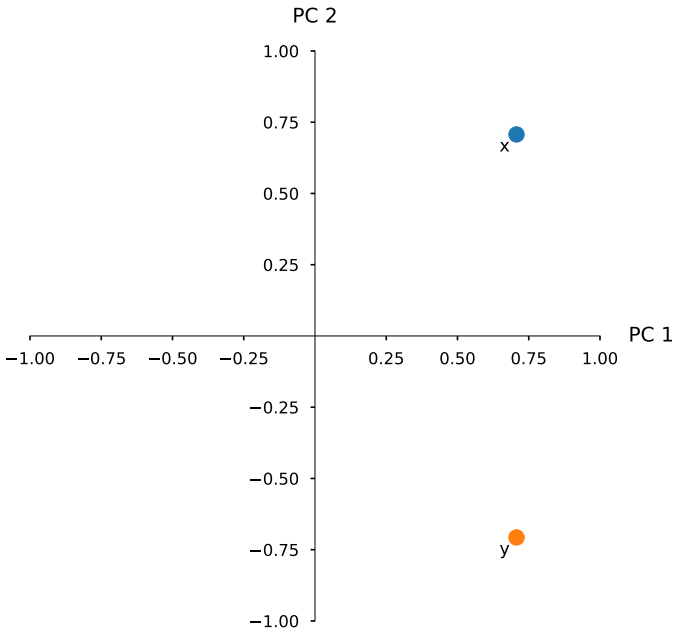

Figure 10: *Contributions of x and y to PC1 and PC2.*

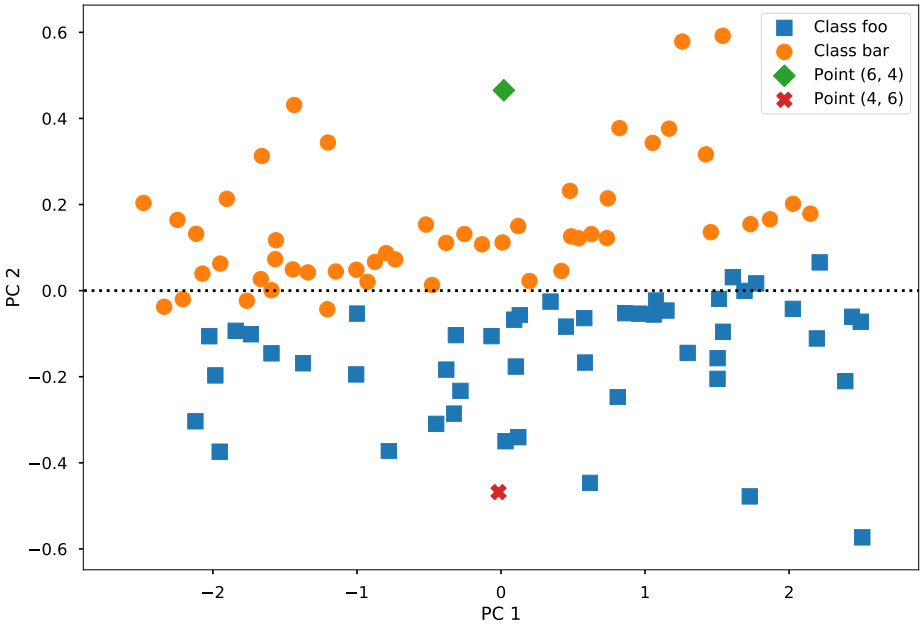

Figure 11: *The scores along PC1 and PC2.*

(iv) If we continue with our simple rule, we have that the score for PC2 should be > 0

for points belonging to the class bar. The score for a given point (x, y) on PC2 is:

$$0.70710678x - 0.70710678y.$$

Thus, the score is > 0 when,

$$0.70710678x - 0.70710678y > 0 \implies x > y.$$

Similarly, we find that points belonging to class foo satisfy $x < y$. Translated, the rule would then be:

$$\text{class} = \begin{cases} \text{foo} & \text{if the point lies above the line } x = y \\ \text{bar} & \text{if the point lies below the line } x = y \end{cases}$$

Note that this does not tell us the class if $x = y$.

