



CFDS® – Chartered Financial Data Scientist Introduction to Python

Prof. Dr. Natalie Packham

1 December 2022

Table of Contents

- ▼ [5 Statistical Methods in Data Science](#)
 - [5.1 Ridge regression, Lasso and Elastic Net](#)
 - [5.2 Logistic regression](#)
 - [5.3 Principal component analysis](#)

5 Statistical Methods in Data Science

- In this part, we study a number of statistical methods that have become very popular in Data Science applications:
- Statistical Learning methods are often classified into:
 - Regression versus classification
 - Supervised versus unsupervised learning (versus reinforcement learning)
- The methods studied here cover these different aspects:
 - Ridge regression and Lasso (regression, supervised)
 - Logistic regression (classification, supervised)
 - Linear discriminant analysis (classification, supervised)
 - Principal components analysis (regression, unsupervised)

5.1 Ridge regression, Lasso and Elastic Net

- In linear regression, we assume a linear relationship between the *target* Y and the feature vector X :

$$Y = a + b_1 X_1 + b_2 X_2 + \dots + b_m X_m + \epsilon$$

where a, b_1, \dots, b_m are constants and ϵ is the error term.

- The ordinary least squares (OLS) estimates of a, b minimise the errors

$$\sum_{i=1}^n \epsilon^2 = \sum_{i=1}^n (Y_i - a - b_1 X_{i1} - b_2 X_{i2} - \dots - b_m X_{im})^2$$

- In machine learning, especially when the number of features is high and when features are highly correlated, overfitting can occur.
- One way of dealing with this is known as **regularisation**.
- The most popular regularisation methods are:
 - Ridge regression
 - Lasso
 - Elastic net

Ridge regression

- In statistics, **ridge regression** is known as **Tikhonov regularisation** or L_2 **regularisation**.
- Building on OLS, a term is added to the objective function that places a **penalty** on the size of the coefficients b_1, \dots, b_m , by minimising:

$$\sum_{i=1}^n (Y_i - a - b_1 X_{i1} - b_2 X_{i2} - \dots - b_m X_{im})^2 + \lambda \sum_{j=1}^m b_j^2.$$

- The constant λ is called **tuning parameter** or **hyperparameter** and controls the strength of the penalty factor.
- The term $\lambda \sum_{j=1}^m b_j^2$ is called the **shrinkage penalty**, as it will shrink the estimates of b_1, \dots, b_m towards zero.
- Selecting a good value of λ is critical and can be achieved, for example, by **cross-validation**.

Ridge regression

- The OLS estimates do not depend on the magnitude of the independent variables: multiplying X_j by a constant c leads to a scaling of the OLS-coefficient by $1/c$.
- This is different in ridge regression (and Lasso, see below): the estimated coefficients can change substantially when re-scaling independent variables.
- Therefore, it is custom, to *standardise* the features:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n} \sum_{i=1}^n (x_{ij} - \bar{x}_j)^2}},$$

so that all variables are on the same scale, i.e., they all have a standard deviation of one.

Lasso

- **Lasso (Least absolute shrinkage and selection operator)**, also known as L_1 **regularisation** adds a different penalty:

$$\sum_{i=1}^n (Y_i - a - b_1 X_{i1} - b_2 X_{i2} - \dots - b_m X_{im})^2 + \lambda \sum_{j=1}^m |b_j|$$

- This has the interesting effect that the less relevant features are completely eliminated.
- For this reason, Lasso is also often used as a feature selection or variable selection method.

Elastic net regression

- **Elastic net regression** is a mixture of ridge regression and Lasso:

$$\sum_{i=1}^n (Y_i - a - b_1 X_{i1} - b_2 X_{i2} - \dots - b_m X_{im})^2 + \lambda_1 \sum_{j=1}^m b_j^2 + \lambda_2 \sum_{j=1}^m |b_j|$$

- Combining the effects of ridge regression and Lasso means that simultaneously
 - some coefficients are reduced to zero (Lasso),
 - some coefficients are reduced in size (ridge regression).

Example

- The following application predicts house prices based on different features of the property.
- The data set is from

Hull: Machine Learning in Business. 3rd edition, independently published, 2021.

In [556]:

```
import pandas as pd # python's data handling package
import numpy as np # python's scientific computing package
import matplotlib.pyplot as plt # python's plotting package

from sklearn.metrics import mean_squared_error as mse
from sklearn.model_selection import train_test_split
# The sklearn library has cross-validation built in!
# https://scikit-learn.org/stable/modules/cross_validation.html
from sklearn.model_selection import cross_val_score
```

In [557]:

```
# Both features and target have already been scaled: mean = 0; SD = 1
data = pd.read_csv('data/Houseprice_data_scaled.csv')
```

In [558]:

```
X = data.drop('Sale Price', axis=1)
y = data['Sale Price']
```

- sklearn can split training and testing data randomly.

