FitchLearning

Certificate in Quantitative Finance: Supervised Learning I and II

Author:
Steve Phelps
sphelps@sphelps.net

September 24, 2020

Contents

1 Supervised Learning

Steve Phelps

1.1 Outline

1. Introduction
2. Regression methods
    - Linear Regression
    - Penalized Regressions: Lasso, Ridge and Elastic Net
    - Polynomial regression and kernel methods
3. Classification methods
    - Support Vector Machines
    - Bayesian Classification
    - K Nearest Neighbors
4. Logistic regression

1.2 Machine learning

- Nearly all machine-learning problem can be posed as *optimisation problems* applied to *data*.
- We define a *loss function*: $L : (\mathbf{x}, F_{\mathbf{w}}) \rightarrow \mathbb{R}$

where $\mathbf{x}$ is some data, and $F_{\mathbf{w}}(\mathbf{x})$ is an arbitrary function parameterised by numeric weights $\mathbf{w}$.

- F can be non-linear and/or stochastic.
- We minimise the expected loss by choosing appropriate weights:

$$\underset{\mathbf{w}}{\operatorname{argmin}} E[L(\mathbf{x}, F_{\mathbf{w}})] \tag{1.1}$$

- In statistical learning we sometimes use the word "coefficient" instead of "weight", but the meaning is identical.

1.3 Supervised versus unsupervised learning

- In a supervised learning problem we have some additional *training data* $\mathbf{y}$.
- The training data provides a finite subset of observed values $\mathbf{y}$ from an unknown function F, such that $\mathbf{y} = F(\mathbf{x}) + \epsilon$

where ϵ are iid. random noise.

- We want to generalise from the training data to derive the complete mapping F.
- This can then used to generalise to unseen cases where no training data is available.
- In a supervised learning problem therefore, our loss function takes the form $L : (\mathbf{x}, \mathbf{y}, \hat{F}_{\mathbf{w}}) \rightarrow \mathbb{R}$.
- Intuitively, we should define a loss function that tells us how good the fit is between the function $\hat{F}$ and the observations of F.

1.4 Statistical and machine-learning

- Standard methods in statistics can also be viewed as optimisation problems.
- For example, in ordinary least-squares estimation (OLS) the loss function is simply the sum of the squared residuals (RSS):

$$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^{n} r_i^2 \tag{1.2}$$

$$r_i = y_i - F_{\mathbf{w}}(x_i) \tag{1.3}$$

$$F_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^T \mathbf{w} \tag{1.4}$$

- In the above, the loss function is a simple linear function.
- The OLS loss function is very straightforward to minimize using calculus.
- In general, however, machine-learning methods allow us to work with arbitrary *non-linear* functions.

1.5 Practical machine learning

- Machine learning has recently become very practical, because:
- we now have very large data sets,
- we have very good algorithms for performing *automated* differentiation of *arbitrary* functions,
- we have heuristic optimization algorithms for minimizing arbitrary functions even when they are not well behaved,

1.6 Practical machine learning: GPUs

Graphical-Processing Units GPUs can perform parallel computation on multi-dimensional numerical data using thousands or tens of thousands of cores simultaneously.

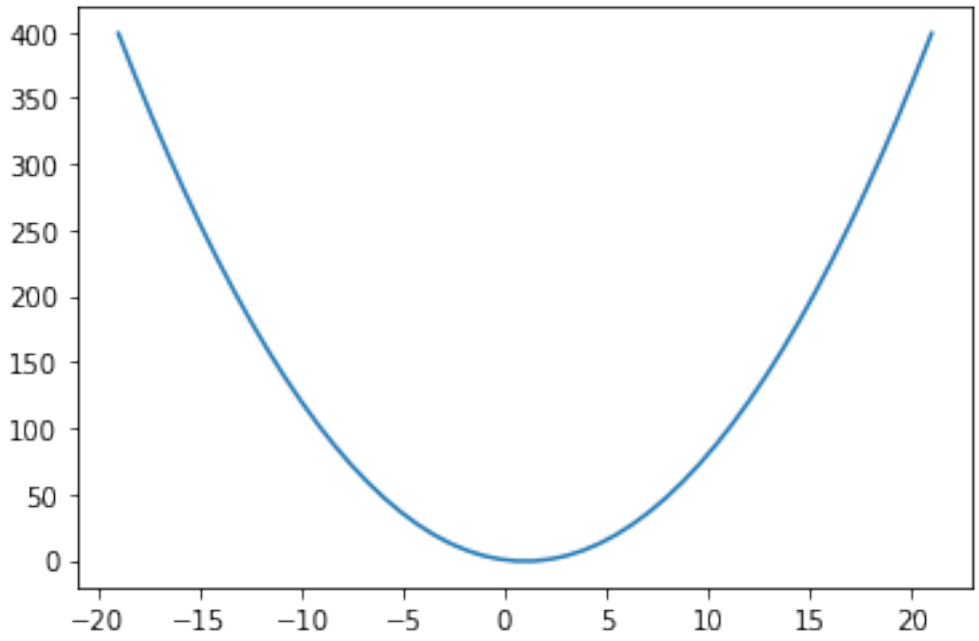
Figure 1.1: GPU

1.7 Python for machine-learning

- There are many powerful tools for machine-learning which are available in the Python programming language:
  - scikit-learn
  - TensorFlow
  - scipy

1.8 Optimizing a simple function in Python

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

def F(x):
    return x**2 - 2*x

domain = np.linspace(-19., +21.0)
plt.plot(domain, F(domain)); plt.show()
```



1.8.1 Using gradient descent

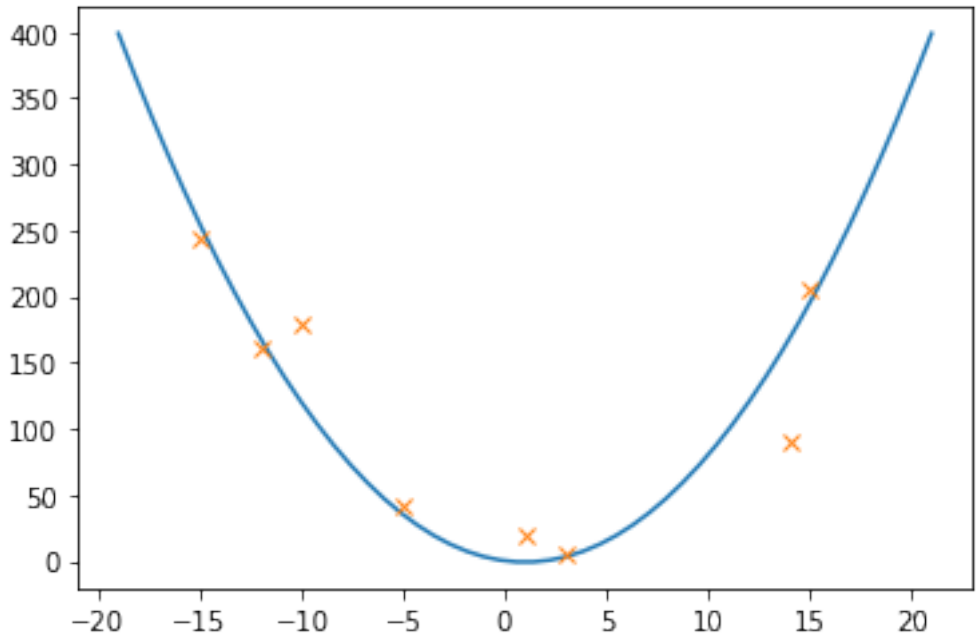
```
import scipy.optimize as opt

initial_guess = [20.]
solution = opt.minimize(F, initial_guess)
solution.x
```

```
array([1.00000009])
```

1.9 Generating some training data

- Now suppose we don't simply want to the minimum of F, instead we want to find F *itself*, given some finite noisy observations $\mathbf{y}$.

```
data = np.array([-15, -12, -5, 1., 3., -10, +14, +15])
training_data = np.array([F(x) + np.random.normal(scale=30.) for x
    ↪ in data])
plt.plot(domain, F(domain))
plt.plot(data, training_data, 'x')
plt.show()
```



1.10 Optimising a simple loss function in Python

- Let's assume that F is a polynomial of order 2.

```
def F_hat(x, w):
    (a, b, c) = w
    return a + b*x + c*x**2
```

- Now we define our loss function, which in this case is simply the RSS:

```
def L(w):
    predicted_data = np.array([F_hat(x, w) for x in data])
    return np.sum((training_data - predicted_data) ** 2)
```

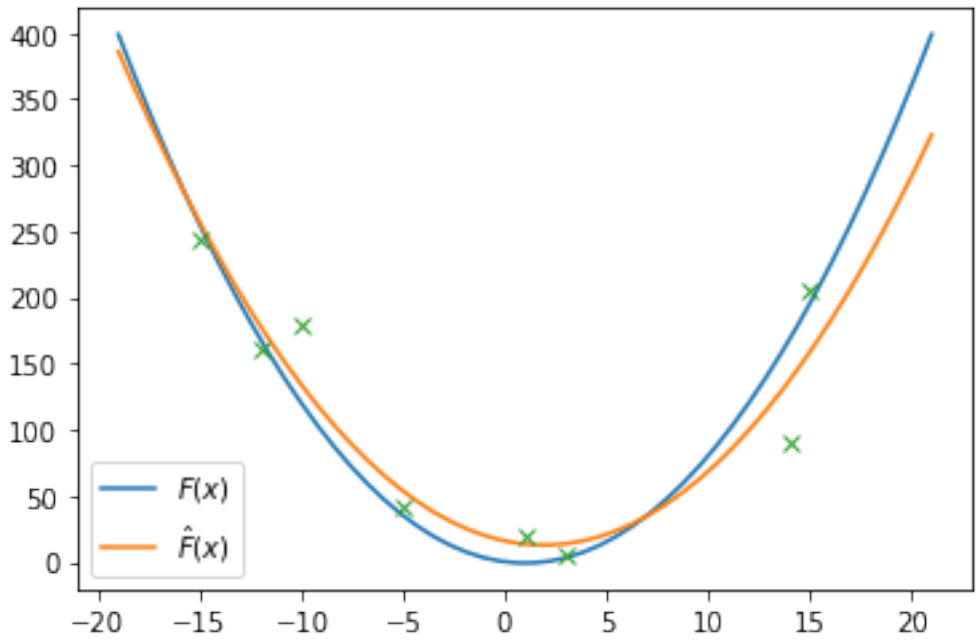
- Finally we optimize the loss function given some initial weights:

```
initial_weights = [0., 0., 0.]
final_weights = opt.minimize(L, initial_weights)
final_weights.x
```

```
array([15.77284701, -3.27938592,  0.85303416])
```

1.10.1 Evaluating the model

```
plt.plot(domain, F(domain))
plt.plot(domain, F_hat(domain, final_weights.x))
plt.plot(data, training_data, 'x')
plt.legend(['$F(x)$', '$\hat{F}(x)$'])
plt.show()
```



1.11 Statistical and machine learning in finance

- Estimating the single-index model to derive alpha and beta.
- Estimating default probabilities in credit-risk models.
- Solving stochastic-control problems for algorithmic trading.
- Time-series forecasting.
- Predicting excess returns for portfolio allocation.
- Asset rating models.

2 Regression methods

2.1 Simple linear regression

- Consider a vector $\mathbf{x}$ containing samples from our predictor, and a vector of $\mathbf{y}$ of corresponding responses, both of size n.
- We assume a stochastic linear relationship between response and predictor: $y = F(x) = \alpha + \beta x + \epsilon$,
- Our weight vector simply consists of our estimated intercept and slope: $\mathbf{w} = (\hat{\alpha}, \hat{\beta})$
- Our predictions are given by:

$$\hat{F}_{(\hat{\alpha},\hat{\beta})} = \hat{\alpha} + \hat{\beta}x \tag{2.1}$$

- The residuals are the differences between our predictions and the observed training response:

$$r_i = y_i - F_{\mathbf{w}}(x_i) \tag{2.2}$$

- The loss function L is the residual sum of squares (RSS):

$$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^{n} r_i^2 \tag{2.3}$$

2.1.1 Assumptions

1. $F(x)$ is linear.
2. The noise term is i.d.d. normally distributed with a mean 0 and constant variance:

$$\epsilon \sim N(0, \sigma_\epsilon^2) \tag{2.4}$$

3. The independent variable is uncorrelated with the noise $E[x_{ij}\epsilon_i] = 0$.

2.1.2 Least squares estimates

- We want to maximise the goodness-of-fit by minimising the expected loss:

$$\underset{\mathbf{w}}{\operatorname{argmin}} E[L(\mathbf{x}, F_{\mathbf{w}})] \tag{2.5}$$

where $\mathbf{w} = (\hat{\alpha}, \hat{\beta})$

- In this case we can use calculus to obtain a closed-form solution without resorting to computational methods:

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.6}$$

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x} \tag{2.7}$$

where $\bar{x}$ and $\bar{y}$ are the sample means.

2.1.3 Applications to Finance - The Single Index Model

$$r_{i,t} - r_f = \alpha_i + \beta_i(r_{m,t} - r_f) + \epsilon_{i,t} \tag{2.8}$$

$$\epsilon_{i,t} \sim N(0, \sigma_i) \tag{2.9}$$

- $r_{i,t}$ is return to stock i in period t.
- r_f is the risk-free rate.
- $r_{m,t}$ is the return to the market portfolio.

Elton, E. J., & Gruber, M. J. (1997). *Modern portfolio theory, 1950 to date*. Journal of Banking and Finance, 21(11–12), 1743–1759. https://doi.org/10.1016/S0378-4266(97)00048-4

2.1.3.1 Loading data into a pandas dataframe

- We will first obtain some data from Yahoo finance using the pandas library.
- First we will import the functions and modules we need.

```
import matplotlib.pyplot as plt
import datetime
import pandas as pd
import numpy as np
```

2.1.3.2 Downloading price data using as CSV

- Here we obtain price data on Microsoft Corporation Common Stock, so we specify the symbol MSFT.

```
def prices_from_csv(fname):
    df = pd.read_csv(fname)
    df.set_index(pd.to_datetime(df['Date']), inplace=True)
    return df
```

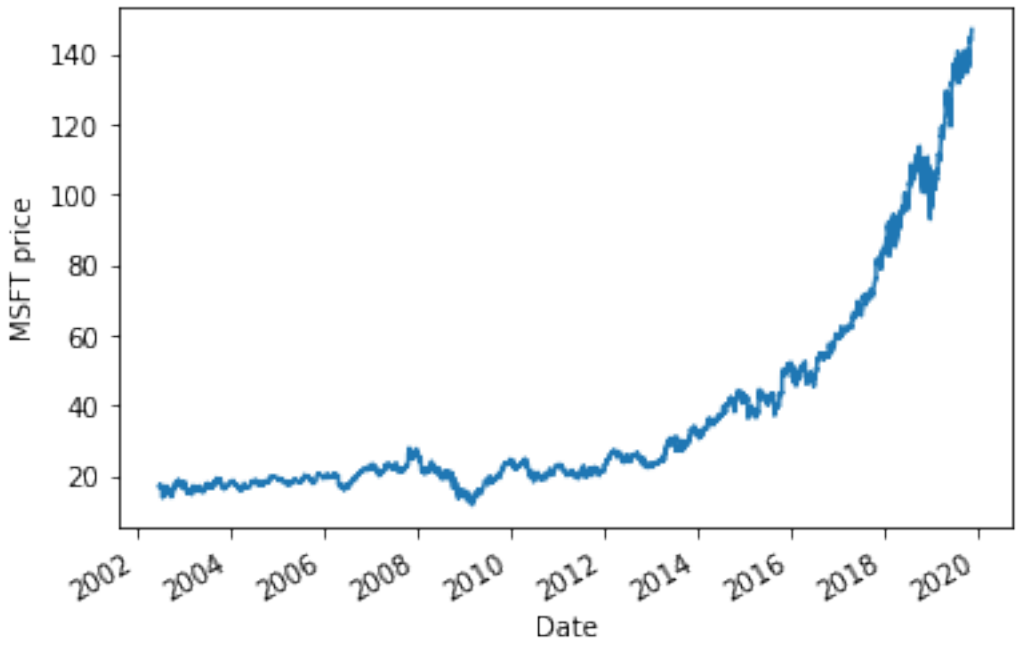
```
msft = prices_from_csv('data/MSFT.csv')
msft.head()
```

```
                  Date        Open        High         Low       Close
Adj Close  \
Date
2002-07-01  2002-07-01  27.059999  27.195000  26.290001  26.330000
16.997240
2002-07-02  2002-07-02  26.190001  26.459999  25.665001  25.719999
16.603462
2002-07-03  2002-07-03  25.620001  26.260000  25.225000  25.920000
16.732574
2002-07-05  2002-07-05  26.545000  27.450001  26.525000  27.424999
17.704121
2002-07-08  2002-07-08  27.205000  27.465000  26.290001  26.459999
17.081167

              Volume
Date
2002-07-01  66473800
2002-07-02  82814200
2002-07-03  80936600
```

```
2002-07-05   35673600
2002-07-08   63199400
```

2.1.3.3 Plotting the price of the stock

```
msft['Adj Close'].plot()
plt.ylabel('MSFT price')
plt.show()
```

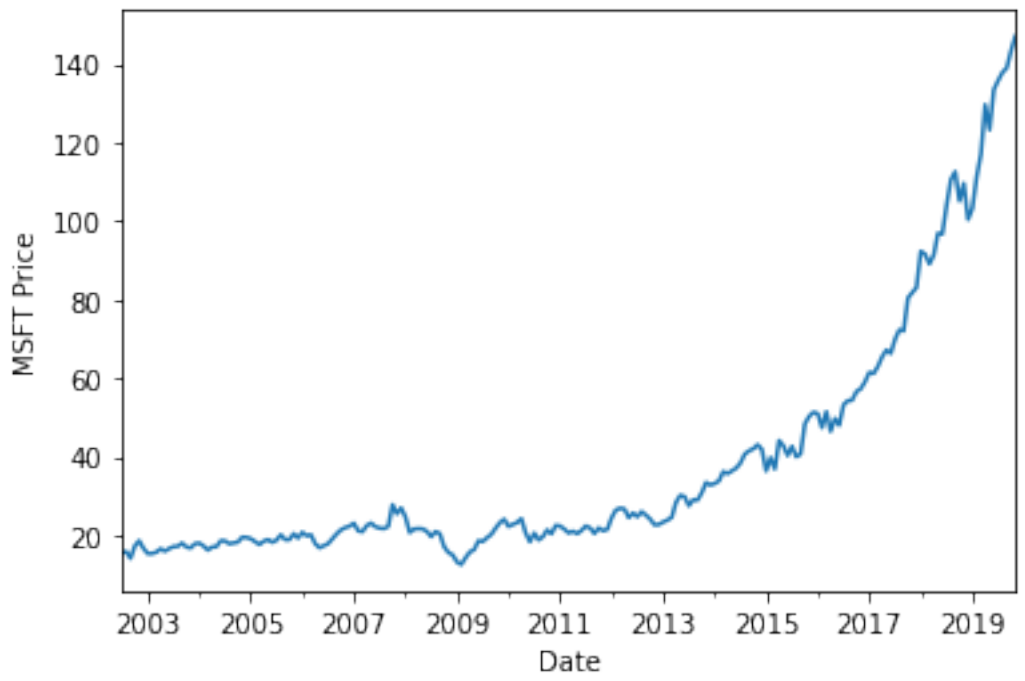


2.1.3.4 Converting to monthly data

- We will resample the data at a frequency of one calendar month.
- The code below takes the last price in every month.

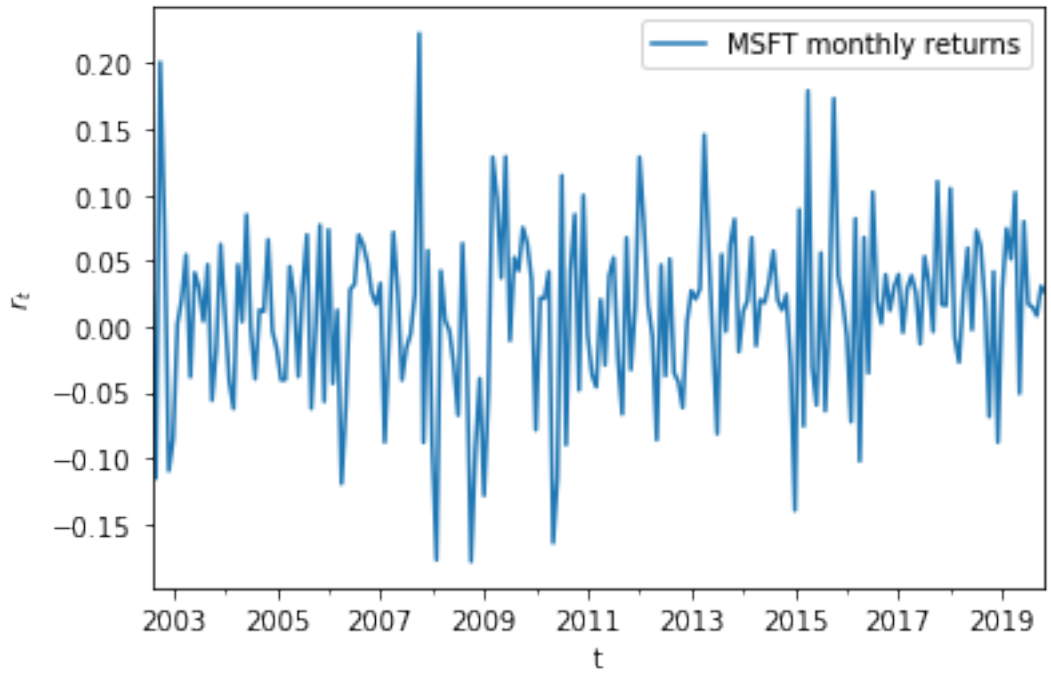
```
daily_prices = msft['Adj Close']
```