(h) PCA does not require us to give it labeled data ("y-values") so PCA is unsupervised. LDA, however, requires labeled data ("y-values") and is thus a supervised method.

Python code

Exercise – Blood pressure

```
from math import ceil
from itertools import combinations
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
from matplotlib.gridspec import GridSpec
import pandas as pd
from scipy.stats import pearsonr
from sklearn.preprocessing import scale
from sklearn.model_selection import LeaveOneOut
import statsmodels.api as sm
plt.style.use('seaborn-talk')


COLORS = {
    'BP': '#1f77b4',
    'Age': '#ff7f0e',
    'Weight': '#2ca02c',
    'BSA': '#d62728',
    'Dur': '#9467bd',
    'Pulse': '#8c564b',
    'Stress': '#e377c2',
}


def heatmap(data, row_labels, col_labels, ax=None,
            cbar_kw={}, cbarlabel="", **kwargs):
    """
    Create a heatmap from a numpy array and two lists of labels.

    Parameters
    ----------
    data
        A 2D numpy array of shape (N, M).
    row_labels
        A list or array of length N with the labels for the rows.
    col_labels
        A list or array of length M with the labels for the columns.
    ax
        A `matplotlib.axes.Axes` instance to which the heatmap is plotted.  If
        not provided, use current axes or create a new one.  Optional.
    cbar_kw
        A dictionary with arguments to `matplotlib.Figure.colorbar`.  Optional
    .
    cbarlabel
        The label for the colorbar.  Optional.
```

```
    **kwargs
        All other arguments are forwarded to `imshow`.
    """

    if not ax:
        ax = plt.gca()

    # Plot the heatmap
    im = ax.imshow(data, **kwargs)

    # Create colorbar
    cbar = ax.figure.colorbar(im, ax=ax, **cbar_kw)
    cbar.ax.set_ylabel(cbarlabel, rotation=-90, va="bottom")

    # We want to show all ticks...
    ax.set_xticks(np.arange(data.shape[1]))
    ax.set_yticks(np.arange(data.shape[0]))
    # ... and label them with the respective list entries.
    ax.set_xticklabels(col_labels)
    ax.set_yticklabels(row_labels)

    # Let the horizontal axes labeling appear on top.
    ax.tick_params(top=True, bottom=False,
                   labeltop=True, labelbottom=False)

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=-30, ha="right",
             rotation_mode="anchor")

    # Turn spines off and create white grid.
    for edge, spine in ax.spines.items():
        spine.set_visible(False)

    ax.set_xticks(np.arange(data.shape[1]+1)-.5, minor=True)
    ax.set_yticks(np.arange(data.shape[0]+1)-.5, minor=True)
    ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
    ax.tick_params(which="minor", bottom=False, left=False)

    return im, cbar


def annotate_heatmap(im, data=None, valfmt="{x:.2f}",
                     textcolors=["black", "white"],
                     threshold=None, **textkw):
    """
    A function to annotate a heatmap.

    Parameters
    ----------
    im
```

```
        The AxesImage to be labeled.
    data
        Data used to annotate.  If None, the image's data is used.  Optional.
    valfmt
        The format of the annotations inside the heatmap.  This should either
        use the string format method, e.g. "$ {x:.2f}", or be a
        `matplotlib.ticker.Formatter`.  Optional.
    textcolors
        A list or array of two color specifications.  The first is used for
        values below a threshold, the second for those above.  Optional.
    threshold
        Value in data units according to which the colors from textcolors are
        applied.  If None (the default) uses the middle of the colormap as
        separation.  Optional.
    **kwargs
        All other arguments are forwarded to each call to `text` used to
    create
        the text labels.