In [559]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

Linear Regression

In [560]:

```
from sklearn.linear_model import LinearRegression
```

In [561]:

```
lr=LinearRegression()  
lr.fit(X_train,y_train)
```

Out[561]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normali  
ze=False)
```

In [562]:

```
pred = lr.predict(X_test)  
mse(y_test, pred)
```

Out[562]:

```
0.12897706428848416
```

- Observe how the OLS coefficients are all non-zero.
- However, some coefficients are negative where a positive coefficient would be expected (e.g. FullBath).
- This is an indication that the model is struggling to fit the large number of features.

In [563]:

```
# Create dataframe with corresponding feature and its respective coefficients
coeffs = pd.DataFrame(['intercept'] + list(X_train.columns), [lr.intercept_] + lr.coef_)
coeffs
```

Out[563]:

	0	1
intercept	0.0155584	
LotArea	0.114813	
OverallQual	0.214531	
OverallCond	0.0887846	
YearBuilt	0.150433	
YearRemodAdd	0.0472879	
BsmtFinSF1	0.117533	
BsmtUnfSF	-0.00289867	
TotalBsmtSF	0.079751	
1stFlrSF	0.119535	
2ndFlrSF	0.0906242	
GrLivArea	0.217879	
FullBath	-0.00945385	
HalfBath	0.014232	
BedroomAbvGr	-0.0676764	
TotRmsAbvGrd	0.0508581	
Fireplaces	0.0293636	
GarageCars	0.0115954	
GarageArea	0.0802374	
WoodDeckSF	0.0333288	
OpenPorchSF	0.0199222	
EnclosedPorch	0.00270256	
Blmngtn	-0.0163658	
Blueste	-0.0116173	
BrDale	-0.022193	
BrkSide	0.0159665	
ClearCr	-0.0081671	
CollgCr	-0.0139869	
Crawfor	0.035561	
Edwards	-0.00185779	
Gilbert	-0.0213031	

	1
0	
IDOTRR	-0.0024635
MeadowV	-0.0154194
Mitchel	-0.034712
Names	-0.0300631
NoRidge	0.0510971
NPkVill	-0.019456
NriddgHt	0.116042
NWAmes	-0.0530869
OLDTown	-0.0142784
SWISU	-0.00529689
Sawyer	-0.0174801
SawyerW	-0.0313221
Somerst	0.027991
StoneBr	0.0861389
Timber	0.0107246
Veenker	-0.0149855
Bsmt Qual	0.0263725

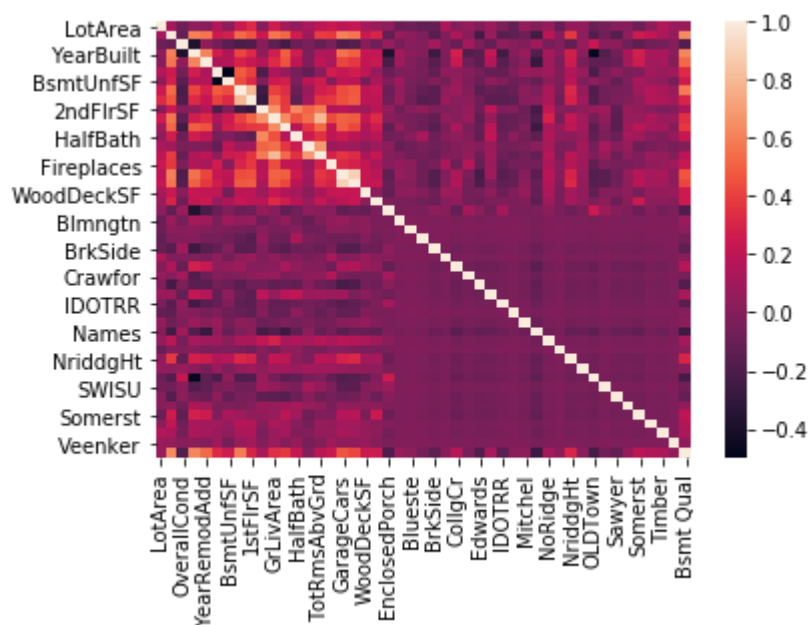
- Indeed, some correlations are high, as the heatmap indicates, which may cause multicollinearity and ill-fitting.

In [564]:

```
sns.heatmap(X_train.corr())
```

Out[564]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7feb887df050>
```



Ridge regression

In [565]:

```
# Importing Ridge
from sklearn.linear_model import Ridge
```

- Train on the training data set and use test data set to determine test MSE.