```
monthly_prices = daily_prices.resample('M').last()
monthly_prices.plot()
plt.ylabel('MSFT Price')
plt.show()
```



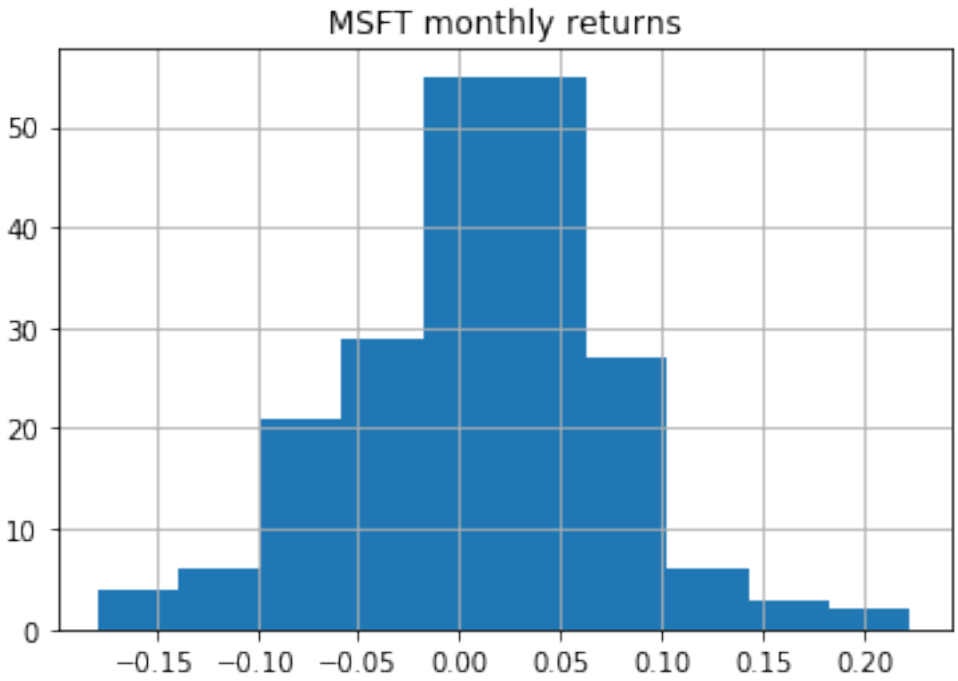
2.1.3.5 Calculating log returns

```
stock_returns = pd.DataFrame({'MSFT monthly returns': np.log(
    ↪ monthly_prices).diff().dropna()})
stock_returns.plot()
plt.xlabel('t'); plt.ylabel('$r_t$')
plt.show()
```



2.1.3.6 Return histogram

```
stock_returns.hist()
plt.show()
```



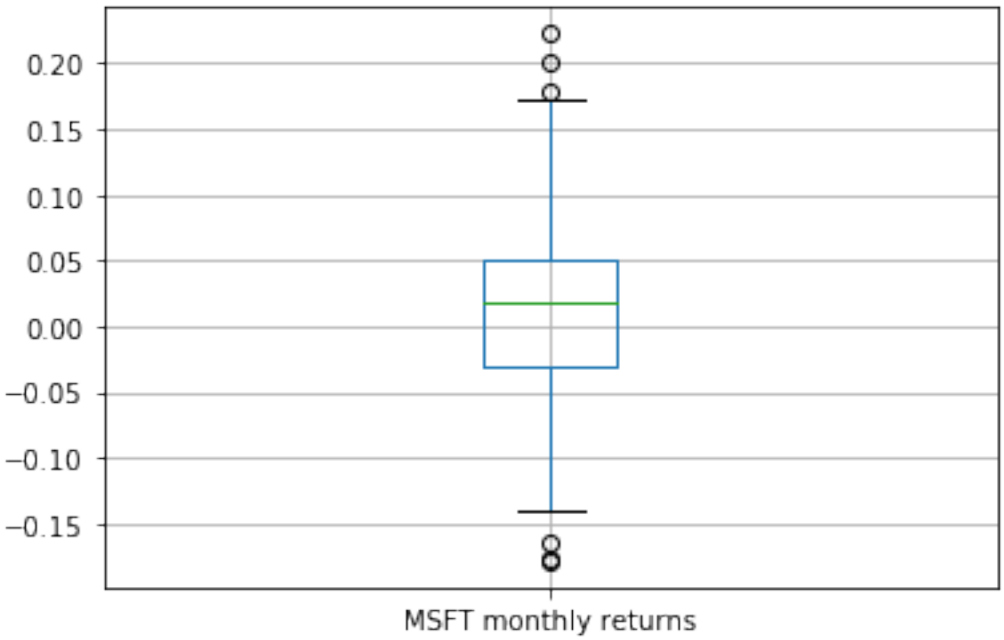
2.1.3.7 Descriptive statistics of the return distribution

```
stock_returns.describe()
```

```
       MSFT monthly returns
count            208.000000
mean               0.010822
std                0.064673
min               -0.178358
25%               -0.031284
50%                0.018196
75%                0.051398
max                0.222736
```

2.1.3.8 Summarising the distribution using a boxplot

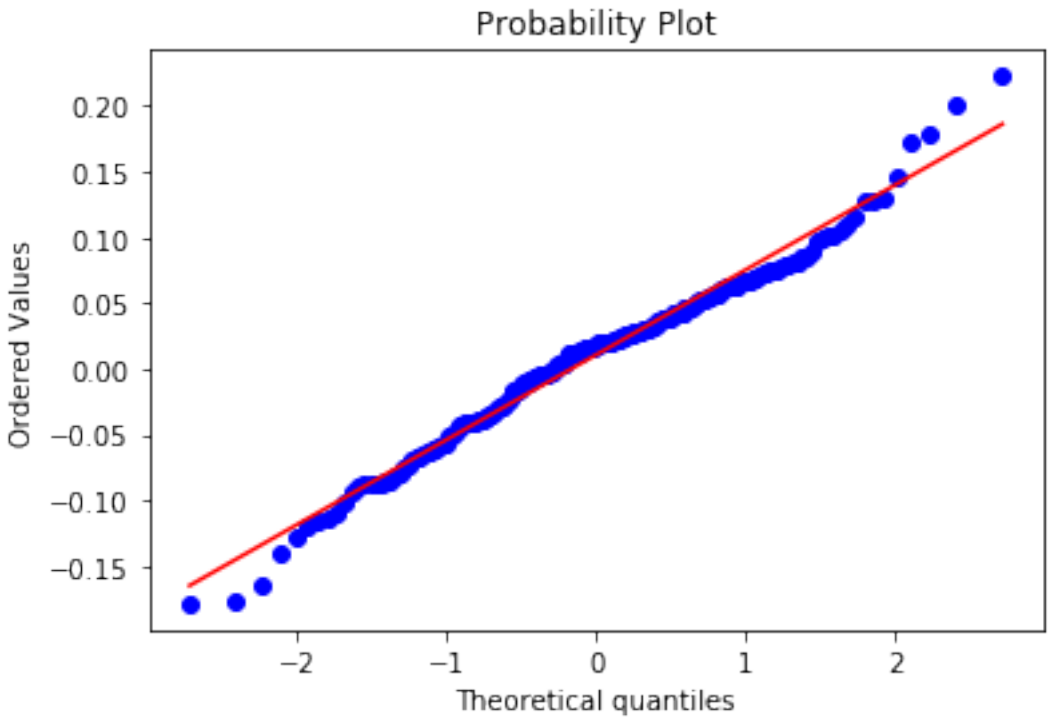
```
stock_returns.boxplot()
plt.show()
```



2.1.3.9 Q-Q plots

- Quantile-Quantile (Q-Q) plots are a useful way to compare distributions.
- We plot empirical quantiles against the quantiles computed the inverted c.d.f. of a specified theoretical distribution.

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

stats.probplot(stock_returns.values[:,0], dist="norm", plot=plt)
plt.show()
```



2.1.3.10 The market index

- We will first obtain data on the market index: in this case the NASDAQ:

```
nasdaq_index = prices_from_csv('data/^NDX.csv')
nasdaq_index.head()
```

```
                  Date          Open          High           Low
Close  \
Date
2002-07-01  2002-07-01  1044.479980  1049.880005   997.969971
998.169983
2002-07-02  2002-07-02   989.250000   993.989990   961.760010
963.659973
2002-07-03  2002-07-03   957.260010   995.950012   950.330017
995.679993
2002-07-05  2002-07-05  1018.630005  1061.050049  1018.630005
1060.890015
2002-07-08  2002-07-08  1051.270020  1066.280029  1008.780029
1014.330017

              Adj Close      Volume
Date
2002-07-01   998.169983  2320650000
2002-07-02   963.659973  2722550000
2002-07-03   995.679993  2661060000
2002-07-05  1060.890015  1120960000
2002-07-08  1014.330017  1708150000
```

2.1.3.11 Converting to monthly data

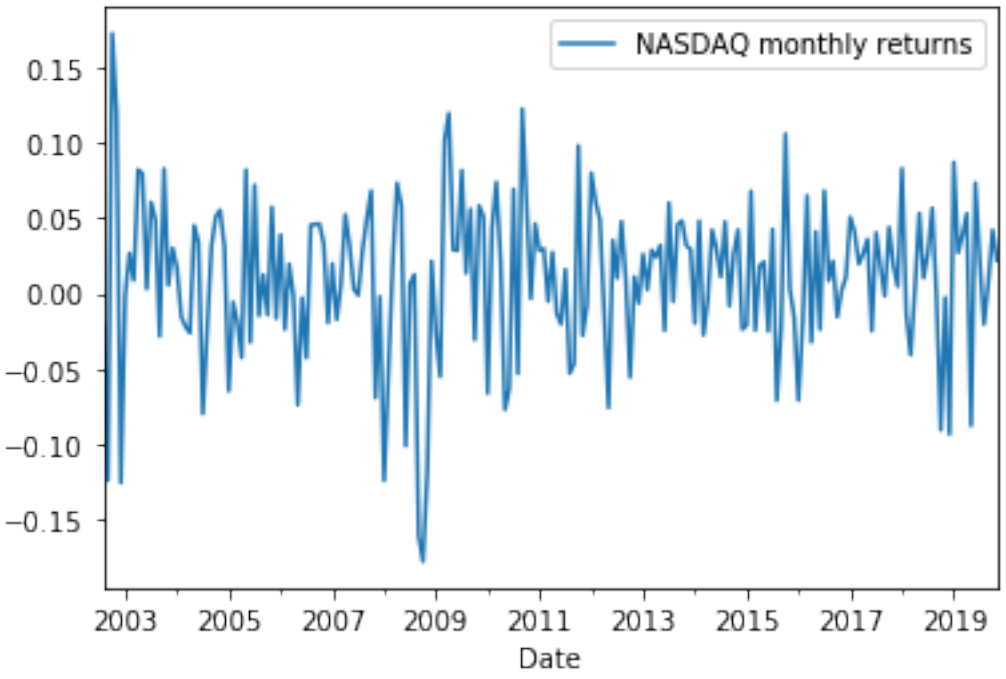
- As before, we can resample to obtain monthly data.

```
nasdaq_monthly_prices = nasdaq_index['Adj Close'].resample('M').
    ↪ last()
nasdaq_monthly_prices.head()
```

```
Date
2002-07-31      962.099976
2002-08-31      942.380005
2002-09-30      832.520020
2002-10-31      989.539978
2002-11-30     1116.099976
Freq: M, Name: Adj Close, dtype: float64
```

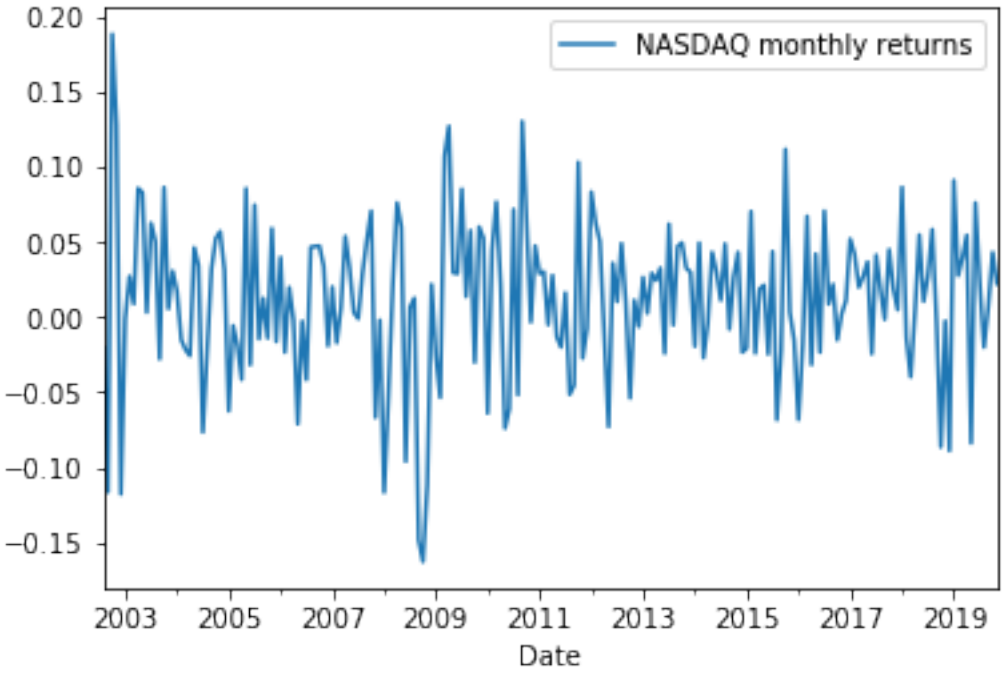
2.1.3.12 Plotting monthly returns

```
index_log_returns = pd.DataFrame({'NASDAQ monthly returns': np.log(
    ↪ nasdaq_monthly_prices).diff().dropna()})
index_log_returns.plot()
plt.show()
```



2.1.3.13 Converting to simple returns

```
index_simple_returns = np.exp(index_log_returns) - 1.
index_simple_returns.plot()
plt.show()
```




```
stock_simple_returns = np.exp(stock_returns) - 1.
```

2.1.3.14 Concatenating data into a single data frame

- We will now concatenate the data into a single data fame.
- We can use `pd.concat()`, specifying an axis of 1 to merge data along columns.
- This is analogous to performing a `zip()` operation.

```
comparison_df = pd.concat([index_simple_returns,
    ↪ stock_simple_returns], axis=1)
comparison_df.head()
```

```
            NASDAQ monthly returns  MSFT monthly returns
Date
2002-08-31               -0.020497              0.022926
2002-09-30               -0.116577             -0.108802
2002-10-31                0.188608              0.222450
2002-11-30                0.127898              0.078736
2002-12-31               -0.118027             -0.103676
```

2.1.3.15 Scatter plots

- We can produce a scatter plot to see whether there is any relationship between the stock returns, and the index returns.
- There are two ways to do this:

  1. Use the function `scatter()` in `matplotlib.pyplot`
  2. Invoke the `plot()` method on a data frame, passing `kind='scatter'`

2.1.3.16 Scatter plots using the `plot()` method of a data frame

- In the example below, the `x` and `y` named arguments refer to column numbers of the data frame.
- Notice that the `plot()` method is able to infer the labels automatically.

```
comparison_df.plot(x=0, y=1, kind='scatter')
plt.show()
```

2.1.3.17 Computing the correlation matrix

- For random variables X and Y, the Pearson correlation coefficient is:

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} \tag{2.10}$$

$$= \frac{E[(X-\mu_x)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{2.11}$$

$$\tag{2.12}$$

2.1.3.18 Covariance and correlation of a data frame

- We can invoke the `cov()` and `corr()` methods on a data frame.

```
comparison_df.cov()
```

```
                        NASDAQ monthly returns  MSFT monthly returns
NASDAQ monthly returns                0.002675              0.002243
MSFT monthly returns                  0.002243              0.004283
```

```
comparison_df.corr()
```

```
                         NASDAQ monthly returns  MSFT monthly returns
NASDAQ monthly returns                 1.000000              0.662686
MSFT monthly returns                   0.662686              1.000000
```

2.1.3.19 Comparing multiple attributes in a data frame

- It is often useful to work with more than two variables.
- We can add columns (attributes) to our data frame.
- Many of the methods we are using will automatically incorporate the additional variables into the analysis.