    """

    if not isinstance(data, (list, np.ndarray)):
        data = im.get_array()

    # Normalize the threshold to the images color range.
    if threshold is not None:
        threshold = im.norm(threshold)
    else:
        threshold = im.norm(data.max())/2.

    # Set default alignment to center, but allow it to be
    # overwritten by textkw.
    kw = dict(horizontalalignment="center",
              verticalalignment="center")
    kw.update(textkw)

    # Get the formatter in case a string is supplied
    if isinstance(valfmt, str):
        valfmt = matplotlib.ticker.StrMethodFormatter(valfmt)

    # Loop over the data and create a `Text` for each "pixel".
    # Change the text's color depending on the data.
    texts = []
    # for i in range(data.shape[0]):
    #     for j in range(data.shape[1]):
    #         kw.update(color=textcolors[int(im.norm(data[i, j]) > threshold)
    ])
    #         text = im.axes.text(j, i, valfmt(data[i, j], None), **kw)
    #         texts.append(text)
    # Let the number of given text colors define the new range.
    # Normalize the values onto this range
```

```
    bins = np.linspace(0, 1, len(textcolors) + 1)
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            val = im.norm(data[i, j])
            idx = np.digitize(val, bins, right=True)
            idx = max(idx - 1, 0)
            kw.update(color=textcolors[idx])
            text = im.axes.text(j, i, valfmt(data[i, j], None), **kw)
            texts.append(text)
    return texts


def get_rsquared(yval, yre):
    """Obtain R^2 for fitted data.

    Parameters
    ----------
    yval : numpy.array
        The y-values used in the fitting.
    yre : numpy.array
        The estimated y-values from the fitting.

    Returns
    -------
    rsq : float
        The estimated value of R^2.

    Notes
    -----
    https://en.wikipedia.org/wiki/Coefficient_of_determination

    """
    ss_tot = np.sum((yval - yval.mean())**2)
    ss_res = np.sum((yval - yre)**2)
    rsq = 1.0 - (ss_res / ss_tot)
    return rsq


def fit_linear(xdata, ydata):
    """Fit a linear function.

    Parameters
    ----------
    xdata : numpy.array
        The x-values for the raw data.
    ydata : numpy.array
        The y-values for the raw data.

    Returns
    -------
```

```
    yre : numpy.array
        The y-values estimated by the fitted function.
    pfit : numpy.array
        Estimated coefficients from the fit.
    rsq : float
        The estimated value of R^2 for the fit.

    """
    pfit = np.polyfit(xdata, ydata, 1)
    yre = np.polyval(pfit, xdata)
    rsq = get_rsquared(ydata, yre)
    return yre, pfit, rsq


def plot_xy(xdata, ydata, lines=None, title=None, axi=None):
    """Plot the given data and lines."""
    fig = None
    if axi is None:
        fig = plt.figure()
        axi = fig.add_subplot(111)
    axi.scatter(xdata, ydata, color='black', alpha=0.7, s=200)
    axi.set_xlabel('Measured y')
    axi.set_ylabel('Predicted y')
    if title is not None:
        axi.set_title(title)
    if lines is not None:
        for line in lines:
            lab = line.get('label', None)
            axi.plot(line['x'], line['y'], label=lab, alpha=0.7, lw=4)
            if lab is not None:
                axi.legend()
    if fig is not None:
        fig.tight_layout()
    return fig, axi


def estimate_only_b(xdata, ydata):
    """Least-squares estimate of b when intercept is zero."""
    yprime = ydata - ydata[0]
    return sum(yprime * xdata) / sum(xdata * xdata)


def create_grid(n_plots, ncol):
    """Create a grid for matplotlib given a number of plots and columns."""
    nrow = ceil(n_plots / ncol)
    grid = GridSpec(nrow, ncol)
    return grid


def plot_correlation_heat(data):