In [566]:

```

n=np.int(len(X)*.75) # choose 75% of data for training
# The alpha used by Python's ridge should be the lambda above times the number of obs
alphas=[0.01*n, 0.02*n, 0.03*n, 0.04*n, 0.05*n, 0.075*n,0.1*n,0.125*n, 0.15*n,0.2*n,
mSES=[]
for alpha in alphas:
    ridge=Ridge(alpha=alpha)
    ridge.fit(X_train,y_train)
    pred=ridge.predict(X_test)
    mSES.append(mse(y_test,pred))
mSES

```

Out[566]:

```

[0.1290016969234629,
0.12917144141741316,
0.1294122653209535,
0.12970105739268908,
0.13002471627663423,
0.13093789509920933,
0.13194883759388032,
0.13302319862969705,
0.13414251976000655,
0.13647537680000923,
0.14645711609868664]

```

- This is how to use cross-validation; just specify the number of folds (cv) and MSE as the scoring function:

In [567]:

```

# The alpha used by Python's ridge should be the lambda above times the number of obs
alphas=[0.01*n, 0.02*n, 0.03*n, 0.04*n, 0.05*n, 0.075*n,0.1*n,0.125*n, 0.15*n,0.2*n,
mSES=[]
for alpha in alphas:
    scores = cross_val_score(Ridge(alpha=alpha), X, y, cv=4, scoring="neg_root_mean_sqr")
    mSES.append(np.mean(scores))
#np.transpose([alphas, mSES])
mSES

```

Out[567]:

```

[0.12607295502403676,
0.12509029977210734,
0.12435770257439963,
0.12381364622001015,
0.12341528032782012,
0.12287421953596336,
0.1227683115115362,
0.1229385883925512,
0.1232942045844323,
0.12435934533876436,
0.13059077561292226]

```

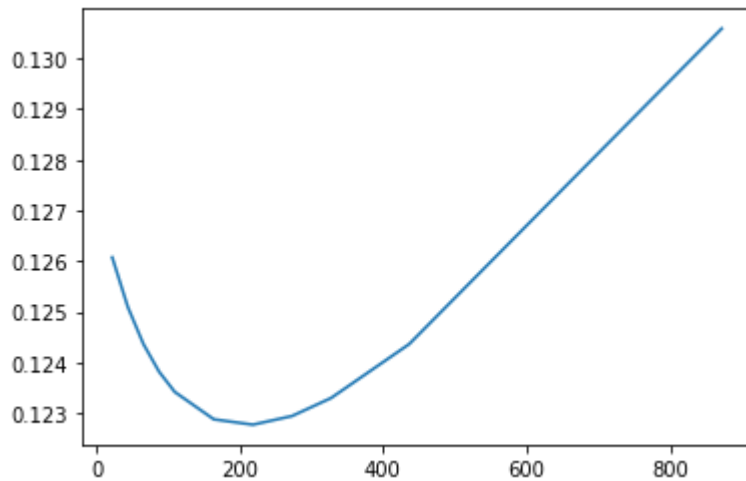
- Average test MSE varies with the hyperparameter α .
- The best model choice is at approximately $0.1 \cdot n = 218$.

In [568]:

```
plt.plot(alphas, mses)
```

Out[568]:

```
[<matplotlib.lines.Line2D at 0x7feb887f2150>]
```



Lasso

In [569]:

```
# Import Lasso
from sklearn.linear_model import Lasso
```

In [570]:

```
# Here we produce results for alpha=0.05 which corresponds to lambda=0.1 in Hull's k
lasso = Lasso(alpha=0.05)
lasso.fit(X_train, y_train)
```

Out[570]:

```
Lasso(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=
None,
      selection='cyclic', tol=0.0001, warm_start=False)
```

- As Lasso acts as a variable selection method we would expect some coefficients to be set to zero:

In [571]:

```
# DataFrame with corresponding feature and its respective coefficients
coeffs = pd.DataFrame(
    [
        ['intercept'] + list(X_train.columns),
        [lasso.intercept_] + lasso.coef_.tolist()
    ]
).transpose().set_index(0)
coeffs
```

Out[571]:

	1
0	
intercept	0.0126168
LotArea	0.0334345
OverallQual	0.297123
OverallCond	0
YearBuilt	0.0367456
YearRemodAdd	0.0734934
BsmtFinSF1	0.105878
BsmtUnfSF	-0
TotalBsmtSF	0.0585557
1stFlrSF	0.0592068
2ndFlrSF	0
GrLivArea	0.290129
FullBath	0
HalfBath	0
BedroomAbvGr	-0
TotRmsAbvGrd	0
Fireplaces	0.0230731
GarageCars	0.0030543
GarageArea	0.0985098
WoodDeckSF	0.00378593
OpenPorchSF	0
EnclosedPorch	-0
Blmngtn	-0
Blueste	-0
BrDale	-0
BrkSide	0
ClearCr	0
CollgCr	-0