2.1.3.20 Using a function to compute returns

- The code below defines a function defines a function which will return a data frame containing a single series of returns for the specified symbol, and sampled over the specified frequency.

```
def returns_df(symbol, frequency='M'):
    df = prices_from_csv('data/%s.csv' % symbol)
    prices = df['Adj Close'].resample(frequency).last()
    column_name = symbol + ' returns (' + frequency + ')'
    return pd.DataFrame({column_name: np.exp(np.log(prices).diff().
    ↪ dropna()) - 1.})
```

```
apple_returns = returns_df('AAPL')
apple_returns.head()
```

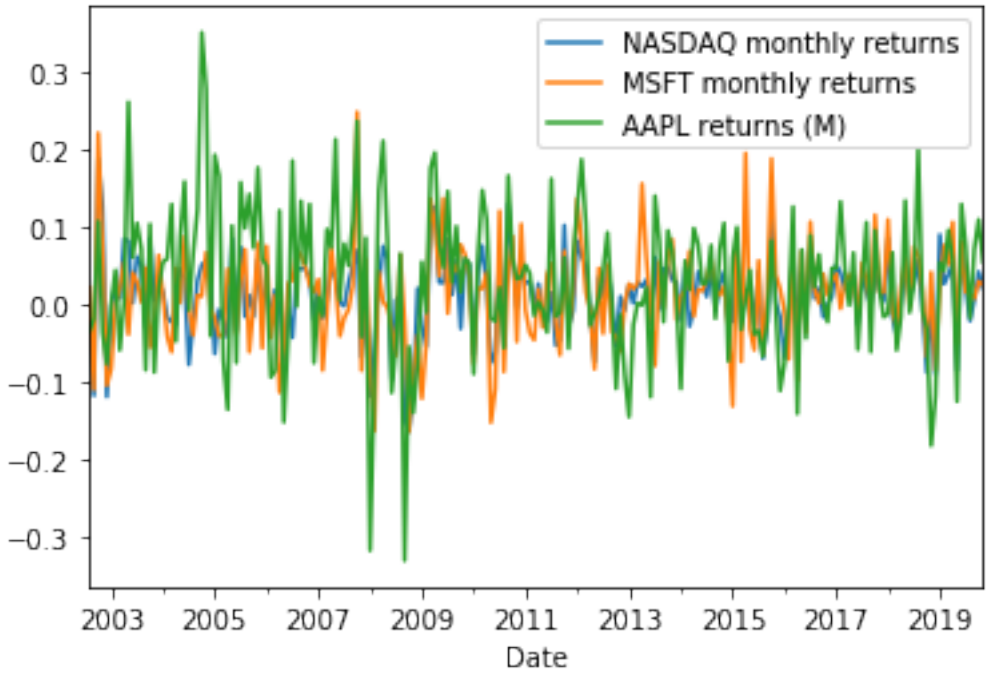
```
            AAPL returns (M)
Date
2002-08-31         -0.033421
2002-09-30         -0.016949
2002-10-31          0.108276
2002-11-30         -0.035470
2002-12-31         -0.075484
```

2.1.3.21 Adding another stock to the portfolio

```
comparison_df = pd.concat([comparison_df, apple_returns], axis=1)
comparison_df.head()
```

```
            NASDAQ monthly returns  MSFT monthly returns  AAPL returns
 (M)
Date
2002-08-31               -0.020497              0.022926         -
0.033421
2002-09-30               -0.116577             -0.108802         -
0.016949
2002-10-31                0.188608              0.222450
0.108276
2002-11-30                0.127898              0.078736         -
0.035470
2002-12-31               -0.118027             -0.103676         -
0.075484
```

```
comparison_df.plot()
plt.show()
```

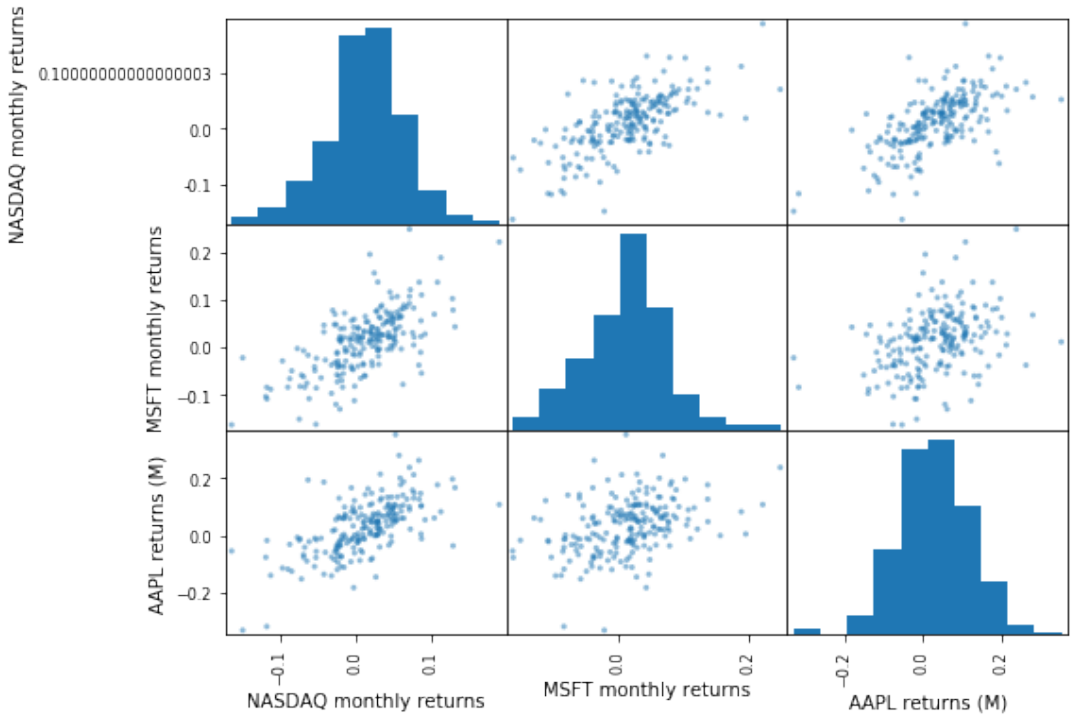



```
comparison_df.corr()
```

```
                        NASDAQ monthly returns  MSFT monthly returns
\
NASDAQ monthly returns                1.000000              0.662686
MSFT monthly returns                  0.662686              1.000000
AAPL returns (M)                      0.629116              0.375185

                        AAPL returns (M)
NASDAQ monthly returns          0.629116
MSFT monthly returns            0.375185
AAPL returns (M)                1.000000
```

2.1.3.22 Scatter matrices

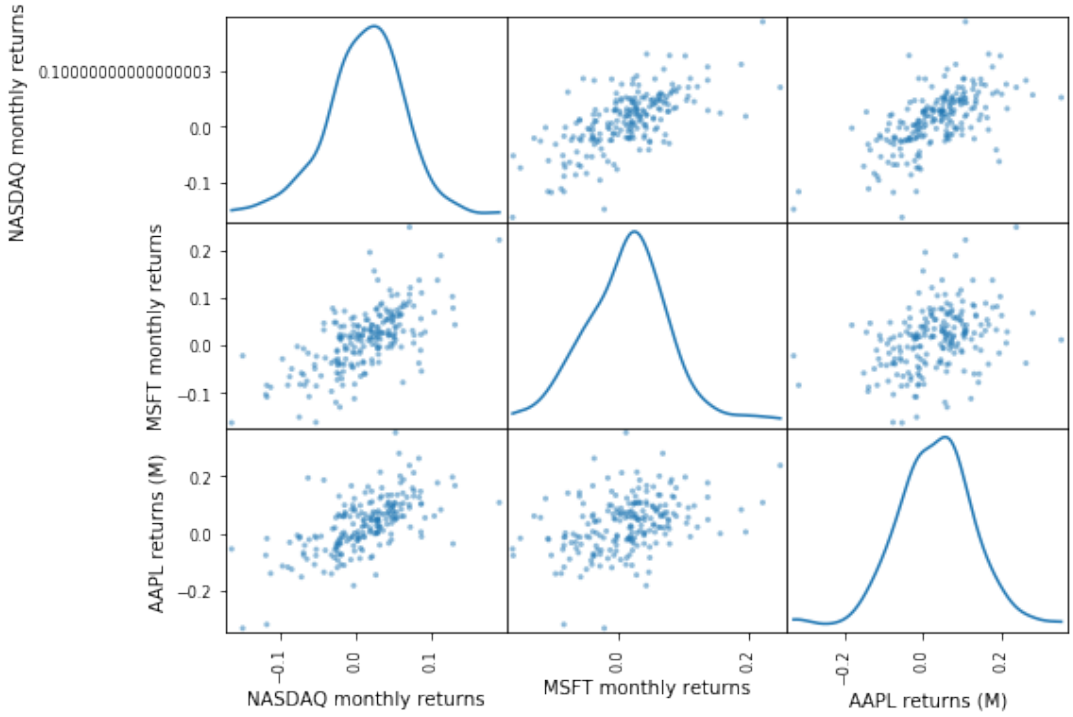
```
pd.plotting.scatter_matrix(comparison_df, figsize=(8, 6))
plt.show()
```



2.1.3.23 Scatter matrices with Kernel-density plots

- We can use Kernel density estimation (KDE) to plot an approximation of the pdf.

```
pd.plotting.scatter_matrix(comparison_df, diagonal='kde', figsize
    ↪ =(8, 6))
plt.show()
```



2.1.4 Ordinary-least squares estimation in scikit-learn

- First we import the required modules:

```
from sklearn.preprocessing import scale
import sklearn.linear_model as skl_lm
from sklearn.metrics import mean_squared_error, r2_score
```

- Now we prepare the data set:

```
rr = 0.01 # risk-free rate
ydata = stock_simple_returns - rr
xdata = index_simple_returns - rr
```

- Finally we fit the model

```
regr = skl_lm.LinearRegression()
regr.fit(xdata, ydata)

alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
print(alpha);  print(beta)
```

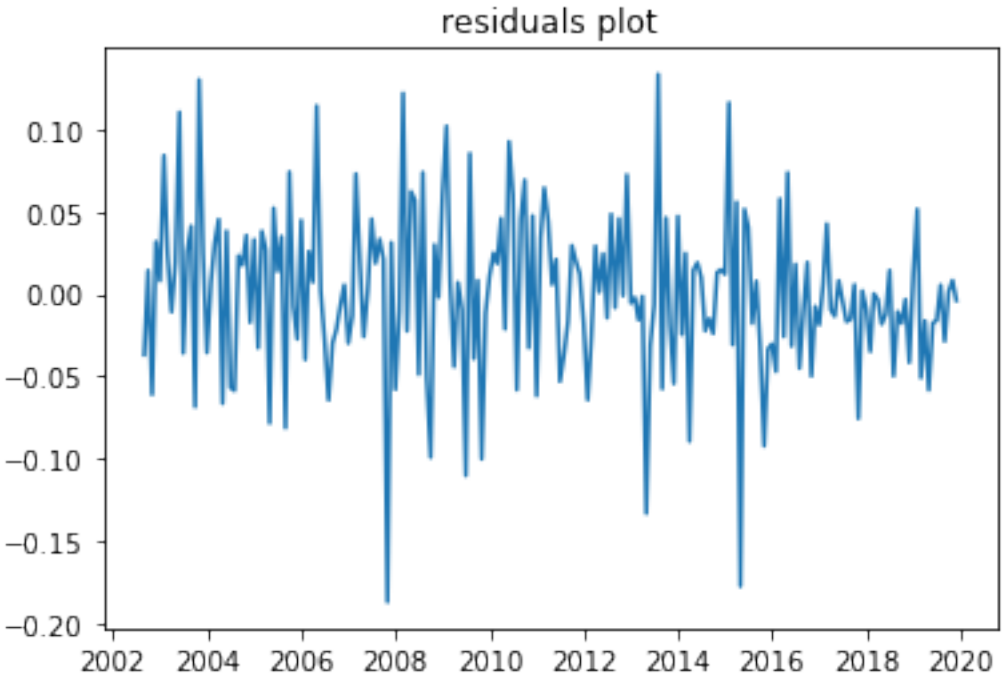
```
0.0015326890415947843
0.8385169376111147
```

2.1.5 Plotting the fitted model

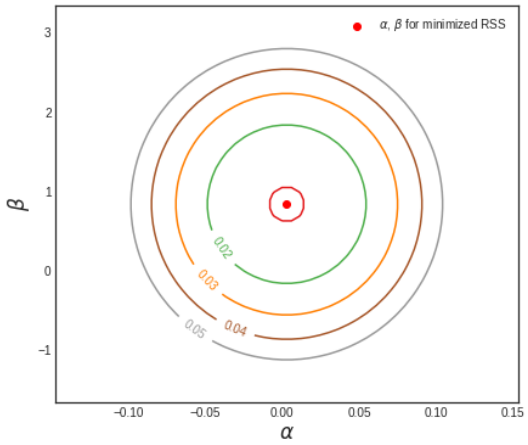
```
plt.scatter(x=xdata, y=ydata)
plt.plot(xdata, alpha + beta * xdata)
plt.xlabel('index return')
plt.ylabel('stock return')
plt.title('Single-index model fit ')
plt.show()
```

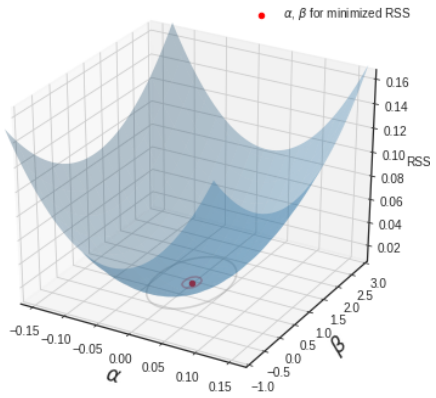
2.1.6 Plotting the residuals

```
residuals = regr.predict(xdata) - ydata
plt.plot(residuals)
plt.title('residuals plot')
plt.show()
```



2.1.7 RSS contours and surface





2.2 Confidence intervals and standard error

- Notice that the optimal values of α and β can be expressed as functions of sample means.
- What happens when we use different training data, i.e. different samples of the training data?

2.2.1 Estimation error

- By the law of large numbers $\lim_{n\to\infty} \bar{\mathbf{x}} = E(X)$.
- However, for finite values of n we will have an estimation error.
- Can we quantify the estimation error as a function of n?

2.2.2 Computing the error numerically

- If we draw from a standard normal distribution, we know that $E(X) = 0$.
- Therefore we can easily compute the estimation error in any given sample.

2.2.3 The error for a small random sample.

- Here $X \sim N(0, 1)$, and we draw a random sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of size $n = 5$.
- We will compute $\epsilon_{\mathbf{x}} = |\bar{\mathbf{x}} - E(X)| = |\bar{\mathbf{x}}|$

```
x = np.random.normal(size=5)
x
```

```
array([ 0.00271478, -1.84092167, -0.51635261,  1.60569759, -
0.88543057])
```

```
np.mean(x)
```

```
-
0.3268584941334501
```

```
estimation_error = np.sqrt(np.mean(x)**2)
estimation_error
```

```
0.3268584941334501
```

2.2.4 Variance in the error

- If we draw a different sample, will the error be different or the same?

```
x = np.random.normal(size=5)
estimation_error = np.mean(x)**2
estimation_error
```

```
0.5909299506743294
```

```
x = np.random.normal(size=5)
estimation_error = np.mean(x)**2
estimation_error
```

```
1.2763231903064298
```

```
x = np.random.normal(size=5)
estimation_error = np.mean(x)**2
estimation_error
```

```
0.006058981922165296
```

2.2.5 Expected error

- The error $\epsilon_{\mathbf{x}}$ is itself a random variable.
- How can we compute $E(\epsilon_{\mathbf{x}})$?

2.2.6 Monte-Carlo estimation of the sampling error

```
def sampling_error(n):
    errors = [np.sqrt(np.mean(np.random.normal(size=n))**2) \
              for i in range(100000)]
    return np.mean(errors)

sampling_error(5)
```

```
0.35633505946280947
```

- Notice that this estimate is relatively stable:

```
sampling_error(5)
```

```
0.35745623640845764
```

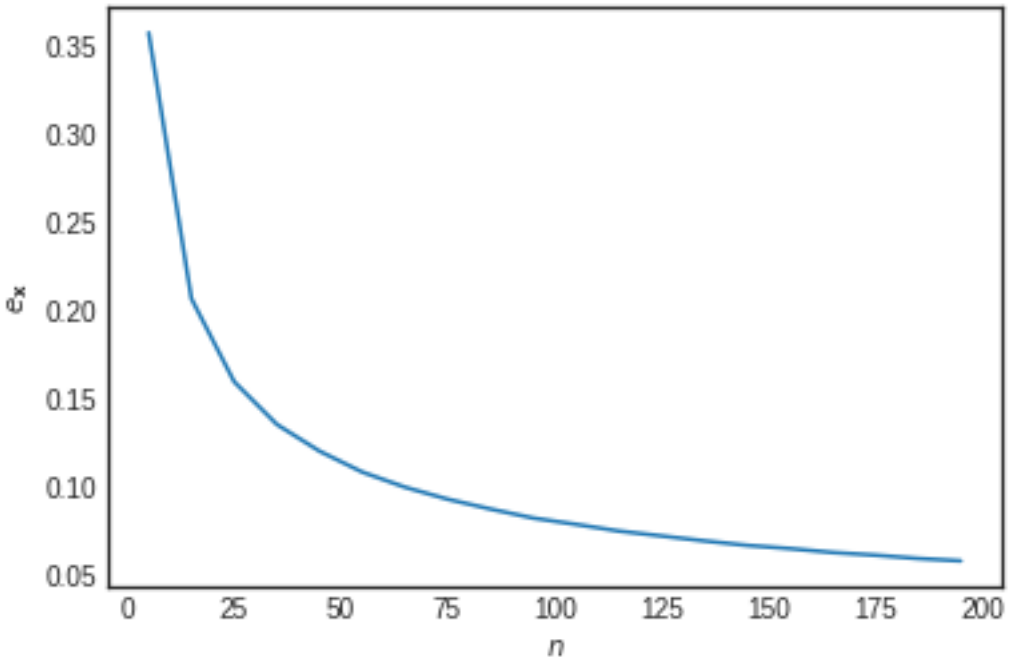
```
sampling_error(5)
```

```
0.3560285000676012
```

2.2.7 Monte-Caro estimation of the standard error

- We can now examine the relationship between sample size n and the expected error using a Monte-Carlo method.

```
import matplotlib.pyplot as plt
n = np.arange(5, 200, 10)
plt.plot(n, np.vectorize(sampling_error)(n))
plt.xlabel('$n$');  plt.ylabel('$e_\mathbf{x}$')
plt.show()
```



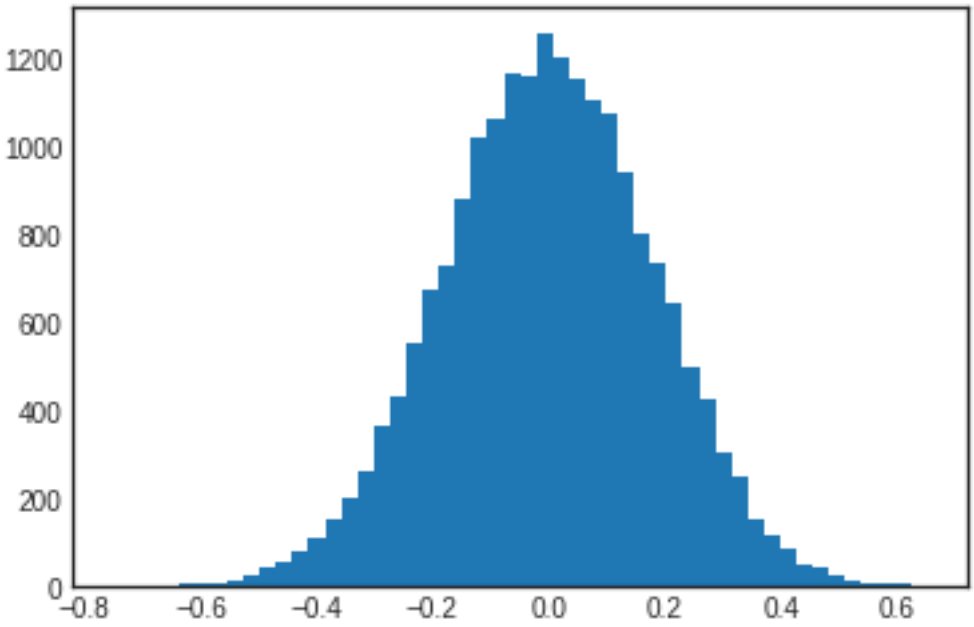
2.2.8 The sampling distribution of the mean

- The variance in the error occurs because the sample mean is a random variable.
- What is the distribution of the sample mean?

2.2.9 The sampling distribution of the mean

- Let's fix the sample size at $n = 30$, and look at the empirical distribution of the sample means.