```

```
    """Plot a heat map to investigate correlations."""
    corr = data.corr(method='pearson')
    fig1, ax1 = plt.subplots()
    img, cbar = heatmap(
        corr,
        data.columns,
        data.columns,
        ax=ax1,
        cmap='PiYG',
        cbarlabel='Pearson correlation coefficient',
    )
    _ = annotate_heatmap(
        img,
        valfmt='{x:.2f}',
        textcolors=['white', 'black', 'black', 'white'],
        fontsize='large'
    )
    fig1.tight_layout()


def plot_correlation(data):
    """Plot pairs of variables to investigate possible correlations."""
    # Number of X-variables:
    n_var = len(data.columns) - 2
    n_plots = (n_var * (n_var - 1)) / 2

    ncol1 = 4
    grid1 = create_grid(n_plots, ncol1)
    idx1 = -1

    ncol2 = 3
    grid2 = create_grid(n_var, ncol2)
    idx2 = -1

    fig1 = plt.figure()
    fig2 = plt.figure()
    for pair in combinations(data.columns, 2):
        if 'Pt' in pair:
            continue
        if 'BP' in pair:
            idx2 += 1
            row, col = divmod(idx2, ncol2)
            axi = fig2.add_subplot(grid2[row, col])
        else:
            idx1 += 1
            row, col = divmod(idx1, ncol1)
            axi = fig1.add_subplot(grid1[row, col])
        yvar = pair[0]
        xvar = pair[1]
        ydata = data[yvar]
```

```
        xdata = data[xvar]
        pers = pearsonr(xdata, ydata)
        if abs(pers[0]) > 0.6:
            axi.set_facecolor('xkcd:light gray')
        xdata = scale(xdata)
        ydata = scale(ydata)
        axi.scatter(xdata, ydata, color='black', alpha=0.5, s=100)
        yre, _, _ = fit_linear(xdata, ydata)
        axi.plot(xdata, yre, ls='--')
        miny = min(min(xdata), min(yre), min(ydata))
        maxy = max(max(xdata), max(yre), max(ydata))
        axi.plot([miny, maxy], [miny, maxy], color='#984ea3')
        axi.set_xlabel(
            xvar,
            color=COLORS.get(xvar, 'black'),
            labelpad=-2
        )
        axi.set_ylabel(
            yvar,
            color=COLORS.get(yvar, 'black'),
            labelpad=-8
        )
    fig1.tight_layout()
    fig2.tight_layout()


def linear_fit(data, yvar, xvars, scale_data=False, add_constant=False,
               axi=None):
    """Do a linear fitting, based on the OLS example of statsmodels.

    Parameters
    ----------
    data : pandas data frame
        The raw data to use in the blotting.
    yvar : string
        The label used to select the y-variable.
    xvars : list of strings
        The label(s) used to select the x-variables.
    scale_data : boolean, optional
        If True, the data will be scaled to have a mean of 0 and
        a variance of 1.
    add_constant : boolean, optional
        If True, we will add a constant (intercept) to the x-data.
    axi : matplotlib.axis, optional
        If given, we will plot in this axis. Otherwise a new axis
        will be created.
    """
    X = data[xvars]
    y = data[yvar]
    if scale_data:
```

```
        X = scale(X)
        y = scale(y)
    if add_constant:
        X = sm.add_constant(X)
    model = sm.OLS(y, X).fit()
    yre = model.predict(X)
    print(model.summary())
    # Plot predicted y vs the real y to visualize the results:
    miny = min(min(y), min(yre))
    maxy = max(max(y), max(yre))
    lines = [
        {
            'x': [miny, maxy],
            'y': [miny, maxy],
            'label': r'$R^2$ = {:6.4f}'.format(model.rsquared)
        }
    ]
    plot_xy(
        y,
        yre,
        lines=lines,
        title='X-variables: {}'.format(', '.join(xvars)),
        axi=axi,
    )


def linear_fit_loo(data, yvar, xvars, scale_data=False, add_constant=False):
    """Do a linear fitting, based on the OLS example of statsmodels.