	1
0	
Crawfor	0
Edwards	-0
Gilbert	-0
IDOTRR	-0
MeadowV	-0
Mitchel	-0
Names	-0
NoRidge	0.00812487
NPkVill	-0
NriddgHt	0.0746783
NWAmes	-0
OLDTown	-0
SWISU	-0
Sawyer	-0
SawyerW	-0
Somerst	0
StoneBr	0.0442504
Timber	0
Veenker	-0
Bsmt Qual	0.0421831

- Now, let's find again the parameter with minimal average test MSE:

In [572]:

```
# We now consider different lambda values. The alphas are half the lambdas
alphas=[0.0025/2, 0.005/2, 0.01/2, 0.015/2, 0.02/2, 0.025/2, 0.03/2, 0.04/2, 0.05/2]
mses=[]
for alpha in alphas:
    scores = cross_val_score(Lasso(alpha=alpha), X, y, cv=4, scoring="neg_root_mean_
    mses.append(np.mean(scores))
#np.transpose([alphas,mses])
mses
```

Out[572]:

```
[0.1265681481618964,
0.1260315944388273,
0.1254610514096055,
0.12537868421196222,
0.12550453489552857,
0.12585810184192925,
0.1264240784171701,
0.12812145502223948,
0.13074985189167782]
```

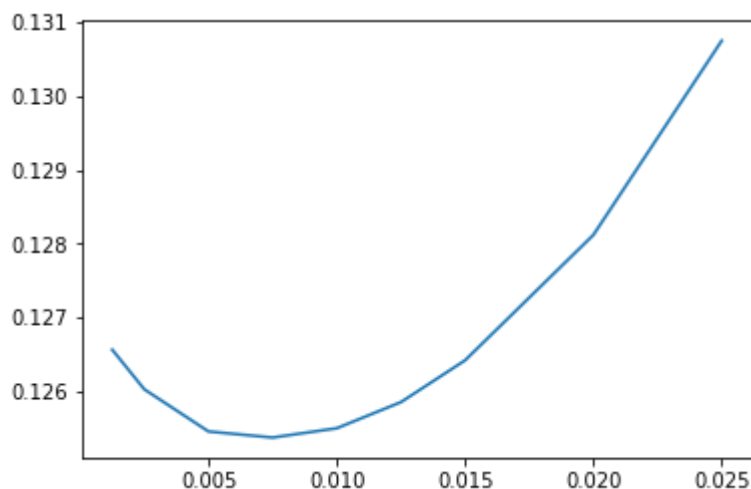
- The optimal parameter is at $\alpha = 0.0075$:

In [573]:

```
plt.plot(alphas, mses)
```

Out[573]:

```
[<matplotlib.lines.Line2D at 0x7feb86847910>]
```



5.2 Logistic regression

- In a regression setting, numerical variables are predicted.
- Another application is classification, which is about predicting the category a new observation belongs to.
- In supervised learning, and with two categories, a variation of regression, called **logistic regression** can be used.
- Given features X_1, \dots, X_m , suppose there are two classes to which observations can belong.

- An example is the prediction of a loan's default risk, given characteristics of the creditor such as age, education, marital status, etc.
- Another example is the classification of e-mails into junk or non-junk e-mails.

Logistic regression

- Logistic regression can be used to calculate the probability of a positive outcome via the **sigmoid function**

$$P(Y = 1|X) = \frac{1}{1 + e^{-X}} = \frac{e^X}{1 + e^X},$$

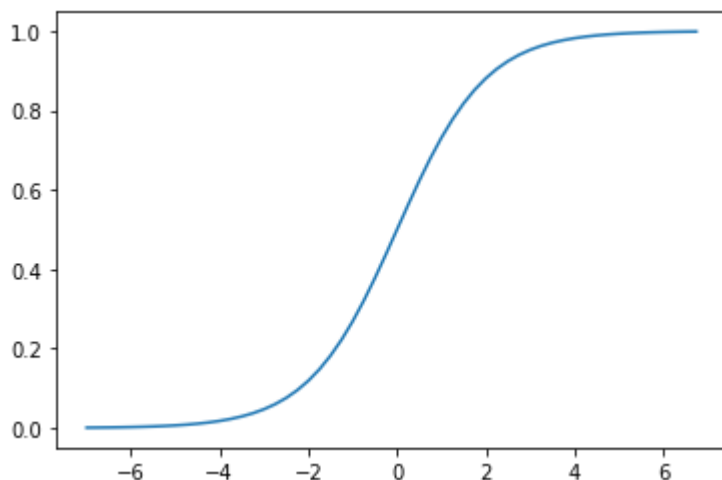
where e is the Euler constant.