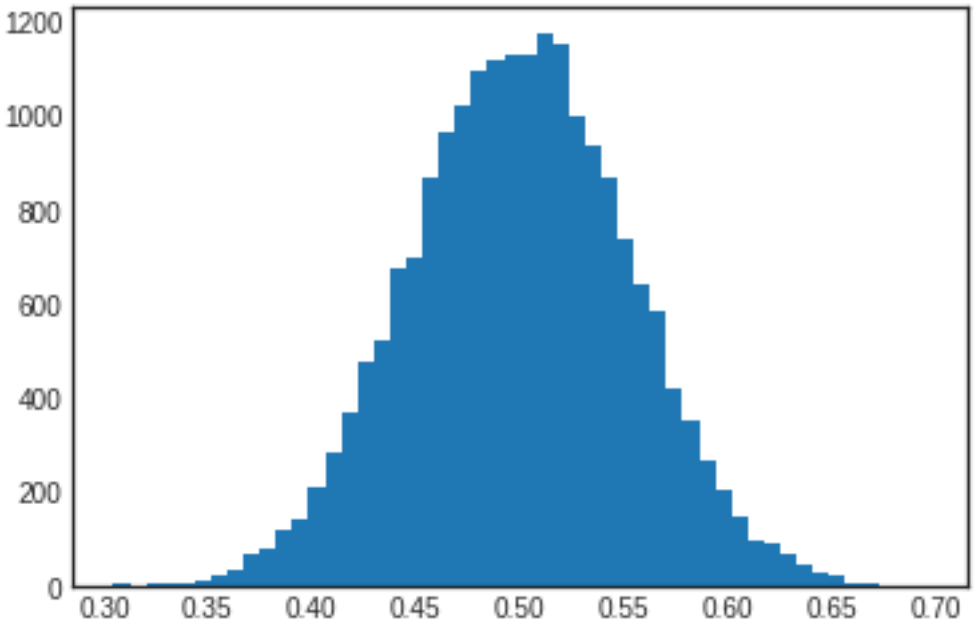
```
# Sample size
n = 30
# Number of repeated samples
N = 20000

means_30 = [np.mean(np.random.normal(size=n)) for i in range(N)]
ax = plt.hist(means_30, bins=50)
plt.show()
```



2.2.10 The sampling distribution of the mean

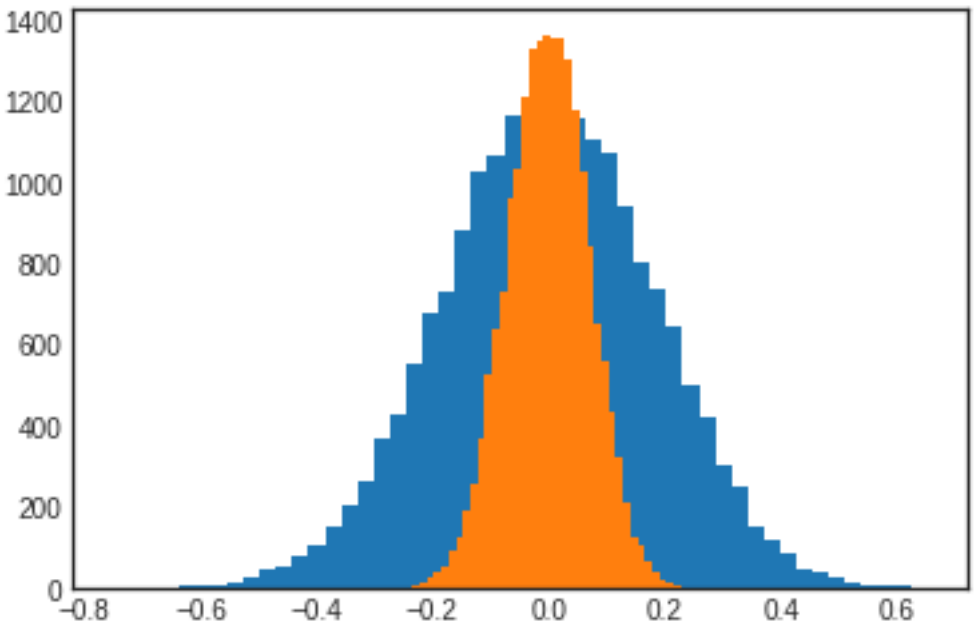
- Now let's do this again for a variable sampled from a *different* distribution: $X \sim U(0,1)$.

```
# Sample size
n = 30
# Number of repeated samples
N = 20000
means_30_uniform = [np.mean(np.random.uniform(size=n)) for i in
    ↪ range(N)]
ax = plt.hist(means_30_uniform, bins=50)
plt.show()
```

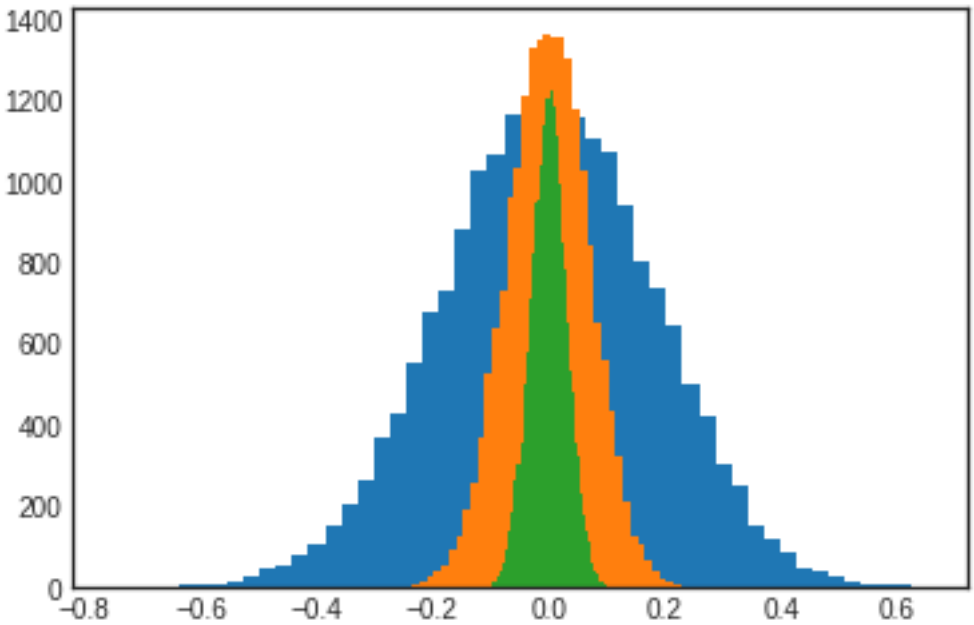


2.2.11 Increasing the sample size

```
# Sample size
n = 200

means_200 = [np.mean(np.random.normal(size=n)) for i in range(N)]
ax1 = plt.hist(means_30, bins=50)
ax2 = plt.hist(means_200, bins=50)
plt.show()
```



```
# Sample size
n = 1000
means_1000 = [np.mean(np.random.normal(size=n)) for i in range(N)]
ax1 = plt.hist(means_30, bins=50)
ax2 = plt.hist(means_200, bins=50)
ax3 = plt.hist(means_1000, bins=50)
plt.show()
```



2.2.12 The sampling distribution of the mean

- In general the sampling distribution of the mean approximates a normal distribution.
- If $X \sim N(\mu, \sigma^2)$ then $\bar{\mathbf{x}_n} \sim N(\mu, \frac{\sigma^2}{n})$.
- The *standard error* of the mean is $\sigma_{\bar{\mathbf{x}}} = \frac{\sigma}{\sqrt{n}}$.
- Therefore sample size must be quadrupled to achieve half the error.

2.3 Errors in simple regression

- Let's perform a similar simulation experiment for OLS.
- We will generate some simulated data from the function $F(x) = 2 + 3x + \epsilon$.

```
n = 20

def noise(size=n):
    return np.random.normal(size=(size, 1), scale=5.0)

def F(x):
    return 2 + 3*x

```

```
def F_noise(x):
    return F(x) + noise(len(x))
```

2.3.1 First sample

```
xdata = np.random.uniform(size=(n, 1), low=-8., high=+8.)
ydata = F_noise(xdata)
regr = skl_lm.LinearRegression()
regr.fit(xdata, ydata)
alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
print(alpha);  print(beta)
```

```
1.5199745767308315
2.7940259728658425
```

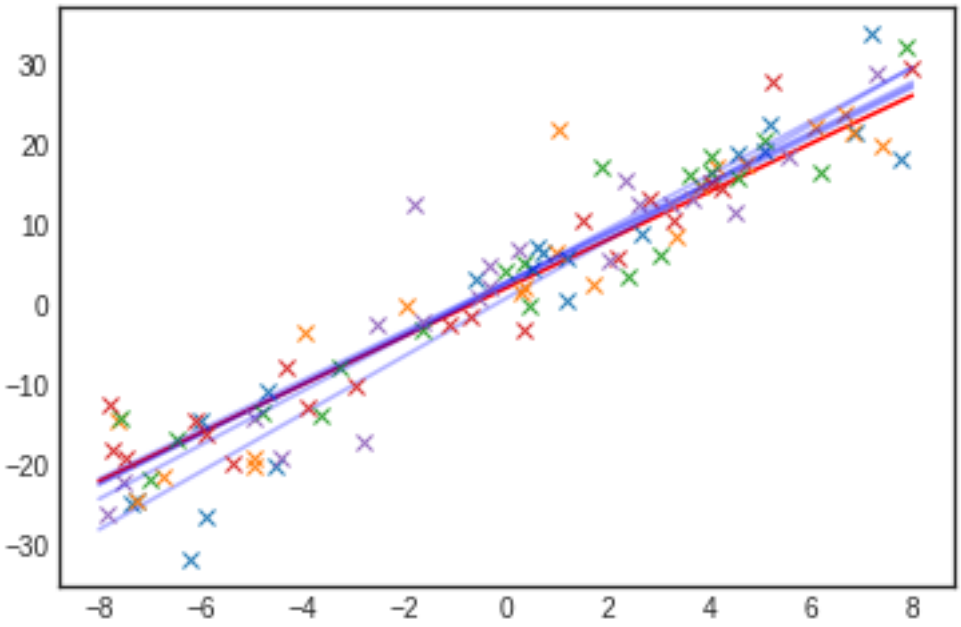
2.3.2 Second sample

```
xdata = np.random.uniform(size=(n, 1), low=-8., high=+8.)
ydata = F_noise(xdata)
regr = skl_lm.LinearRegression()
regr.fit(xdata, ydata)
alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
print(alpha);  print(beta)
```

```
1.9396337527445575
3.0576042625406683
```

2.3.3 Regression plots for different training sets

```
plt.figure()
domain = np.linspace(-8, 8.)
plt.plot(domain, F(domain), color='red')
for i in range(5):
    xdata = np.random.uniform(size=(20, 1), low=-8., high=+8.)
    ydata = F_noise(xdata)
    plt.plot(xdata, ydata, 'x')
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
    plt.plot(domain, alpha + beta * domain, color='blue', alpha
    ↪ =0.3)
plt.show()
```



2.3.4 The distribution of the estimates

```
domain = np.linspace(-8, 8.)
num_experiments = 10000; results = np.zeros((num_experiments, 2))
for i in range(num_experiments):
    xdata = np.random.uniform(size=(n, 1), low=-8., high=+8.)
    ydata = F_noise(xdata)
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
    results[i, 0] = alpha;  results[i, 1] = beta
```

```
pd.DataFrame(results, columns=['$\\alpha$', '$\\beta$']).describe()
```

```
            $\alpha$        $\beta$
count   10000.000000   10000.000000
mean        2.007958       3.000428
std         1.152220       0.250060
min        -2.995561       2.105802
25%         1.240216       2.831471
50%         2.007551       3.002951
75%         2.783609       3.167646
max         6.434260       4.092616
```

2.3.5 Quantifying the *variance of a model*

- We can quantify the goodness-of-fit of our model when we train it on different data sets.
- Consider some out-of-sample data $x_0 = 10$ and $y_0 = f(x_0) + \epsilon$.
- Over the different training sets, we can estimate:

– the variance of our forecasts: $\mathrm{Var}(\hat{f}(x_0))$,
– the expected squared-error $E(y_0 - \hat{f}(x_0))^2$.

2.3.6 Estimating MSE and variance using Monte-Carlo

```
domain = np.linspace(-8, 8.);
out_of_sample_data = np.array([10.])
num_experiments = 50000; results = np.zeros((num_experiments, 2))
for i in range(num_experiments):
    xdata = np.random.uniform(size=(n, 1), low=-8., high=+8.)
    ydata = F_noise(xdata)
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
    prediction = alpha + beta * out_of_sample_data
    squared_error = (F_noise(out_of_sample_data) - prediction) ** 2
    results[i] = [squared_error, prediction]
```

2.3.7 The relationship between error and variance

```
mean_squared_error = np.mean(results[:,0])
mean_squared_error
```

```
32.58961393651053
```

```
prediction_variance = np.var(results[:,1])
prediction_variance
```

```
7.842930817022927
```

```
noise_variance = np.var(noise(size=50000))
noise_variance
```

```
25.024375390926163
```

```
noise_variance + prediction_variance
```

```
32.86730620794909
```

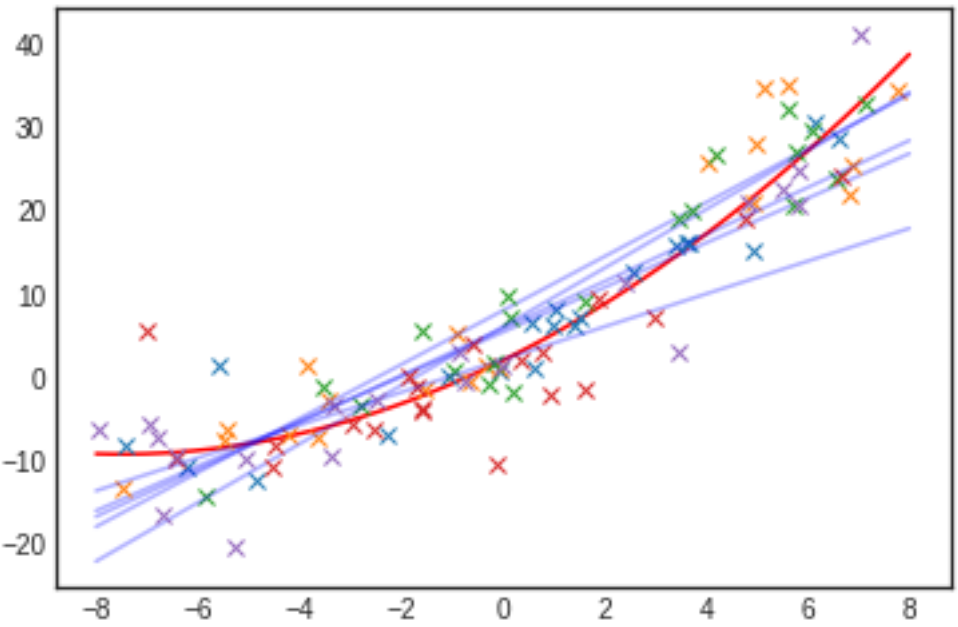
2.4 Fitting a biased model

- In the previous example we fitted a linear model to linear data.
- Let's see what happens if we attempt to fit a linear model to non-linear data.

```

def F(x):
    return 2 + 3*x + 0.2*x**2

def F_noise(x):
    return F(x) + noise(len(x))
```

2.4.1 Regression plots for different training sets

```
plt.figure()
domain = np.linspace(-8, 8.)
plt.plot(domain, F(domain), color='red')
for i in range(5):
    xdata = np.random.uniform(size=(20, 1), low=-8., high=+8.)
    ydata = F_noise(xdata)
    plt.plot(xdata, ydata, 'x')
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
    plt.plot(domain, alpha + beta * domain, color='blue', alpha
    ↪ =0.3)
plt.show()
```



2.4.2 Estimating error and variance

```
domain = np.linspace(-8, 8.)
out_of_sample_data = np.array([10.])
num_experiments = 50000; results = np.zeros((num_experiments, 2))
for i in range(num_experiments):
    xdata = np.random.uniform(size=(n, 1), low=-8., high=+8.)
    ydata = F_noise(xdata)
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]; beta = regr.coef_[0][0]
    prediction = alpha + beta * out_of_sample_data
    squared_error = (F_noise(out_of_sample_data) - prediction) ** 2
    results[i] = [squared_error, prediction]
```

2.4.3 Results

```
mean_squared_error = np.mean(results[:,0])
mean_squared_error
```

```
294.1317532077317
```

```
prediction_variance = np.var(results[:,1])
prediction_variance
```

```
15.36177739248392
```

```
noise_variance = np.var(noise(size=50000))
noise_variance
```

```
24.877729388078404
```

```
noise_variance + prediction_variance
```

```
40.23950678056232
```

2.5 The bias-variance trade-off

- The additional source of error in the previous example is called the bias.
- Bias arises when our model is of the wrong form to be able to fully fit the data.
- In general:

$$E[y_0 - \hat{f}(x)]^2 = \text{Var}[\hat{f}(x)] + \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\epsilon] \tag{2.13}$$

where

$$\text{Bias}[\hat{f}(x)] = E[\hat{f}(x)] - f(x) \tag{2.14}$$

```
bias = np.sqrt(mean_squared_error - (noise_variance +
    ↪ prediction_variance))
bias
```

```
15.933996561665545
```

2.6 Improving the fit the model

- We cannot eliminate the final term in the previous equation; $\text{Var}(\epsilon)$ is *irreducible error*.
- In general, if we reduce bias, we increase variance.
- This is called the bias-variance trade-off.
- We can reduce variance by reducing the number of weights in $\mathbf{w}$.
- In multiple regression this can be done by reducing the number of predictors.
- Correspondingly as we increase the number of predictors we increase variance, but reduce bias.
- This can result in *over-fitting* to the training data.

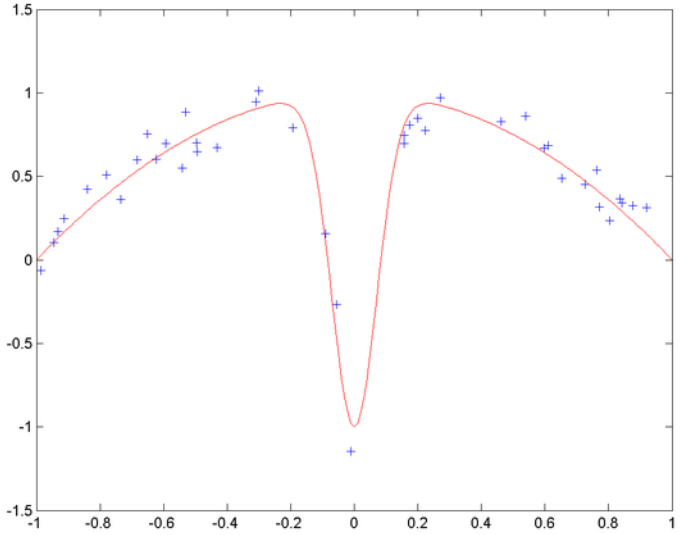


Figure 2.1: testfn

2.6.1 Bias-variance tradeoff example: test function and data

2.6.2 Bias-variance trade-off example: spread=5

2.6.3 Bias-variance trade-off example: spread=1

2.6.4 Bias-variance trade-off example: spread=0.1

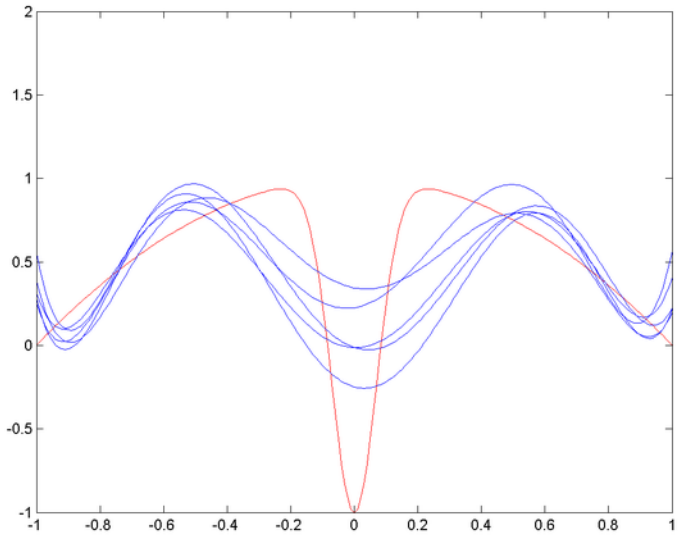


Figure 2.2: spread=5

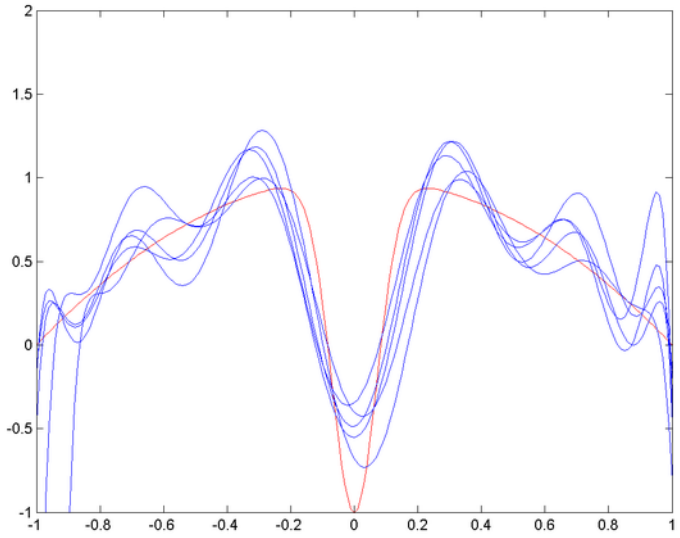


Figure 2.3: spread=1

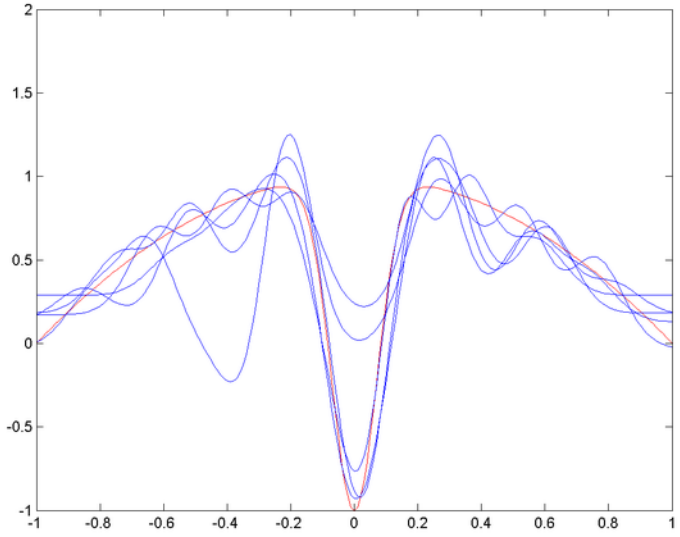


Figure 2.4: spread=0.1

2.7 Bias-variance trade-off as multi-objective optimization

2.7.1 Basis Functions

- We can use non-linear basis functions to decrease the bias of our model.
- We transform a variable X into $b_1(X), b_2(X), \ldots, b_K(X)$
- We then fit the model:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \beta_3 b_3(x_i) + \ldots + \beta_K b_K(x_i) + \epsilon_i \tag{2.15}$$

- The basis functions are known and chosen.
- The above equation is still linear in the basis functions.