    Parameters
    ----------
    data : pandas data frame
        The raw data to use in the blotting.
    yvar : string
        The label used to select the y-variable.
    xvars : list of strings
        The label(s) used to select the x-variables.
    scale_data : boolean
        If True, the data will be scaled to have a mean of 0 and
        a variance of 1.
    add_constant : boolean
        If True, we will add a constant (intercept) to the x-data.
    """
    X = data[xvars]
    y = data[yvar]
    if scale_data:
        X = scale(X)
        y = scale(y)
    if add_constant:
        X = sm.add_constant(X)
```

```
    loo = LeaveOneOut()
    error = []
    for train_index, test_index in loo.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        model = sm.OLS(y_train, X_train).fit()
        yre = model.predict(X_test)
        error.append((yre - y_test)**2)
    avg_error = np.average(error)
    print()
    print('Variables: {}'.format(', '.join(xvars)))
    print('Leave one out error: {:6.4f}'.format(avg_error))
    mse2 = polynomial_mse(X, y)
    print('Leave one out error (alt. calculation): {:6.4f}'.format(mse2))


def polynomial_mse(X, y, y_hat=None):
    """Calculate the MSE using the alternative formulation."""
    if y_hat is None:
        y_hat = sm.OLS(y, X).fit().predict(X)
    H = np.dot(np.dot(X, np.linalg.inv(np.dot(X.T, X))), X.T)
    h = np.diagonal(H)
    mse = np.average(((y - y_hat)/(1.0 - h))**2)
    return mse


def main():
    """Read in the data and run the fitting."""
    data = pd.read_csv('Data/bloodpress.txt', delim_whitespace=True)
    plot_correlation(data)
    plot_correlation_heat(data)
    # Fitting using all variables, scaled and non-scaled:
    fig0 = plt.figure()
    ax01 = fig0.add_subplot(121)
    ax02 = fig0.add_subplot(122)
    linear_fit(
        data,
        'BP',
        ['Age', 'Weight', 'BSA', 'Dur', 'Pulse', 'Stress'],
        scale_data=False,
        add_constant=False,
        axi=ax01
    )
    linear_fit(
        data,
        'BP',
        ['Age', 'Weight', 'BSA', 'Dur', 'Pulse', 'Stress'],
        scale_data=True,
        add_constant=False,
        axi=ax02,
```

```
    )
    fig0.tight_layout()

    fig1 = plt.figure()
    ax11 = fig1.add_subplot(121)
    ax12 = fig1.add_subplot(122)
    linear_fit(
        data,
        'BP',
        ['Age', 'Weight', 'BSA', 'Dur', 'Pulse', 'Stress'],
        scale_data=True,
        add_constant=False,
        axi=ax11
    )
    linear_fit(
        data,
        'BP',
        ['Age', 'Weight', 'Dur', 'Stress'],
        scale_data=True,
        add_constant=False,
        axi=ax12
    )
    fig1.tight_layout()
    fig2 = plt.figure()
    ax21 = fig2.add_subplot(131)
    ax22 = fig2.add_subplot(132)
    ax23 = fig2.add_subplot(133)
    linear_fit(
        data,
        'BP',
        ['Age', 'Weight'],
        scale_data=True,
        add_constant=False,
        axi=ax21
    )
    linear_fit(
        data,
        'BP',
        ['Age'],
        scale_data=True,
        add_constant=False,
        axi=ax22
    )
    linear_fit(
        data,
        'BP',
        ['Weight'],
        scale_data=True,
        add_constant=False,
        axi=ax23
```

```
    )
    fig2.tight_layout()
    # Test leave one out:
    linear_fit_loo(
        data,
        'BP',
        ['Age', 'Weight', 'BSA', 'Dur', 'Pulse', 'Stress'],
        scale_data=True,
        add_constant=False
    )
    linear_fit_loo(
        data,
        'BP',
        ['Age', 'Weight'],
        scale_data=True,
        add_constant=False
    )
    plt.show()


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

Listing 1: *Python code for performing the fitting of the blood pressure data.*

Exercise – PCA