In [574]:

```
x=np.arange(-7,7,0.25)
plt.plot(x, 1/(1+np.exp(-x)))
```

Out[574]:

[<matplotlib.lines.Line2D at 0x7feb8685f850>]



Logistic regression

- Setting $Y = a + b_1 X_1 + b_2 X_2 + \dots + b_m X_m$, the probability of a positive outcome is

$$P(Y = 1|X_1, \dots, X_m) = \frac{1}{1 + \exp(-a - \sum_{j=1}^m b_j X_j)}.$$

- The objective is to find the coefficients a, b_1, \dots, b_m that best classify the given data.
- **Maximum likelihood** is a versatile method for this type of problem, when OLS does not apply.
- Without going into detail, the **log likelihood function** is given as

$$\ell(a, b_1, \dots, b_m | x_1, \dots, x_n) = \sum_{k: y_k=1} \ln(p(x_k)) + \sum_{k: y_k=0} \ln(1 - p(x_k))$$

and the parameters are chosen that maximise this function.

- (Note: The likelihood function is derived by considering the observations to be independent outcomes of a Bernoulli random variable.)

Example: Credit risk

- The dataset in this example is taken from James et al.: An Introduction to Statistical Learning. Springer, 2013.
- It contains simulated data of defaults on credit card payments, on the basis of credit card balance (amongst other things).
- An excellent tutorial and examples on logistic regression in Python is available here: <https://realpython.com/logistic-regression-python/> (<https://realpython.com/logistic-regression-python/>).
- We will use the `sklearn` package below. Logistic regression can also be performed with the `statsmodels.api`, in which case p -values and other statistics are calculated.

In [575]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
```

In [576]:

```
data = pd.read_csv("../data/Default_JamesEtAl.csv")
```

In [577]:

```
x=np.array(data["balance"]).reshape(-1,1) # array must be two-dimensional
y=np.array([True if x=="Yes" else False for x in data["default"]]) # list comprehension
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)
```

In [578]:

```
model = LogisticRegression(solver='liblinear', random_state=0)
model.fit(x_train,y_train)
```

Out[578]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, l1_ratio=None, max_iter=100,
                    multi_class='auto', n_jobs=None, penalty='l2',
                    random_state=0, solver='liblinear', tol=0.0001, verbose=0,
                    warm_start=False)
```

In [579]:

```
# fitted parameters
a=model.intercept_[0]
b=model.coef_[0,0]
[a,b]
```

Out[579]:

```
[-8.537515117344592, 0.0041960838665424756]
```

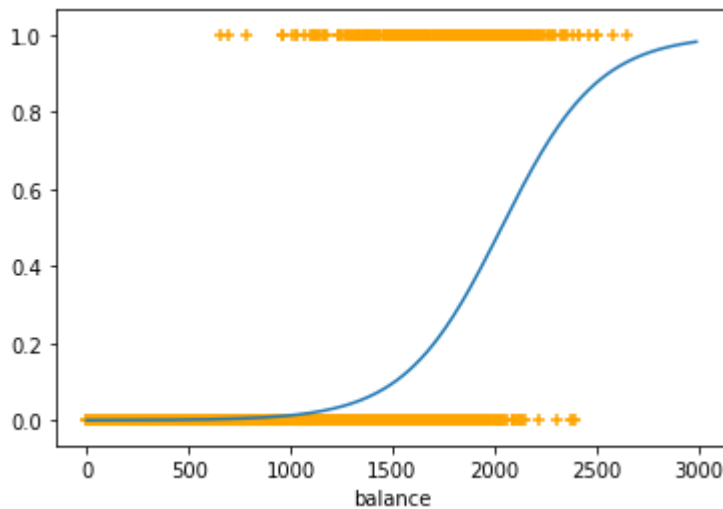
- Scatter plot of data and fitted logistic function:

In [580]:

```
plt.scatter(x,y,c='orange', marker="+")
plt.xlabel('balance')
xrange=range(0,3000,10)
plt.plot(xrange,1/(1+np.exp(-a-b *xrange)))
```

Out[580]:

[<matplotlib.lines.Line2D at 0x7feb88fd7d10>]



- Predictions:

In [581]:

```
model.predict_proba(x_train)[:5]
```

Out[581]:

```
array([[0.98633215, 0.01366785],
       [0.98600606, 0.01399394],
       [0.98133591, 0.01866409],
       [0.998236  , 0.001764  ],
       [0.99643573, 0.00356427]])
```