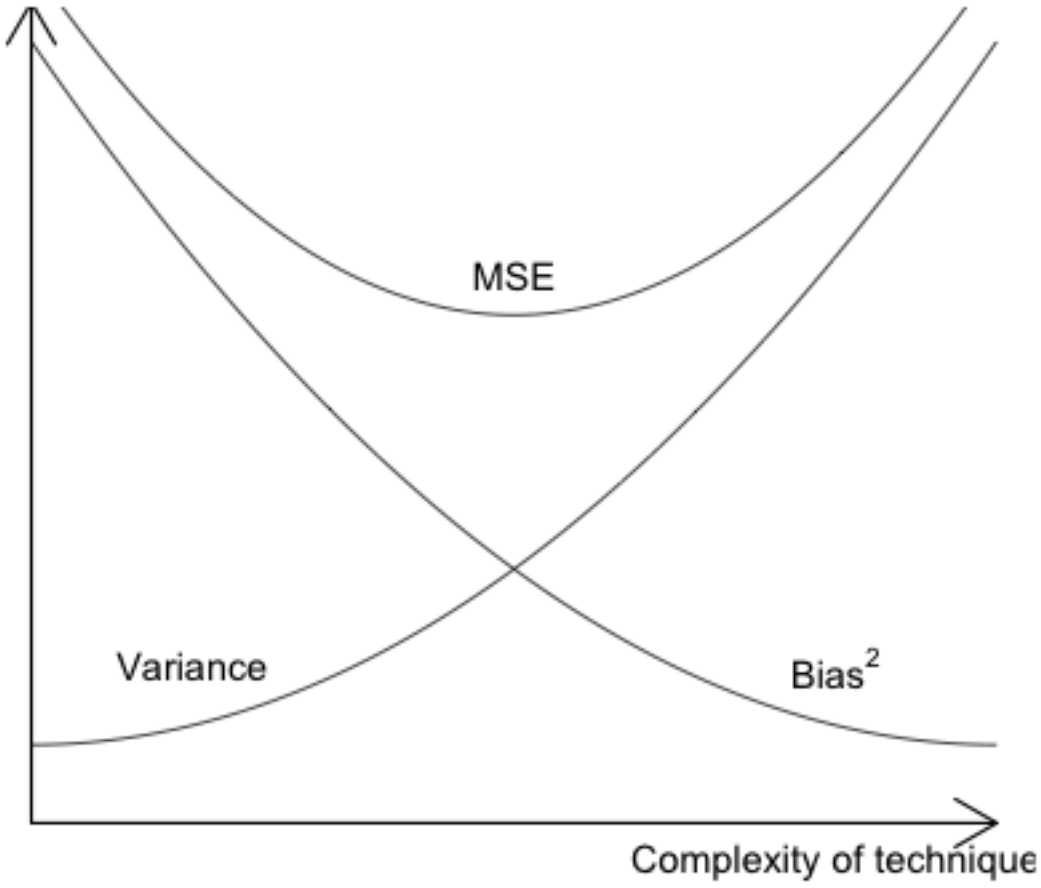


Figure 2.5: tradeoff

2.8 Multiple linear-regression

- Simple regression can be generalized to multiple predictors.
- Consider p distinct predictors with corresponding observations of size n: $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_p$, then:

$$\mathbf{y} = \beta_0 + \beta_1 \mathbf{x}_1 + \beta_2 \mathbf{x}_2 + \ldots + \beta_p \mathbf{x}_p + \epsilon \tag{2.16}$$

- In matrix notation:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{ffl} \tag{2.17}$$

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} 1 & X_{11} & X_{21} & \ldots & X_{p1} \\ 1 & X_{12} & X_{21} & \ldots & X_{p2} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & X_{1n} & X_{2n} & \ldots & X_{pn} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix} \tag{2.18}$$

- The closed-form solution to the loss-minimization problem is:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.19}$$

2.8.1 Applications to Finance: factor models

Fama-French 3-factor model:

$$r_i - r_f = \beta_0 + \beta_1(r_m - r_f) + \beta_2 s_i + \beta_3 v_i + \epsilon_{i,t} \tag{2.20}$$

- r_i is the return for stock i.
- s_i is the size of stock i.
- v_i is the value of stock i.
- r_f is the risk-free rate.
- r_m is the return to the market portfolio.

Fama, Eugene F., and Kenneth R. French. "Common risk factors in the returns on stocks and bonds." Journal of Financial Economics 33:1 (1993). https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.5892&rep=rep1&type=pdf

2.8.2 Polynomial regression

- For polynomial regression we use:

$$b_j(x_i) = x_i^{j-1} \tag{2.21}$$

- For $K = 2$ the simple linear regression model becomes:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 \tag{2.22}$$

- Note that we have *introduced an additional feature* into our model.

2.8.3 Polynomial basis functions in scikit-learn

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])
```

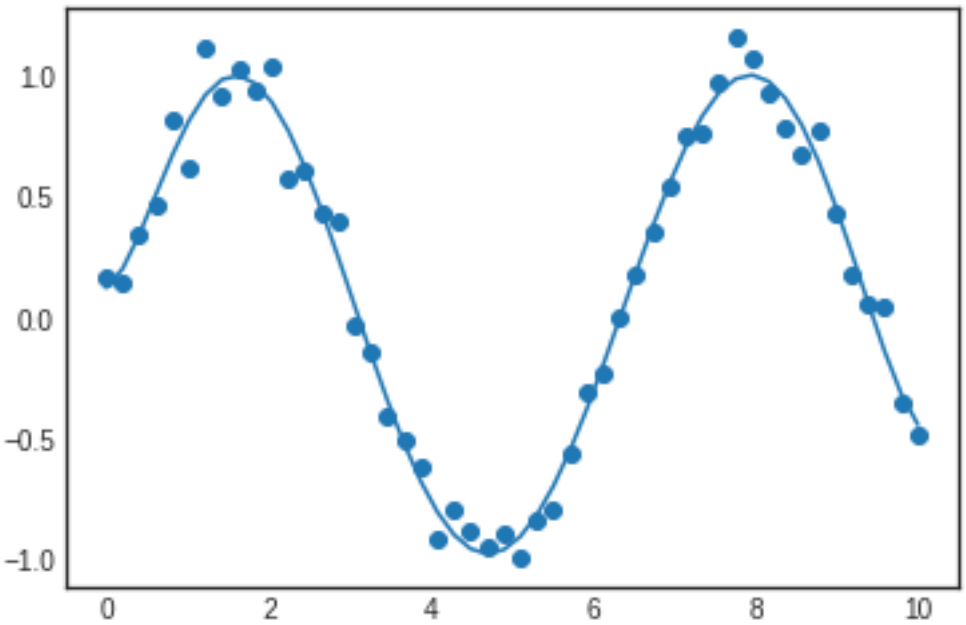
```
array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

2.8.3.1 Plotting the fitted model

- Here we use polynomial basis functions with $K = 7$ to fit $y = \sin(x) + \epsilon$.

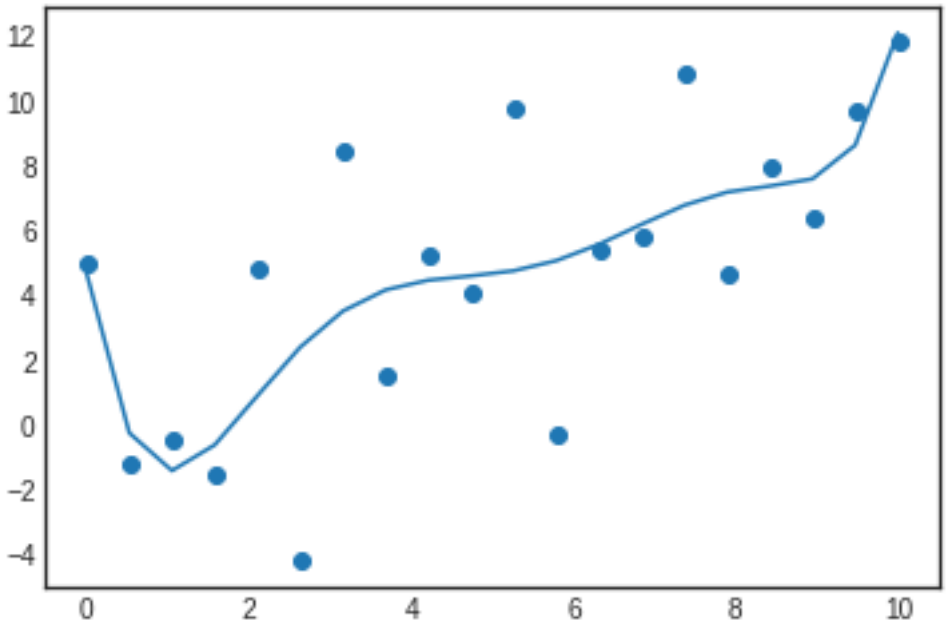
```
rng = np.random.RandomState(1)
x = np.linspace(0, 10.)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x[:, np.newaxis], y)
xfit = x
yfit = poly_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit);
```



2.9 Overfitting

- Let's see what happens if we fit a polynomial model to linear data.
- We see that the model has simply fitted to the noise; it has over-fitted to the training data.

```
rng = np.random.RandomState(1)
x = np.linspace(0, 10., 20)
y = 0.1 + x + rng.randn(20)*3.
poly_model.fit(x[:, np.newaxis], y)
xfit = x
yfit = poly_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit);
```



2.10 Model validation

- It is very important to test for over-fitting, particularly with high-variance models.
- To test for over-fitting we hold back some data from the training.
- We divide data into two subsets:
  1. training data
  2. validation data
- We validate the model by estimating the MSE on the validation data.
- This is also called *out-of-sample validation*.

2.11 Cross-validation

- In cross-validation we create multiple partitions of the data
- Each way of paritition data is called a round.
- We take an average of the error across rounds.

2.11.1 Leave-one-out cross-validation

2.11.2 k-fold cross-validation

2.11.3 Validating the single-index model

- Let's perform model validation with the data we used to train the single-index model:

```
rr = 0.01 # risk-free rate
ydata = stock_simple_returns - rr
xdata = index_simple_returns - rr
```

- We first partition the data into chunks of equal size n:

```
def in_subset(data, i, n=10):
    return data[i*n:(i+1)*n]
```

2.11.4 The first subset

```
in_subset(xdata, 0, n=5)
```

```
            NASDAQ monthly returns
Date
2002-08-31               -0.030497
2002-09-30               -0.126577
2002-10-31                0.178608
2002-11-30                0.117898
2002-12-31               -0.128027
```

2.11.5 The second subset

```
in_subset(xdata, 1, n=5)
```

```
            NASDAQ monthly returns
Date
2003-01-31               -0.011341
2003-02-28                0.017150
2003-03-31               -0.001166
2003-04-30                0.075799
2003-05-31                0.073024
```

2.11.6 The third subset

```
in_subset(xdata, 2, n=5)
```

```
            NASDAQ monthly returns
Date
2003-06-30               -0.006828
2003-07-31                0.052620
2003-08-31                0.040323
2003-09-30               -0.037960
2003-10-31                0.076439
```

2.11.7 The remaining data

```
def out_subset(data, i, size=10):
    return pd.concat([data[:i*size], data[(i+1)*size:]])
```

2.11.8 Model fitting

- Here we fit the model to given data, returning a function that can later be applied to make a prediction.

```
def fit_model(xdata, ydata):
    regr = skl_lm.LinearRegression()
    regr.fit(xdata, ydata)
    alpha = regr.intercept_[0]
    beta = regr.coef_[0][0]
    return lambda x: alpha + beta*x
```

2.11.9 Leave-one-out cross-validation in Python

```
n = 10; N = int(np.floor(len(xdata) / n)); mse = np.zeros(N)
for i in range(N):
    test_data_x = in_subset(xdata, i)
    test_data_y = in_subset(ydata, i)
    training_data_x = out_subset(xdata, i)
    training_data_y = out_subset(ydata, i)
    model = fit_model(training_data_x, training_data_y)
    predicted_ydata = model(test_data_x)
    mse[i] = np.mean((test_data_y.values - predicted_ydata.values)
    ↪ **2)
mse
```

```
array([0.00269961, 0.00310609, 0.00176697, 0.00256889, 0.00236646,
       0.00100183, 0.00639265, 0.00375142, 0.00325358, 0.00297072,
       0.00168612, 0.00098084, 0.00265149, 0.00387894, 0.0016184 ,
       0.00523279, 0.00165305, 0.00054646, 0.00082052, 0.00106942])
```

```
np.mean(mse)
```

```
0.002500812960734237
```

2.11.10 Potential problems with linear-regression

2.11.10.1 Non-linearity of the response-predictor relationships.

- If the actual relationship between response and predictor is non-linear then linear regression with give a very biased result.

2.11.10.2 Correlation or non-constant variance in error terms.

- We assume that the ϵ variate is i.i.d.
- If there are correlations in ϵ, linear regression can give misleading results.
- This often occurs in time-series data.
- Similarly if the variance of the residuals is not constant.
- As we saw with the single-index model fit, we can examine the plot of the residuals for patterns.

2.11.10.3 Outliers and high-leverage points

- Extreme values of response variables are called outliers.
- They can be analyzed using box-plots.
- Outliers can cause over-fitting.
- Extreme values of predictor variables are called high-leverage points.

2.11.10.4 Collinearity.

- We assume there are no linear relationships between predictors.
- If there are, then we are using redundant information, which can make the model hard to interpret.
- Collinearity can be detected by looking at scatter matrices.
- In general, combinations of variables can exhibit a linear relationship, which is called *multi-collinearity*.
- Multi-collinearity can be detected by computing the Variable Inflation Factor (VIF).
- If there are redundant variables, then typically we want to omit them from the regression.
- Finding the correct set of predictor variables is called *variable selection*.

2.12 Variable-selection and regularization

- We can address some of these issues using penalized regression techniques.
- Penalized regression can be used to reduce the variance of a model through regularization.
- It can also be used to eliminate redundant variables.
- This is particularly important for "big-data" problems with large numbers of predictor variables p;
  - for an example in Finance, see the WRDS database.
- Provided the number of observations is significantly greater that the number of preditors $n > p$, ordinary least squares estimation can give low-variance estimates.
  - If $n < p$ there is no single optimal solution to the OLS optimization problem.

2.12.1 Shrinkage using Ridge regression

- Recall that in linear regression our loss function L is simply the residual sum of squares (RSS):

$$\text{RSS} = \sum_{i=1}^{n} r_i^2 \tag{2.23}$$

where

$$\begin{aligned} r_i &= y_i - F_{\mathbf{w}}(x_i) && (2.24)\\ &= y_i - w_0 - \sum_{j=1}^{p} w_j x_{ij} && (2.25) \end{aligned}$$

- In *ridge regression* we add an additional *shrinkage penalty* term:

$$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \text{RSS} + \lambda \sum_{j=1}^{p} w_j^2 \tag{2.26}$$

where λ is a tunable *hyper-parameter*.

The optimal weights are:

$$w^* = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.27}$$

2.12.2 Ridge regression in Python

- In scikit-learn you can perform ridge regression simply by using the `Ridge()` function instead of the `LinearRegression()` function:

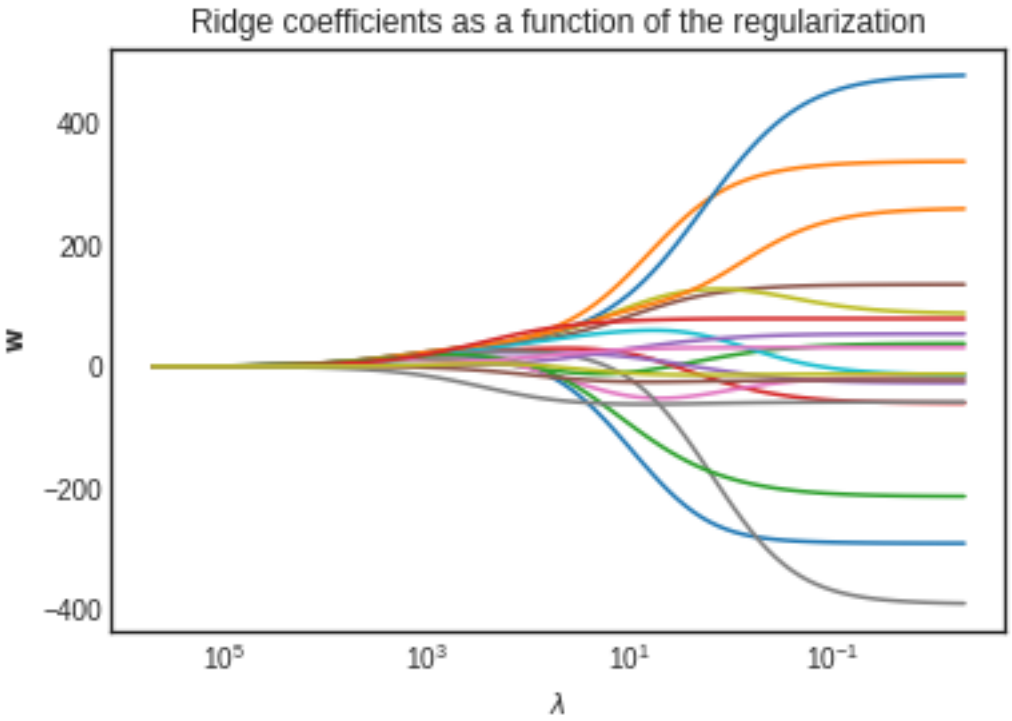
```
1 from sklearn.linear_model import Ridge
```

2.12.3 The norm

- The norm of a vector gives its Euclidian distance from the origin:

$$||\mathbf{w}||_2 = \sqrt{\sum_{j=1}^{p} {w_j}^2} \tag{2.28}$$

- As λ increases $||\hat{\mathbf{w}}_\lambda||_2$ decreases.