```
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
plt.style.use('seaborn-talk')

# Load data:
data = pd.read_csv('Data/data_exercise4.txt', delim_whitespace=True)
# Extract the two classes for plotting:
class1 = data[data['class'] == 'foo']
class2 = data[data['class'] == 'bar']

# Plotting the raw data:
fig1, ax1 = plt.subplots()
ax1.scatter(class1['x'], class1['y'], label='Class foo', s=200, marker='o')
ax1.scatter(class2['x'], class2['y'], label='Class bar', s=200, marker='s')
ax1.set(xlabel='x', ylabel='y')
ax1.legend()

# Run PCA:
X = data[['x', 'y']]
scaler = StandardScaler()
scaler.fit(X)
```

```
X = scaler.transform(X)
pca = PCA()
scores = pca.fit_transform(X)

# Number of principal components:
print('The number of principal components:', pca.n_components_)
# The explained variance. We have several options for plotting,
# we will here do the cumulative sum:
var =  [0] + list(np.cumsum(pca.explained_variance_ratio_))
fig2, ax2 = plt.subplots()
ax2.plot(var, marker='o', lw=4)
ax2.axhline(y=1, ls=':', color='black')
ax2.set_xlabel('Number of principal components')
ax2.set_ylabel('Fraction of variance explained.')

# There are two principal components, since there are so few,
# we will just grab them:
pc1 = pca.components_[0, :]
pc2 = pca.components_[1, :]
print('PC1', pc1)
print('PC2', pc2)
print('PC1 norm:', np.dot(pc1, pc1))
print('PC2 norm:', np.dot(pc2, pc2))
print('PC1 dot PC2:', np.dot(pc1, pc2))

# Plot PC1 and PC2 in the original plot:
fig3, ax3 = plt.subplots()
ax3.set_aspect('equal')
ax3.scatter(class1['x'], class1['y'], label='Class foo', s=200, marker='o')
ax3.scatter(class2['x'], class2['y'], label='Class bar', s=200, marker='s')
xcenter = scaler.mean_[0]
ycenter = scaler.mean_[1]
for i, pci in enumerate(pca.components_):
    # Make an arrow, pointing in the direction of the vector:
    arrow = ax3.arrow(xcenter, ycenter, pci[0]*4, pci[1]*4, lw=3,
                      color='black', head_width=0.3, head_length=0.3)
    # Add some text close to the end of the arrows:
    ax3.text(xcenter + pci[0]*5, ycenter + pci[1]*5,
             'PC{}'.format(i + 1), fontsize='xx-large')
ax3.set(xlabel='x', ylabel='y')
ax3.legend()


# Plot the contributions from 'x' and 'y' to PC1 and PC2:
fig4, ax4 = plt.subplots()
ax4.set_aspect('equal')
for i, variable in enumerate(('x', 'y')):
    coeff1 = pc1[i]
    coeff2 = pc2[i]
    ax4.scatter(coeff1, coeff2, marker='o', s=200)
```

```
    ax4.text(coeff1 - 0.1, coeff2 - 0.1, variable, fontsize='xx-large')
ax4.set_xlim(-1, 1)
ax4.set_ylim(-1, 1)
ax4.spines['left'].set_position('zero')
ax4.spines['right'].set_visible(False)
ax4.spines['bottom'].set_position('zero')
ax4.spines['top'].set_visible(False)
ax4.text(1.1, 0.0, 'PC1', fontsize='x-large', verticalalignment='center')
ax4.text(0.0, 1.1, 'PC2', fontsize='x-large', horizontalalignment='center')

# Next, we'll do the scores plot:
fig5, ax5 = plt.subplots()
scores1 = scores[data['class'] == 'foo']
scores2 = scores[data['class'] == 'bar']
ax5.scatter(scores1[:, 0], scores1[:, 1], label='Class foo', s=200, marker='o'
    )
ax5.scatter(scores2[:, 0], scores2[:, 1], label='Class bar', s=200, marker='s'
    )
ax5.set(xlabel='PC1', ylabel='PC2')

# Also plot the two points we are asked to predict:
points = np.array([[4., 6.], [6., 4.]])
points = scaler.transform(points)
# Subtract the mean:
scores_p = pca.transform(points)
print(scores_p)
ax5.scatter(scores_p[0][0], scores_p[0][1], s=250, marker='X',
            label='Point (4, 6)')
ax5.scatter(scores_p[1][0], scores_p[1][1], s=250, marker='D',
            label='Point (6, 4)')
ax5.axhline(y=0, ls=':', color='black')
ax5.legend()

for figi in (fig1, fig2, fig3, fig4, fig5):
    figi.tight_layout()
plt.show()
```

Listing 2: *Python code for performing the principal component analysis.*