In [582]:

```
model.predict(x_train)[:10]
```

Out[582]:

```
array([False, False, False, False, False, False, False, False, False,
       False])
```

- Mean accuracy of the model:

In [583]:

```
[model.score(x_train,y_train), model.score(x_test,y_test)]
```

Out[583]:

```
[0.972875, 0.968]
```

- Confusion matrix:

		Actual (True) Values	
		Positive	Negative
Predicted Values	Positive	TP	FP
	Negative	FN	TN

<https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec>

In [584]:

```
confusion_matrix(y_train,model.predict(x_train))
```

Out[584]:

```
array([[7726, 15],
       [ 202, 57]])
```

In [585]:

```
confusion_matrix(y_test,model.predict(x_test))
```

Out[585]:

```
array([[1923, 3],
       [ 61, 13]])
```

- See link below for an explanation of the metrics:

<https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec>
<https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec>

In [586]:

```
print(classification_report(y_train, model.predict(x_train)))
```

	precision	recall	f1-score	support
False	0.97	1.00	0.99	7741
True	0.79	0.22	0.34	259
accuracy			0.97	8000
macro avg	0.88	0.61	0.67	8000
weighted avg	0.97	0.97	0.97	8000

In [587]:

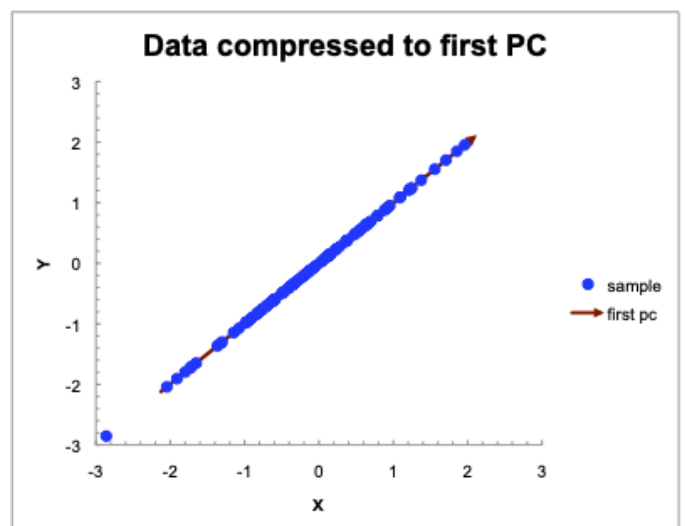
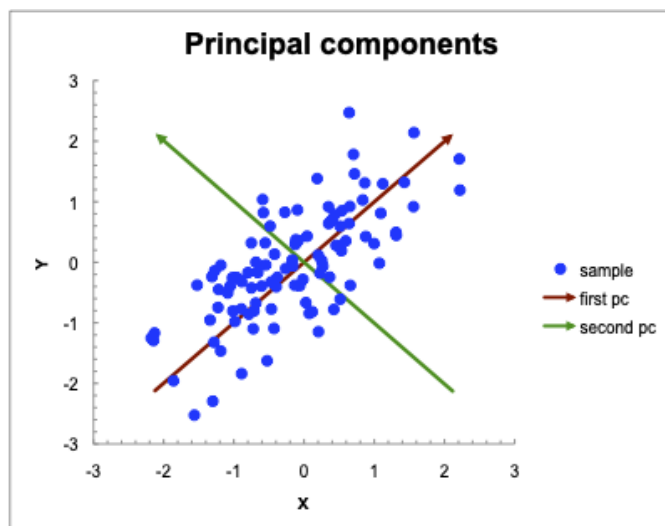
```
print(classification_report(y_test, model.predict(x_test)))
```

	precision	recall	f1-score	support
False	0.97	1.00	0.98	1926
True	0.81	0.18	0.29	74
accuracy			0.97	2000
macro avg	0.89	0.59	0.64	2000
weighted avg	0.96	0.97	0.96	2000

5.3 Principal component analysis

- **Principal Component Analysis (PCA)** summarises a large set of correlated variables by smaller number of representative variables that explain most of the variability of the original data set.
- It is a standard method for reducing the dimension of high dimensional, highly correlated systems.
- The principal components are common factors that are unobservable (latent) and directly estimated from the data.
- While being explanatory in a statistical sense, the factors do not necessarily have an economic interpretation.