2.12.4 Scaling and standardization

- The ridge penalty term is not scale equivariant.
- Therefore when using penalised regression we must use standardized predictors:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i j - \bar{x}_j)^2}} \tag{2.29}$$

- In scikit-learn you can use the `scale()` function:

```
from sklearn.preprocessing import scale
```

2.13 Shrinkage, bias and variance

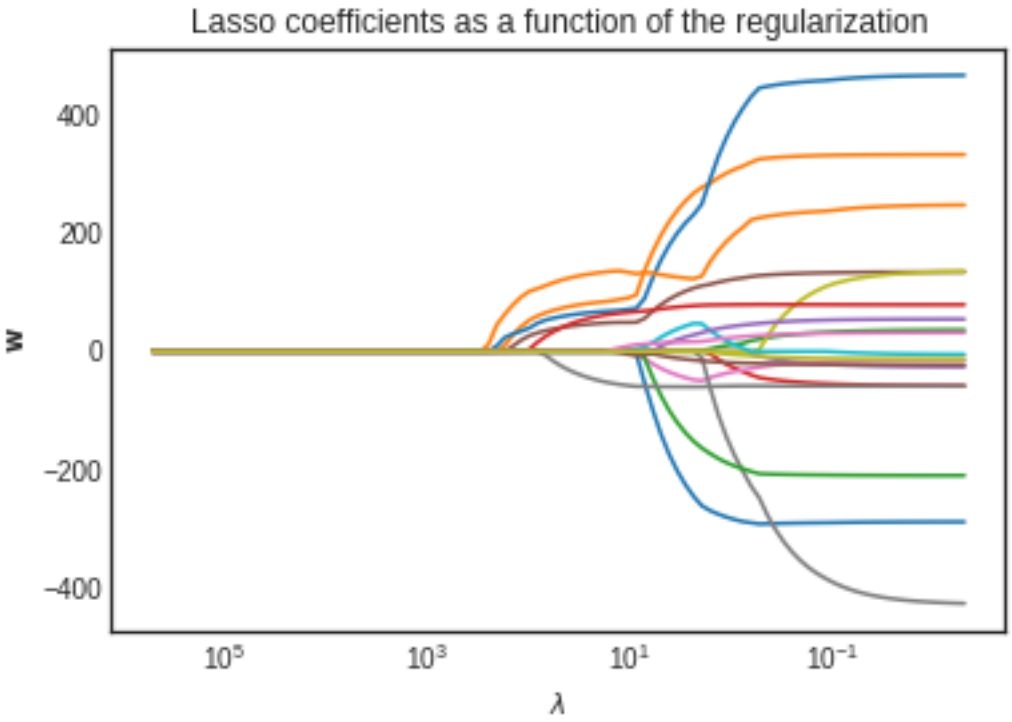
- Shrinkage reduces variance.
- Therefore it increases bias.
- Therefore there exist optimal values of λ which minimize out-of-sample MSE.
- This hyper-parameter can be tuned by using cross-validation methods to find the optimal value.

2.14 Lasso regression

- The lasso regression uses an penalty term:

$$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \text{RSS} + \lambda \sum_{j=1}^{p} |w_j| \tag{2.30}$$

- In contrast to ridge regression, the lasso does not shrink all coefficients to zero for the same λ.
- This means that it can be used for *variable selection*.



2.14.1 Penalized regression as constrained optimization

- An alternative formulation of penalized regression is to view it as constrained optimization.
- We minimize the standard OLS loss function:

$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \text{RSS}$

- For ridge regression we use the constraint:

$$\sum_{j=1}^{p} w_j^2 \leq s \tag{2.31}$$

- For lasso regression we use the constraint:

$$\sum_{j=1}^{p} |w_j| \leq s \tag{2.32}$$

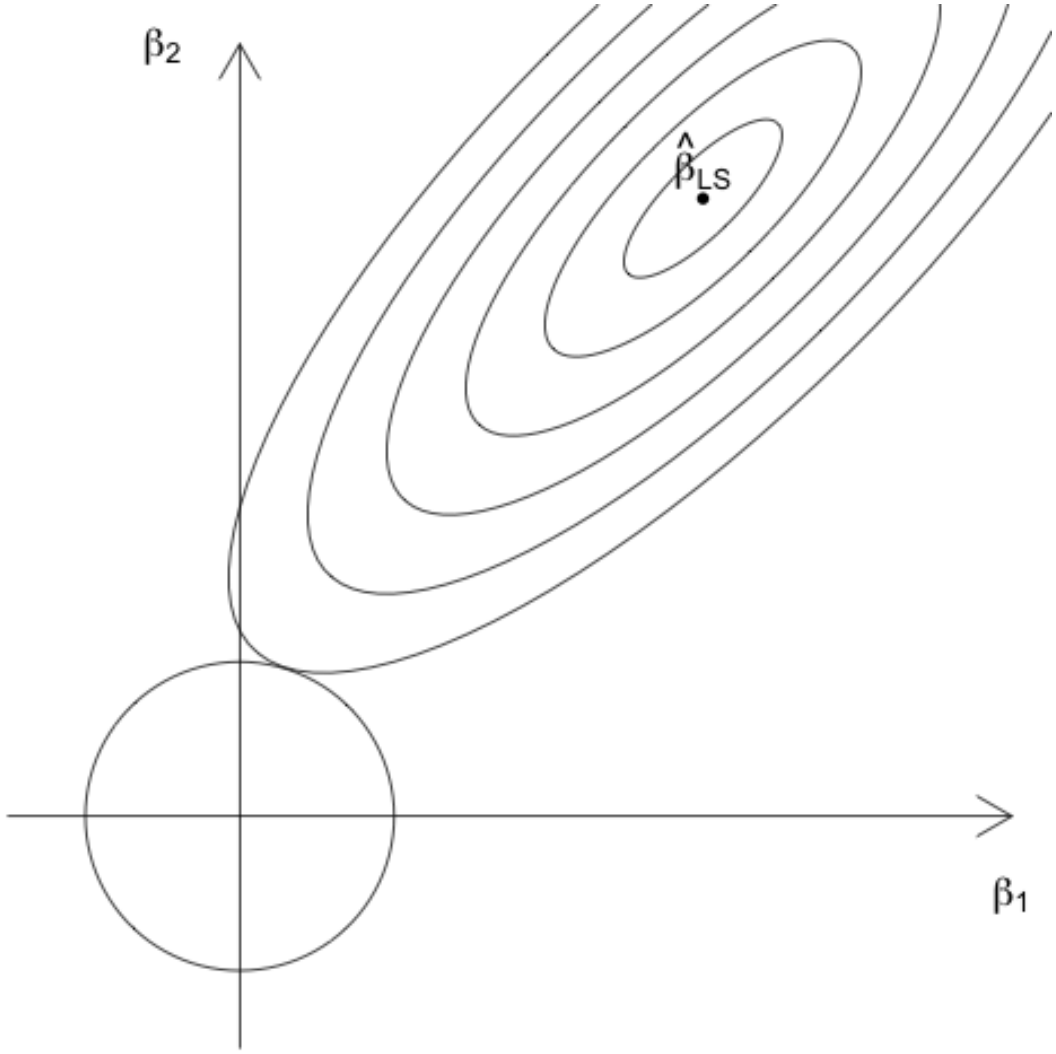


Figure 2.6: ridge

- For every λ there is a corresponding constant budget s for which solutions to the constrained and unconstrained problem are identical.

2.14.2 Error contours and contraints for the ridge penalty

2.14.3 Error contours and contraints for the lasso penalty

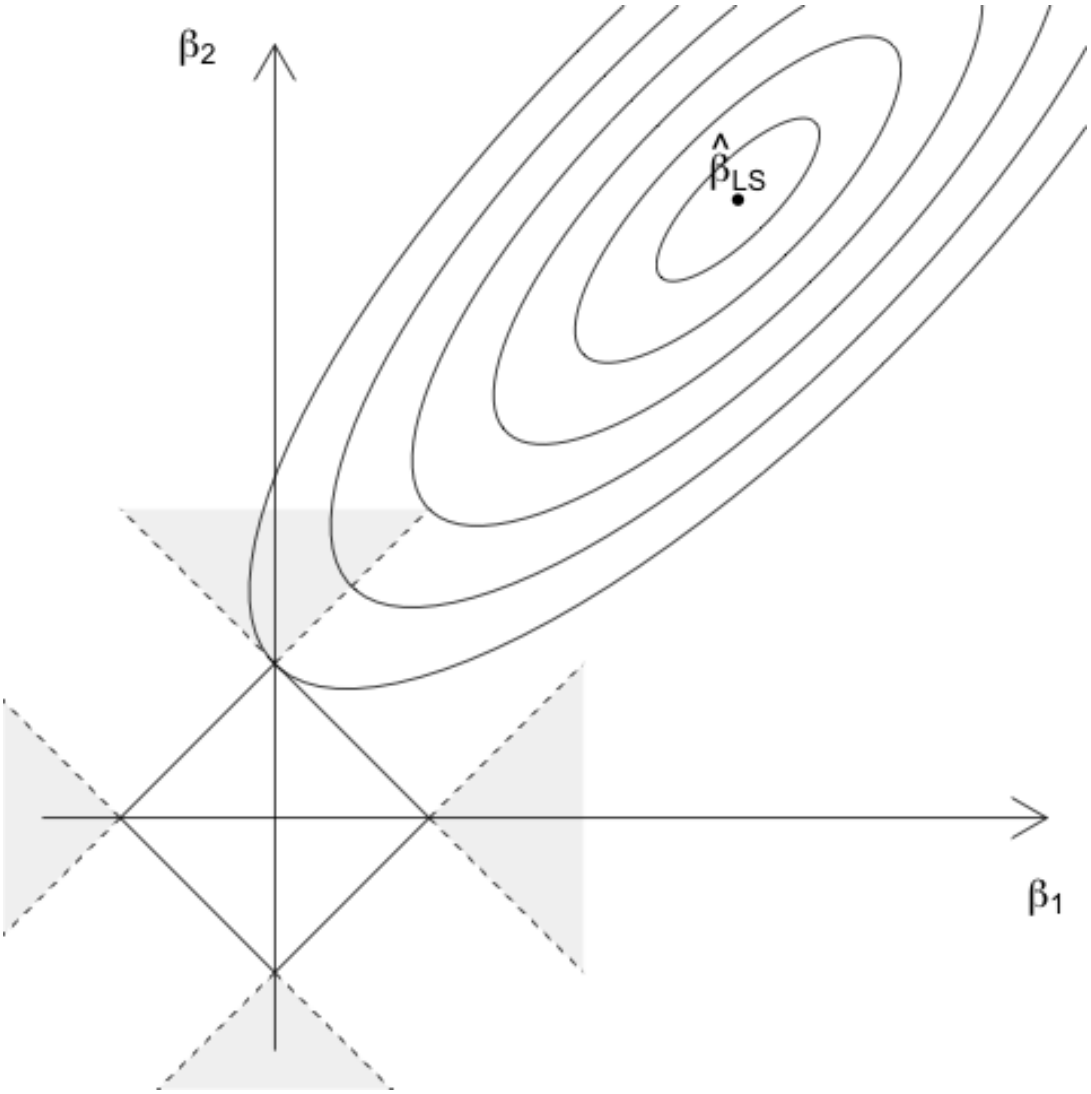


Figure 2.7: lasso

2.15 Lasso and Ridge comparision

- Lasso is able to perform variable-selection when the data-set contains many redundant features that do not influence the response.
- Ridge works when most predictors impact the response.
- In practice we don't know the true model, so how should we select the penalization?

2.16 Elastic-net

- We can combine both penalties into a single term by using a linear combination.
- This transforms the problem into a multi-objective optimisation problem (MOO).
- The parameter α specifies the weighting towards a given objective.

$$L(\mathbf{x}, \mathbf{y}, \mathbf{w}) = \text{RSS} + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{p} |w_j| + \alpha \sum_{j=1}^{p} w_j^2 \right) \tag{2.33}$$

2.16.1 Penalized regression in Finance

- Bianchi and McAlinn (2020) attempt to predict monthly excess returns based on 70 predictors.
- The predictors are financial ratios taken from the WRDS database.

Bianchi, D., & McAlinn, K. (2020). Divide and Conquer: Financial Ratios and Industry Returns Predictability (No. 3136368; SSRN). https://doi.org/10.2139/ssrn.3136368

2.16.2 Predictor categories

- The predictors are grouped into seven categories:
  1. Capitalization
  2. Efficiency
  3. Financial Soundness/Solvency
  4. Liquidity
  5. Profitability
  6. Valuation
  7. Other

2.16.2.1 Example predictors

- Gross profitability as a fraction of total assets,
- Net Income as a fraction of average of Common Equity based on most recent two periods,

...

(Bianchi and McAlinn 2020, p. 70)

Method	Durables	NonDurables	Manuf	Energy	HiTech	Health	Other	Shops	Telecom	Utils		

Method	Durables	NonDurables	Manuf	Energy	HiTech	Health	Other	Shops	Telecom	Utils		
OLS	-1.660	-0.527	-1.072	-0.874	-0.856	-0.353	-0.898	-0.163	-0.411	-0.227	-0.729	-1.409
Lasso	-0.478	-0.425	-0.291	-0.734	-0.182	-0.081	-0.135	-0.753	-0.057	-0.248	-0.001	-0.567
Ridge	-0.151	-0.256	-0.138	-1.027	-0.138	-0.328	-0.187	-0.097	-0.188	-0.202	-0.718	-0.455
Enet	-0.013	-0.553	-0.346	-0.048	-0.434	-0.499	-0.086	-0.582	-1.250	-0.092	-0.125	-0.490
EW	-0.142	-0.020	-0.202	-0.919	-0.606	-0.058	-0.050	-0.681	-0.355	-0.323	-0.127	-0.441
BMA	-0.409	-0.158	-0.284	-0.211	-0.467	-0.337	-0.042	-0.018	-0.335	-0.126	-0.146	-0.670
Factor	-0.017	-0.383	-0.277	-0.139	-0.272	-0.113	-0.041	-0.292	-0.391	-0.571	-0.144	-0.001
Macro	-0.050	-0.262	-0.422	-0.009	-0.012	-0.066	-0.039	-0.030	-0.352	-0.283	-0.347	-0.001
DRS	-0.972	-1.458	-0.544	-0.181	-0.281	-0.755	-0.031	-0.581	-0.712	-0.807	-0.622	-1.564

2.16.2.2 Annualized certainty-equivalent by method

(Bianchi and McAlinn 2020, p. 58)

2.17 Kernel methods

- Recall that we can sometimes we can fit to non-linear data using basis functions.
- We use a set of functions that *increases the number of the features*, e.g.:

$$\{1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, x_1^2, x_2^2, x_3^2\} \tag{2.34}$$

- Our set of basis functions is

$$\begin{aligned} \phi_1(\mathbf{x}) &= 1 \\ \phi_2(\mathbf{x}) &= x_1 \\ &\dots \\ \phi_6(\mathbf{x}) &= x_1x_3 \\ &\dots \end{aligned}$$

2.17.1 Model

$$F_{\mathbf{w}}(\mathbf{x}) = \sum_{m=1}^{M_\phi} w_m \phi_m(\mathbf{x}) \tag{2.35}$$

2.17.2 Features

$$\mathbf{\Phi} = \begin{pmatrix} \phi_1(\mathbf{x}_1) & \cdots & \phi_{M_\phi}(\mathbf{x}_1) \\ \vdots & \vdots & \vdots \\ \phi_1(\mathbf{x}_n) & \cdots & \phi_{M_\phi}(\mathbf{x}_n) \end{pmatrix} \tag{2.36}$$

2.18 Basis vector

$$\phi(\mathbf{x}) = (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_{M_\phi}) \tag{2.37}$$

2.19 Vectorized model

$$F_{\mathbf{w}}(\mathbf{x}) = \mathbf{\Phi}\mathbf{w} \tag{2.38}$$

2.20 Penalised least-squares with basis functions

- Penalised loss function:

$$\begin{aligned} L(\mathbf{w}) &= \sum_{i=1}^{n} \left(y_i - \sum_{m=1}^{M_\phi} w_m \phi_m(\mathbf{x}_i) \right)^2 + \frac{\lambda}{2} \sum_{m=1}^{M_\phi} w_m^2 \\ &= (\mathbf{y} - \mathbf{\Phi}\mathbf{w})^T (\mathbf{y} - \mathbf{\Phi}\mathbf{w}) + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w} \end{aligned}$$

- Solution

$$\mathbf{w}^* = (\mathbf{\Phi}^T \mathbf{\Phi} + \lambda I)^{-1} \mathbf{\Phi}^T \mathbf{y} \tag{2.39}$$

2.21 Kernelization

$$\begin{aligned} (\mathbf{\Phi^T \Phi} + \lambda I)\mathbf{w}^* &= \mathbf{\Phi}^T \mathbf{y} && (2.40) \\ \lambda \mathbf{w}^* &= \mathbf{\Phi}^T \mathbf{y} - \mathbf{\Phi}^T \mathbf{\Phi} \mathbf{w}^* && (2.41) \\ &= \mathbf{\Phi}^T (y - \mathbf{\Phi}\mathbf{w}^*) && (2.42) \\ \mathbf{w}^* &= \lambda^{-1} \mathbf{\Phi}^T (y - \mathbf{\Phi}\mathbf{w}^*) && (2.43) \\ &= \mathbf{\Phi}^T \mathbf{f} \mathbf{f} && (2.44) \\ & && (2.45) \end{aligned}$$

where

$$\alpha_i = \lambda^{-1} \left[y_i - \mathbf{w}^T \phi(\mathbf{x_i}) \right] \tag{2.46}$$

$$\begin{aligned} \lambda \mathbf{f} \mathbf{f} &= \mathbf{y} - \mathbf{\Phi}\mathbf{w}^* && (2.47) \\ &= \mathbf{y} - \mathbf{\Phi}\mathbf{\Phi}^T \alpha && (2.48) \\ \mathbf{\Phi}\mathbf{\Phi}^T \alpha + \lambda \alpha &= \mathbf{y} && (2.49) \\ \alpha &= (\mathbf{\Phi}\mathbf{\Phi}^T + \lambda I)^{-1} \mathbf{y} && (2.50) \\ &= (K + \lambda I)^{-1} \mathbf{y} && (2.51) \\ & && (2.52) \end{aligned}$$

2.21.1 Kernel matrix

$$\mathbf{\Phi\Phi}^T = \mathbf{K} = \begin{pmatrix} \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_n) \\ \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_n) \\ \vdots & \vdots & \vdots & \vdots \\ \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_n) & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_n) \end{pmatrix} \tag{2.53}$$

2.21.2 Kernelization

- The non-kernelized optimal weight vector is:

$$\mathbf{w}^* = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda I)^{-1}\mathbf{\Phi}^T\mathbf{y} \tag{2.54}$$

- In contrast the kernelized version is:

$$\mathbf{w}^* = \mathbf{\Phi}^T\mathbf{f}\mathbf{f} = \mathbf{\Phi}^T(\mathbf{K} + \lambda I)^{-1}\mathbf{y} \tag{2.55}$$

2.21.3 Kernalized predictions

$$\begin{aligned} \mathbf{F} &= \mathbf{\Phi}\mathbf{w}^* && (2.56) \\ \mathbf{F}^T &= \mathbf{w}^{*T}\mathbf{\Phi}^T && (2.57) \\ &= \mathbf{y}^T(\mathbf{\Phi\Phi}^T + \lambda I)^{-1}\mathbf{\Phi}^\mathbf{T}\mathbf{\Phi} && (2.58) \\ &= \mathbf{y}^T(\mathbf{K} + \lambda I)^{-1}\mathbf{K} && (2.59) \end{aligned}$$

- We do not require $\mathbf{\Phi}$.
- Predictions can be made using only the $n \times n$ kernel matrix K.