PCA



PCA

- The objective is to take p random variables X_1, X_2, \dots, X_p and find (linear) combinations of these to produce random variables Z_1, \dots, Z_p , the **principal components**, that are uncorrelated.
- Geometrically, expressing X_1, \dots, X_p through linear combinations Z_1, \dots, Z_p can be thought of as shifting and rotating the axes of the coordinate system (see left graph).
- Hence, the transform leaves the data points unchanged, but expresses them using different coordinates.
- The lack of correlation is a useful property because it means that the principal components are measuring different "dimensions" of the data.

- The principal components can be ordered according to their variance, that is, $\text{Var}(Z_1) \geq \text{Var}(Z_2) \geq \dots \geq \text{Var}(Z_p)$
- If the variance captured in the higher dimensions is sufficiently small, then discarding those higher dimensions will retain most of the variability, so only little information is lost.

PCA

- Let (X, Y) be standard normally distributed random variables with a correlation of 0.7.
- The left graph above shows a scatterplot of a sample of 100 random numbers $(x_1, y_1), \dots, (x_{100}, y_{100})$.
- By shifting and rotating the axes, new variables Z_1, Z_2 , the principal components, with $Z_i = a_i X + b_i Y$ are obtained.
- The data sample expressed in terms of Z_1, Z_2 is uncorrelated.
- Also, the variance of the data contribution from the first principal component Z_1 is much greater than the variance contribution from the second principal component.
- The graph on the right shows the data points when the data are onto the first principal component, discarding the second dimensions.

PCA

- The sample variance of (x_1, \dots, x_n) is 0.9244 and the sample variance of (y_1, \dots, y_n) is 0.9226, whereas the sample variance of the data in the first principal component is 1.6014 and of the second principal component is 0.2456.
- In other words, while the original axes each account for roughly 50% of the total variance, the first principal component accounts for 87% of the sample variance.
- Neglecting the second principal component and expressing the data in the first principal component only retains 87% of the variance (see right graph).
- In practice, the number of dimensions will be higher, and one will choose the number of principal components to reflect a certain variance contribution such as 99%.
- If the data are sufficiently correlated, then only few dimensions (principal components) will be required even for high-dimensional data.

PCA example for interest rate term structure

- Interest rates of different maturities are known to exhibit large correlations.
- Interest rate term structures are therefore a good candidate for a representation by a few factors only.
- What do you think are the main ways in which an interest term structure moves over time?
- The data below is taken from

Hull: Machine Learning in Business. 3rd edition, independently published, 2021.

In [588]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import linalg
```

In [589]:

```
data=pd.read_excel("./data/Treasuries_Hull.xlsx", index_col=0, parse_dates=True)
data.head()
```

Out[589]:

	1yr	2yr	3yr	5yr	7yr	10yr	20yr	30yr
2010-01-04	0.45	1.09	1.66	2.65	3.36	3.85	4.60	4.65
2010-01-05	0.41	1.01	1.57	2.56	3.28	3.77	4.54	4.59
2010-01-06	0.40	1.01	1.60	2.60	3.33	3.85	4.63	4.70
2010-01-07	0.40	1.03	1.62	2.62	3.33	3.85	4.62	4.69
2010-01-08	0.37	0.96	1.56	2.57	3.31	3.83	4.61	4.70

- Our interest lies in movements of interest rate term structures, therefore we take first differences of the data.

In [590]:

```
d=100*data.diff();
d.dropna(inplace=True)
d.head()
```

Out[590]:

	1yr	2yr	3yr	5yr	7yr	10yr	20yr	30yr
2010-01-05	-4.0	-8.0	-9.0	-9.0	-8.0	-8.0	-6.0	-6.0
2010-01-06	-1.0	0.0	3.0	4.0	5.0	8.0	9.0	11.0
2010-01-07	0.0	2.0	2.0	2.0	0.0	0.0	-1.0	-1.0
2010-01-08	-3.0	-7.0	-6.0	-5.0	-2.0	-2.0	-1.0	1.0
2010-01-11	-2.0	-1.0	-1.0	1.0	1.0	2.0	3.0	4.0

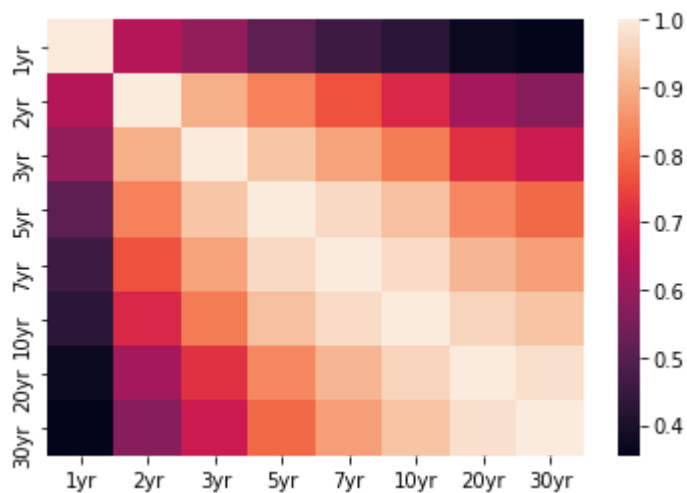
- Correlations of term structure movements:

In [591]:

```
sns.heatmap(d.corr())
```

Out[591]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7feb6e19e750>
```



- The principal components are the eigenvectors of the correlation matrix.
- These are also called factor loadings. They express the "weight" of each factor for each maturity.