2.21.4 Kernel in terms of features

$$\mathbf{K} = \begin{pmatrix} \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_n) \\ \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_n) \\ \vdots & \vdots & \vdots & \vdots \\ \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_n) & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_n) \end{pmatrix} \tag{2.60}$$

2.21.5 Using a kernel function

$$\mathbf{K} = \begin{pmatrix} \mathrm{k}(\mathbf{x}_1, \mathbf{x}_1) & \mathrm{k}(\mathbf{x}_1, \mathbf{x}_2) & \cdots & \mathrm{k}(\mathbf{x}_1, \mathbf{x}_n) \\ \mathrm{k}(\mathbf{x}_2, \mathbf{x}_1) & \mathrm{k}(\mathbf{x}_2, \mathbf{x}_2) & \cdots & \mathrm{k}(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \vdots & \vdots \\ \mathrm{k}(\mathbf{x}_n, \mathbf{x}_1) & \mathrm{k}(\mathbf{x}_n, \mathbf{x}_2) & \cdots & \mathrm{k}(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix} \tag{2.61}$$

- where k is a symmetric function such that K is positive semi-definite $\mathbf{x}^T\mathbf{K}\mathbf{x} \geq 0\ \forall \mathbf{x}$.

2.21.6 The kernel trick

- The linear kernel is simply the inner product:

$\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \cdot \mathbf{y}_j$

- In this case our basis mapping is simply the identity function.
- However, there are generalizations of the inner product called kernel functions.
- Every kernel function defines an implicit set of basis functions.

2.21.6.1 Example

- Consider an original feature space with $p = 2$ dimensions, and with basis functions:

$$\begin{aligned} \phi_1(\mathbf{x}) &= x_1^2 \\ \phi_2(\mathbf{x}) &= \sqrt{2}x_1x_2 \\ \phi_3(\mathbf{x}) &= x_2^2 \end{aligned}$$

$$\phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

2.21.6.2 The polynomial kernel

$$\begin{aligned} \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j) &= x_{i1}^2x_{j1}^2 + 2x_{i1}^2x_{j1}^2 + x_{i2}^2x_{j2}^2 \\ &= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 \\ &= (\mathbf{x}_i^T\mathbf{x}_j)^2 \\ &= \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) \end{aligned}$$

- Our kernel matrix is defined only in terms of k.
- Using this kernel function we can work in the original 2-dimensional feature space instead of the 3-dimensional mapped feature space.
- This is an example of a Polynomial kernel:

$$\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + r)^d \tag{2.62}$$

- For other kernels see Murphy (2012), ch. 14.

2.21.6.3 Out of sample forecasts

- For an out of sample observation $\mathbf{x}_j$:

$$\begin{aligned} \hat{F} &= \mathbf{w}^{*^T}\phi(\mathbf{x}_j) && (2.63) \\ &= \sum_i^n \alpha_i\phi^T(\mathbf{x}_i)\phi(\mathbf{x}_j) && (2.64) \\ &= \sum_i^n \alpha_i k(\mathbf{x}_i, \mathbf{x}_j) && (2.65) \end{aligned}$$

3 Classification methods

- In a classification problem we attempt to map quantitative data onto *categorical* variables.
- Categorical variables are also called qualitative variables.
- They take one of a discrete set of categories.
- Examples:

  1. Given data on a banking transaction, is it a) *fraudulent*, or b) *legitimate*?
  2. Given a credit-history, is the applicant a) *high-risk*, or b) *low-risk*?

  3. Given the state of the order-book, will the mid-price move a) *up*, or b) *down*.

3.1 Classification methods in Finance

- Business failure prediction
- Credit risk assessment
- Corporate mergers and acquisitions
- Stock rating models
- Bond rating models
- Price movement prediction

3.2 Example features

- x_1 Total assets
- x_2 Total debt
- x_3 Long-term debt / total invested capital
- x_4 Debt ratio

...

3.3 Example categories

Ratings: AAA, AA, A, BBB, BB, B, ...

3.4 Validating classifiers

- Because data are qualitative as opposed to quantitative, we cannot use the same metrics, such as R^2 or out-of-sample RSS that we used for regression methods.
- We explicitly count false positives (FP), false negatives (FN), true positives (TP) and true negatives (TN).
- For example, in a credit-risk model, a false positive would occur if we mis-classify an individual as high-risk, when in fact they were low-risk.

3.4.0.1 Accuracy

$$accuracy = \frac{correct}{correct + incorrect} \tag{3.1}$$

3.4.0.2 Precision

$$precision = \frac{TP}{TP + FP} \tag{3.2}$$

3.4.0.3 Recall (aka sensitivity)

$$recall = \frac{TP}{TP + FN} \tag{3.3}$$

3.5 Support Vector Machines

- Support vector machines (SVMs) can be used to classify data into one of two classes (labels).
- It is a supervised learning algorithm, so we start with known labels in our training data: for every training input $\mathbf{x}_i \in X$ we have a known label $y_i \in +1, -1$.
- The task of the learning algorithm is to generalize to unseen cases not in our training set X.
- The output from the classification is a definitive and non-probabilistic.
- As in regression, we make use of a linear algebra to model the problem.
- Non-linear problems can be solved using kernel functions.

3.5.1 scikit-learn

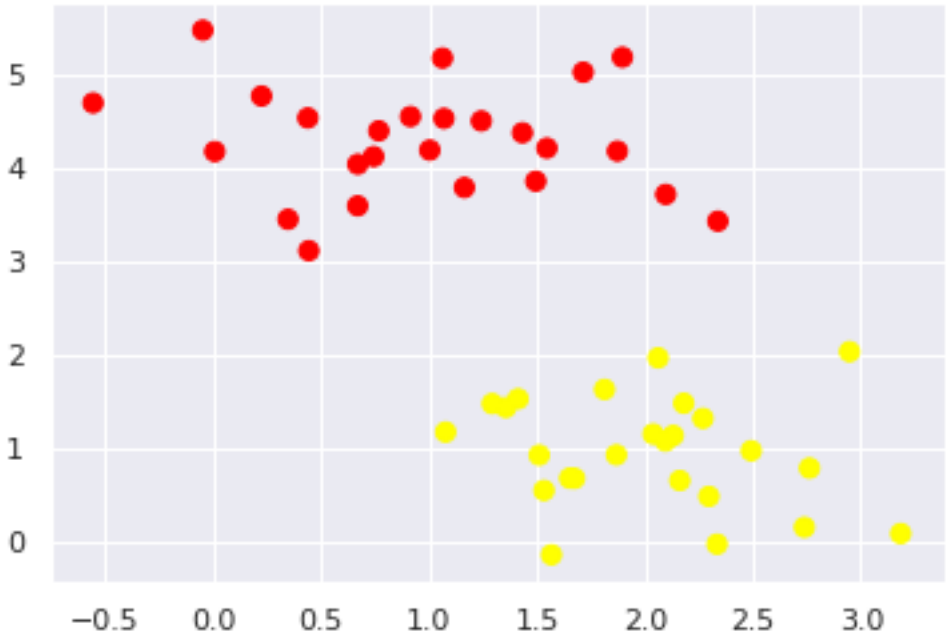
- We will first illustrate the problem with some plots and solutions and then move to the math.
- We begin with the standard imports:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import sklearn

# use seaborn plotting defaults
import seaborn as sns; sns.set()
```

3.5.2 Example classification problem

- We first consider some training data (in this case we use simulated data).
- Each input $\mathbf{x}_i$ has two features $p = 2$, so we can plot the inputs in a two-dimensional plane.
- We color-code the associated class red for $y_i = 1$ and yellow for $y_i = -1$.

```
from sklearn.datasets import samples_generator
X, y = samples_generator.make_blobs(n_samples=50, centers=2,
                    random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```



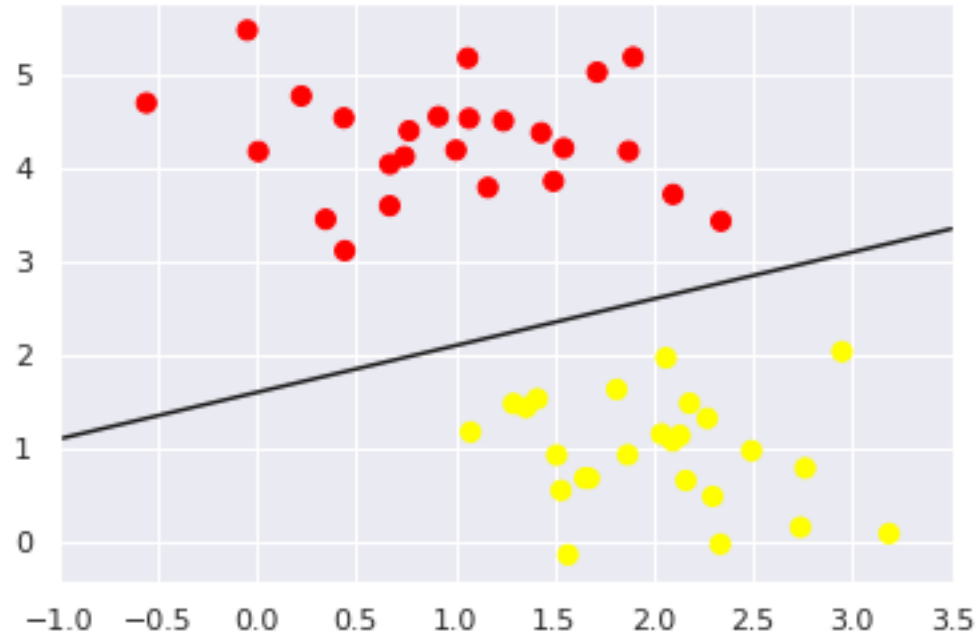
3.5.3 Linear separability

- We attempt to find a hyper-plane that cleanly separates the two classes.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0 \tag{3.4}$$

- The hyper-plane has dimensions one less than the original feature space.
- For two-dimensional data the hyper-plane is therefore simply a line:

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

plt.plot(xfit, 0.5 * xfit + 1.6, '-k')

plt.xlim(-1, 3.5);
```



3.5.4 Classification using hyperplanes

- We can classify the data according to which side of the hyper-plan a data-point lies on.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p > 0 \tag{3.5}$$
$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p < 0 \tag{3.6}$$
$$\tag{3.7}$$

- That is, for a test observation $\mathbf{x}_j$ with p features we can make predictions by examining the sign of:

$$f(\mathbf{x}) = \beta_0 + \beta_1 x_{j1} + \beta_2 x_{j2} + \ldots + \beta_p x_{jp} \tag{3.8}$$

3.5.5 Model selection

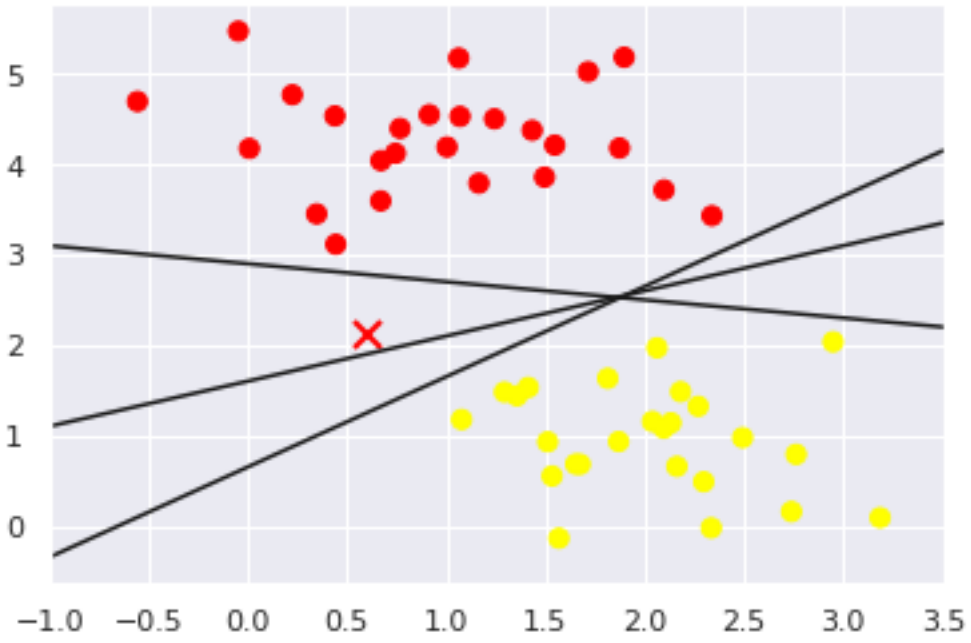
- The problem of finding a separating hyper-plane does not have a unique solution.
- Going back to our previous example, there are three *very* different separators which perfectly discriminate between these samples.

3.5.5.1 Model selection illustrated

- The point marked with a red cross represents out-of-sample data whose true label is $+1$.
- However, depending on which model we choose, this observations will be assigned a different label.

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2,
    ↪ markersize=10)
```

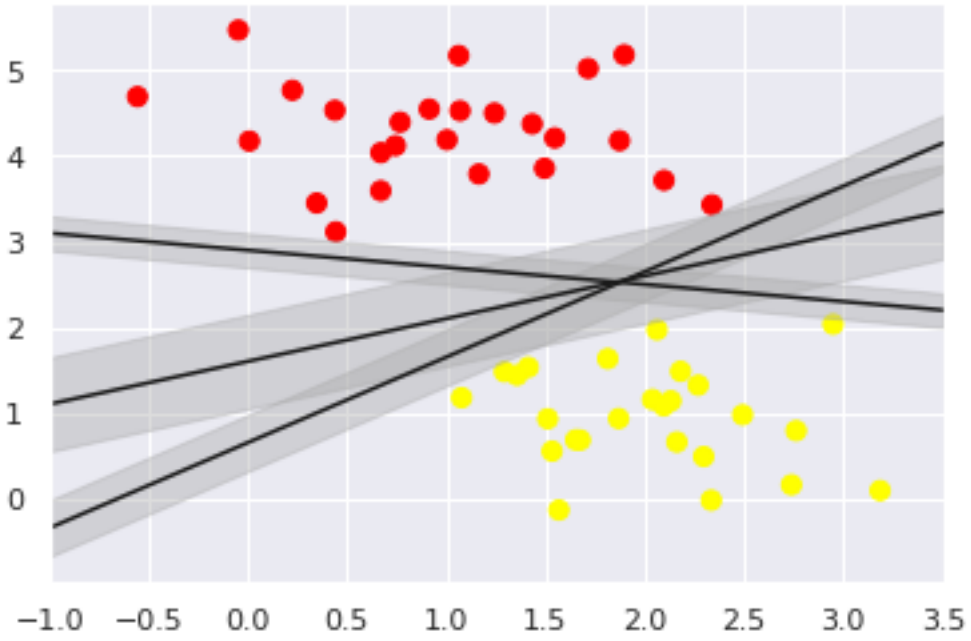
```
4
5  for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
6      plt.plot(xfit, m * xfit + b, '-k')
7
8  plt.xlim(-1, 3.5);
```



3.5.6 The Margin

- Given a particular hyper-plane, we calculate a margin M which is the distance from the hyper-plane to the nearest training input.

```
1  xfit = np.linspace(-1, 3.5)
2  plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
3
4  for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)
       ↪ ]:
5      yfit = m * xfit + b
6      plt.plot(xfit, yfit, '-k')
7      plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
8                       color='#AAAAAA', alpha=0.4)
9
10 plt.xlim(-1, 3.5);
```



3.5.7 Maximising the margin

- We can use M to translate the classification into a loss-minimization problem with a unique solution.
- First we will vectorize our equation for the hyperplane:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{3.9}$$

- where the vector $\mathbf{w}$ is perpendicular to the plane.
- For an observation $\mathbf{x}_i$ with label y_i, the distance to the hyperplane is:

$$D_i = \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{||\mathbf{w}||_2} \tag{3.10}$$

- The margin is the distance to the closest observation:

$$M = \min_i D_i \tag{3.11}$$

- The optimal model is the hyper-plane that gives us the best separation, i.e. the maximum margin:

$$\mathbf{w}^* = \max_{\mathbf{w},b} M \tag{3.12}$$

3.5.8 Predictions

- We can make predictions by examining the sign of $f(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b$.
- To test against training data $\mathbf{y}$ we can examine the sign of the product $y_i f(\mathbf{x}_i)$.
- Therefore the constraint that the classification is correct can be expressed:

$$y_i[\mathbf{w}^T \cdot \mathbf{x} + b] \geq 0 \tag{3.13}$$

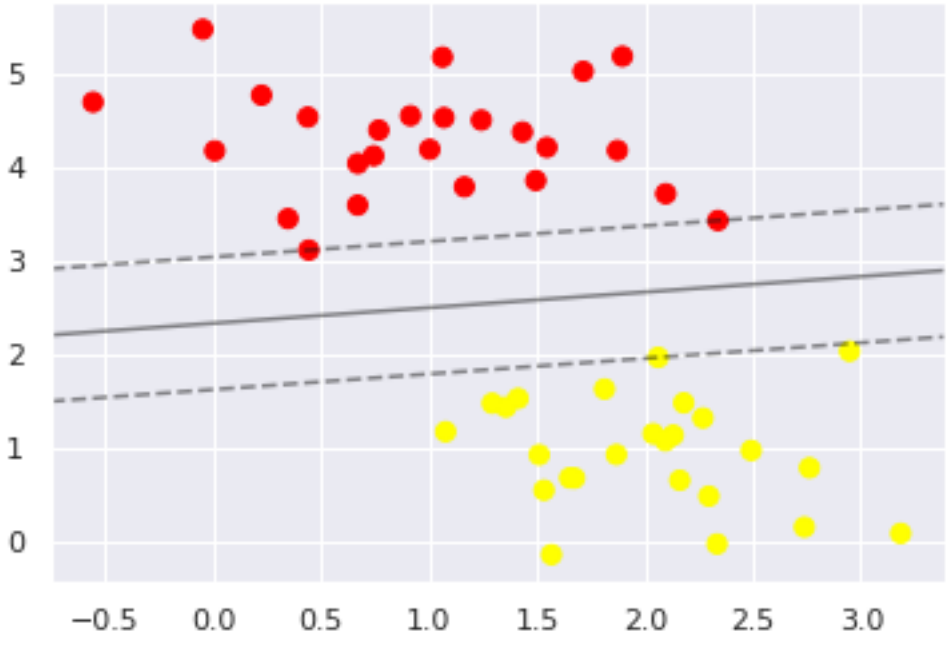
3.5.9 Using Scikit-learn to fit a linear support vector machine

```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

```
SVC(C=10000000000.0, break_ties=False, cache_size=200, class_weight=
None,
    coef0=0.0, decision_function_shape='ovr', degree=3, gamma='scale',
    kernel='linear', max_iter=-1, probability=False, random_state=None
,
    shrinking=True, tol=0.001, verbose=False)
```

3.5.10 The fitted model

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model, plot_support=True);
```



- This is the dividing line that maximizes the margin between the two sets of points.
- Notice that a few of the training points just touch the margin.
- These points are are known as the *support vectors*.