In [592]:

```
w, vr=linalg.eig(d.corr()) # eigenvalues, eigenvectors
```

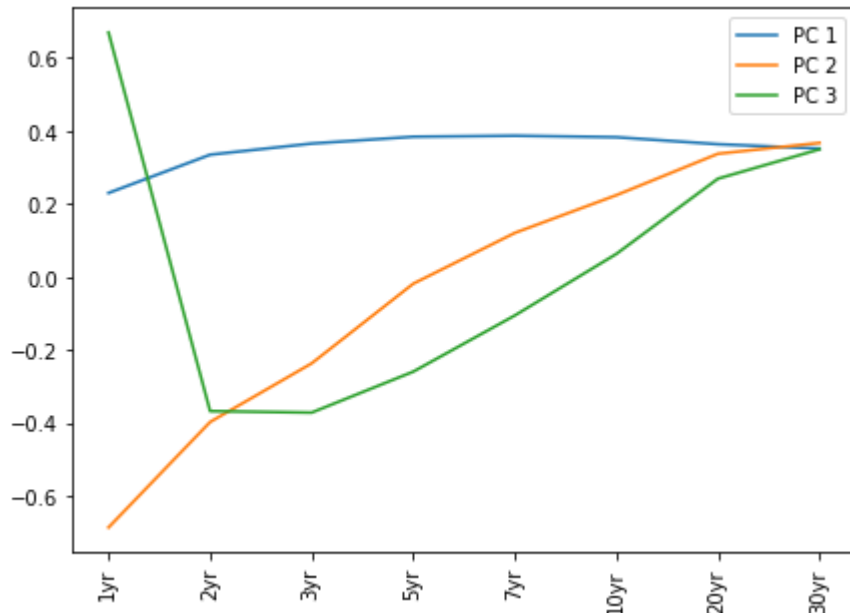
- The first PC captures changes in the level the term structure.
- The second PC captures changes in the slope.
- The third PC can be interpreted as a hump in the term structure.

In [593]:

```
plt.figure(figsize=(7,5))
plt.xticks(range(len(d.columns)), d.columns, rotation='vertical')
plt.plot(np.real(vr[:,0:3])); # first factor loadings
plt.legend(["PC 1", "PC 2", "PC 3"])
```

Out[593]:

<matplotlib.legend.Legend at 0x7feb6e386090>



- The eigenvalues express the variance captured by each PC.

In [594]:

```
np.real(100*w/w.sum()) # percentage variances
```

Out[594]:

```
array([79.2785935 , 12.44724531,  5.31159001,  1.67939721,  0.6688140
7,
       0.26405994,  0.18916142,  0.16113855])
```

- The principal component scores are the original data expressed in the PC coordinate system, dimension by dimension.
- These are obtained by multiplying each PC vector with the original data (de-meaned).
- Since the principal components are determined from maximising the variance explained by the model, this can be used to interpret the principal components.
- The correlation of each score with the original data allow for an interpretation of what each principal component represents.

In [595]:

```
pc1 = np.transpose(vr[:,0]*(d-d.mean())).sum()
pc2 = np.transpose(vr[:,1]*(d-d.mean())).sum()
pc3 = np.transpose(vr[:,2]*(d-d.mean())).sum()
pc4 = np.transpose(vr[:,3]*(d-d.mean())).sum()
pc5 = np.transpose(vr[:,4]*(d-d.mean())).sum()
```

- This gives the same result:

In [596]:

```
pc = np.transpose(np.matmul(np.transpose(vr), np.transpose(d.values-d.values.mean()))
```

In [597]:

```
for i in range(5):
    d.insert(i, 'pc' + str(i), pc[:,i])
```

- The plot below shows the correlations of each interest rate with the first five PC's.
- Note how the interpretation is similar to the factor loadings above.

In [598]:

```
plt.figure(figsize=(7,5))
sns.heatmap(np.transpose(d.corr().head(5)).iloc[5:])
```

Out[598]:

<matplotlib.axes._subplots.AxesSubplot at 0x7feb82b10a50>