3.5.11 Support-vectors

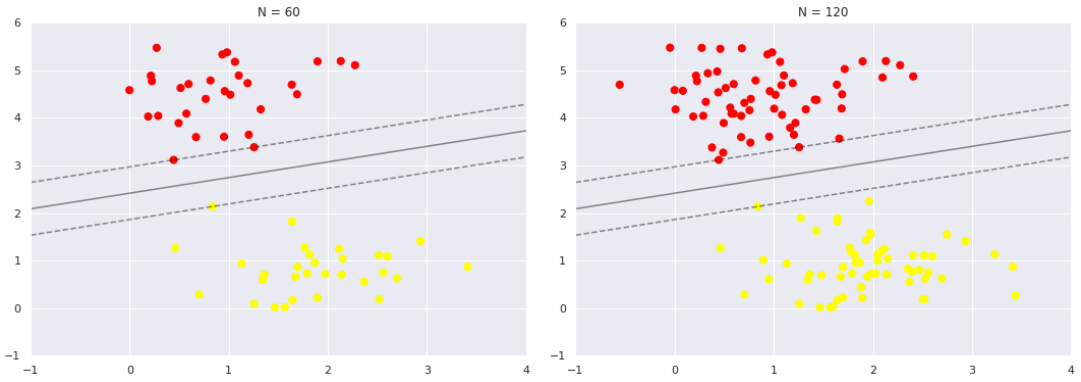
- In Scikit-Learn, the identity of these points are stored in the `support_vectors_` attribute of the classifier:

```
model.support_vectors_
```

```
array([[0.44359863, 3.11530945],
       [2.33812285, 3.43116792],
       [2.06156753, 1.96918596]])
```

- Only the position of the support vectors matter
- Any points further from the margin which are on the correct side do not modify the fit.
- These points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.
- We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset:

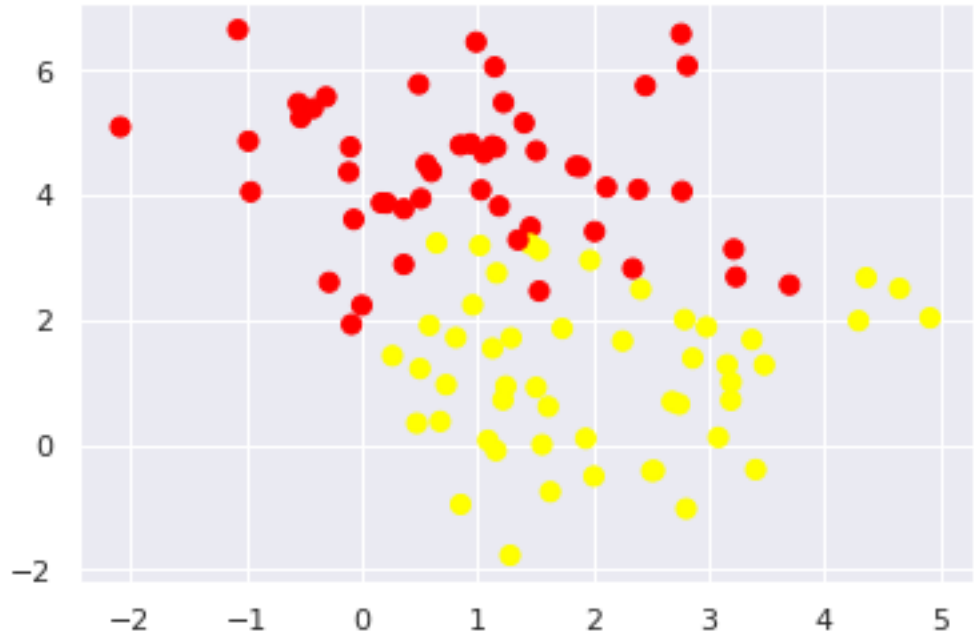
3.5.12 Support-vector machine fit with different training sets



3.5.13 Overlapping data

- Consider a data set that has some overlap, e.g.:

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```



3.5.14 Soft margins

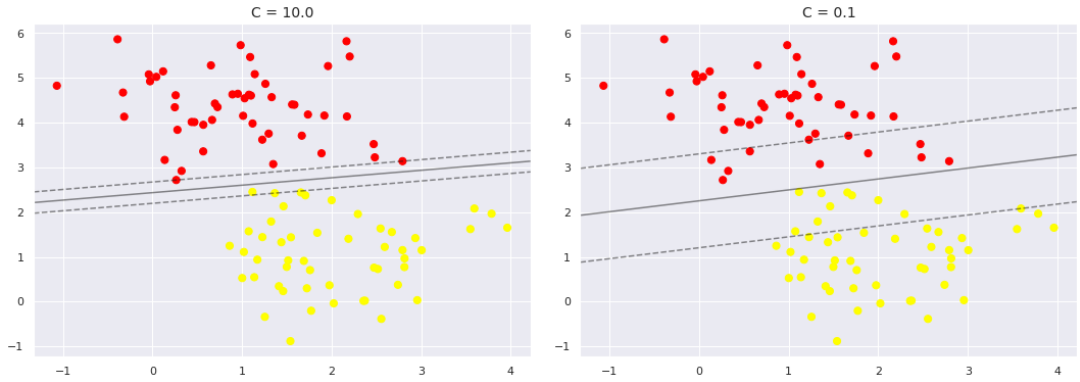
- We introduce a non-negative hyper-parameter C which "softens" the margin, and our constraint becomes:

$$y_i(B_0 + B_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip})) \geq M(1 - \xi_i) \; \forall i \in \{1, \ldots, n\}, \tag{3.14}$$

$$\xi_i \geq 0, \; \sum_{i=1}^{n} \xi_i \leq C. \tag{3.15}$$

3.5.15 Softening the margins

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```



3.5.16 SVM as constrained optimization

- Define the functional margin:

$$M = \min_i y_i \left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x}_i + \frac{b}{||\mathbf{w}||} \right) \tag{3.16}$$

- We want to solve

$$\max_{\mathbf{w},b} M \tag{3.17}$$

subject to:

$$y_i \left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x}_i + \frac{b}{||\mathbf{w}||} \right) \geq M \; \forall i \tag{3.18}$$

3.5.16.1 Eliminating M

Define

$$F = \min_i y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \tag{3.19}$$

Then the above constraint can be written:

$$y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq F \; \forall i \tag{3.20}$$

- We can normalize by rescaling $\mathbf{w}$ so that $F = 1$ which gives us $M = \frac{1}{||\mathbf{w}||}$.

3.5.16.2 Reformulating as a quadratic

Our optimization problem is now:

$$\max_{\mathbf{w},b} \frac{1}{||\mathbf{w}||} \tag{3.21}$$

subject to:

$$y_i (\mathbf{w} \cdot x + b) - 1 \geq 0 \; \forall i \tag{3.22}$$

- This can be written as a quadratic optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 \tag{3.23}$$

subject to the same constraint:

$$y_i(\mathbf{w} \cdot x + b) - 1 \geq 0 \; \forall i \tag{3.24}$$

3.5.16.3 Softening the margins

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + \frac{C}{n}\sum_i^n \xi_i \tag{3.25}$$

subject to constraints:

$$\begin{aligned} y_i(\mathbf{w} \cdot x + b) &\geq 1 - \xi_i \; \forall i \\ \xi_i &\geq 0 \; \forall i \end{aligned}$$

3.5.16.4 Standard form

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_i^n \xi_i \tag{3.26}$$

subject to constraints:

$$\begin{aligned} 1 - \xi_i - y_i(\mathbf{w} \cdot x + b) &\leq 0 \; \forall i \\ -\xi_i &\leq 0 \; \forall i \end{aligned}$$

3.5.16.5 KKT conditions

$$\nabla f(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(\mathbf{x}^*) + \sum_{i=1}^{p} v_i^* \nabla h_i(\mathbf{x}^*) = 0 \tag{3.27}$$

$$g_i(\mathbf{x}^*) \leq 0, \; i = 1, \ldots, m \tag{3.28}$$

$$h_i(\mathbf{x}^*) = 0, \;\; i = 1, \ldots, p \tag{3.29}$$

$$\lambda_i^* \geq 0, \; i = 1, \ldots, m \tag{3.30}$$

$$\lambda_i^* g_i(\mathbf{x}^*) = 0, \; i = 1, \ldots, m \tag{3.31}$$

3.5.16.6 Lagrangian Allocating Lagrange multipliers **ff, fi**:

$$L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \frac{C}{n}\sum_{i=1} \xi_i + \sum_{i=1}^{n} \alpha_i(1 - \xi_i - y_i(\mathbf{w}^T x_i + b)) - \sum_{i=1}^{n} \beta_i \xi_i \tag{3.32}$$

$$\tag{3.33}$$

3.5.16.7 Lagrangian dual

$$L_D(\mathbf{ff}, \mathbf{fi}) = \min_{\mathbf{w}, b, \xi_i} L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) \tag{3.34}$$

The dual optimization problem is:

$$\max_{\mathbf{w}, b, \xi_i} L_D(\mathbf{ff}, \mathbf{fi}) \tag{3.35}$$

3.5.16.8 KTT conditions

$$L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \frac{C}{n}\sum_{i=1}\xi_i + \sum_{i=1}^{n}\alpha_i(1 - \xi_i - y_i(\mathbf{w}^T x_i + b)) - \sum_{i=1}^{n}\beta_i\xi_i \tag{3.36}$$

$$\tag{3.37}$$

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) = \mathbf{w} - \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i = 0 \tag{3.38}$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) = -\sum_{i=1}^{n}\alpha_i y_i = 0 \tag{3.39}$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \text{,}, \mathbf{ff}, \mathbf{fi}) = \frac{C}{n} - \alpha_i - \beta_i = 0 \tag{3.40}$$

3.5.16.9 Substituting into the dual

$$L_D(\mathbf{ff}, \mathbf{fi}) = -\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i}^{n}\alpha_i \tag{3.41}$$

- We can now write the dual optimization problem as:

$$\max_{\mathbf{ff}, \mathbf{fi}} -\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i}^{n}\alpha_i \tag{3.42}$$

subject to:

$$\sum_i \alpha_i y_i = 0 \tag{3.43}$$

$$\alpha_i + \beta_i = \frac{C}{n} \tag{3.44}$$

$$\alpha_i, \beta_i \geq 0 \; i = 1, \ldots, n \tag{3.45}$$

3.5.16.10 As a quadratic programming problem

We an eliminate **fi** to transform this into a quadratic programming problem:

$$\max_{\mathbf{ff}} -\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_i \alpha_i \tag{3.46}$$

subject to:

$$\sum_i \alpha_i y_i = 0 \tag{3.47}$$

$$0 \leq \alpha_i \leq \frac{C}{n}\ i = 1, \ldots, n \tag{3.48}$$

$$\tag{3.49}$$

- This can be solved numerically using a quadratic optimizer to obtain $\mathbf{ff}^*$.

3.5.16.11 Optimal weights

- From KKT condition 1:

$$\mathbf{w} * - \sum_{i=1}^{n} \alpha_i^* y_i \mathbf{x}_i = 0 \tag{3.50}$$

Thus:

$$\mathbf{w}* = \sum_{i=1} \alpha_i^* y_i \mathbf{x}_i \tag{3.51}$$

3.5.16.12 Support vectors

- From KKT condition 5:

$$\alpha_i^*(1 - \xi_i^* - y_i(\mathbf{w}^{*^T}\mathbf{x}_i + b^*)) = 0 \tag{3.52}$$

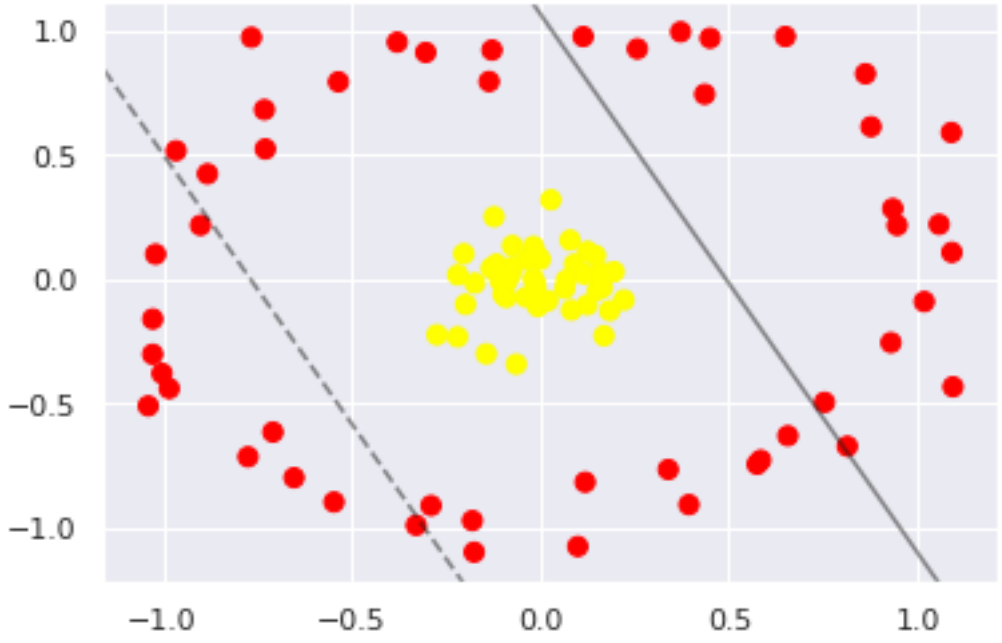
- The data that satisfy these constraints are the support vectors.
- By the KKT conditions, $\alpha_i^* \neq 0$ i.f.f. $\mathbf{x}_i$ is a support vector.
- In typical learning problems only a small fraction of the training inputs are support vectors S.
- Our classifier has a compact representation:

$$F(\mathbf{x}_j) = \text{sign}(\sum_i \alpha_i^* y_i \mathbf{x}_j^T \mathbf{x}_i + b^*) \tag{3.53}$$

3.5.17 Non linearly-seperable data

```
from sklearn.datasets.samples_generator import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

clf = SVC(kernel='linear').fit(X, y)

```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf, plot_support=False);
```



3.5.18 Radial basis functions

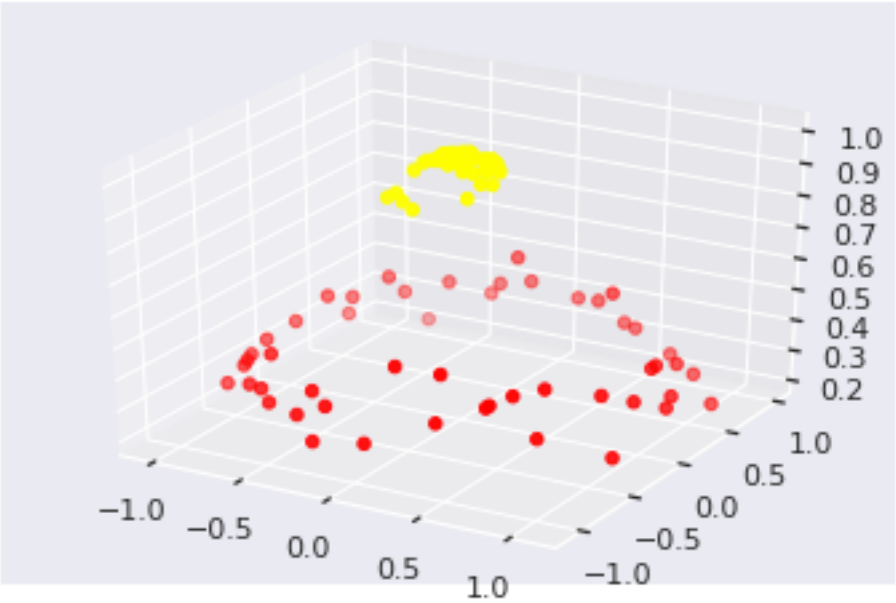
- We can project the data into higher dimensions to obtain linear-separability.
- For example, we can use a radial basis function:

```
r = np.exp(-(X ** 2).sum(1))
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent
↪ call last)

<ipython-input-1-fc15a303f7b6> in <module>
----> 1 r = np.exp(-(X ** 2).sum(1))

NameError: name 'np' is not defined
```

3.5.19 Projection into three dimensions

```
ax = plt.subplot(projection='3d')
ax.scatter3D(X[:, 0], X[:, 1], r, c=y, cmap='autumn')
plt.show()
```



3.5.20 Inner products

- The solution to the linear support-vector classifier optimization problem involves only the inner products of the observations:

$$\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j} \tag{3.54}$$

- The linear SVC can thus be represented:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle \tag{3.55}$$

3.5.21 The kernel trick

- A kernel is a function $\mathrm{k}(x_i, x_{i'})$ which quantifies the similarity of two observations:
- The linear kernel is:

$$\mathrm{k}(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j} \tag{3.56}$$

- If we use centered and scaled data, it is equivalent to the Pearson correlation.
- We can use non-linear kernels, e.g. polynomial of degree d:

$$\mathrm{k}(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d \tag{3.57}$$

3.5.21.1 Kernel-trick illustration Consider

$$\phi(x) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)\phi(z) = (1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, z_2^2, \sqrt{2}z_1z_2) \tag{3.58}$$

$$\begin{aligned} \phi(x) \cdot \phi(z) &= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 && (3.59)\\ &= (1 + x_1z_1 + x_2z_2)^2 && (3.60)\\ &= (1 + x.z)^2 && (3.61) \end{aligned}$$

3.5.22 Non-linear kernels in scikit-learn

To use the radial basis function kernel (RBF):

$$\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}\right) \tag{3.62}$$

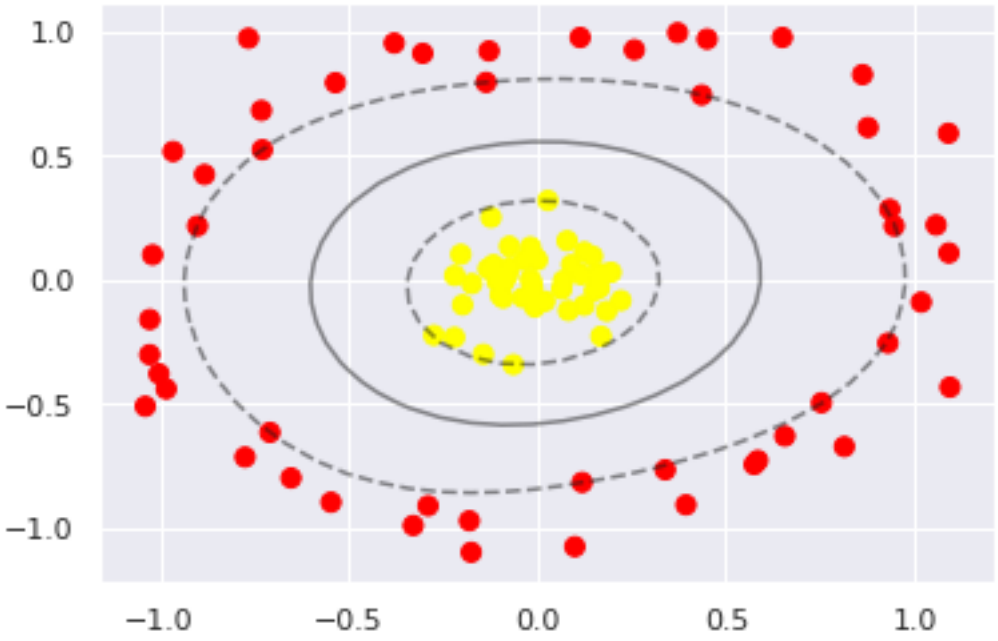
```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

```
SVC(C=1000000.0, break_ties=False, cache_size=200, class_weight=None,
coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='
rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

3.5.23 Plotting the decision boundary

- We can plot the decision boundary in the unmapped space:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
```



3.5.24 Modeling high-frequency limit-order book dynamics

- In Kercheval & Zhang (2015) the feature vector consists of the prices and volumes on each side of the order-book.
- The model is trained to predict the movement of the mid-price in the subsequent time period.

Kercheval, A. N., & Zhang, Y. (2015). Modelling high-frequency limit order book dynamics with support vector machines. Quantitative Finance, 15(8), 1315-1329.

3.5.24.1 Categories

1. Mid-price will move up
2. Mid-price will move down
3. Mid-price will stay the same

3.5.24.2 Features (Kercheval and Zhang 2015)

3.5.24.3 Results

3.5.24.4 Validation

Basic Set	Description(i = *level index*, $n = 10$)
$v_1 = \{P_i^{ask},\ V_i^{ask},\ P_i^{bid},\ V_i^{bid}\}_{i=1}^{n},$	*price and volume (n levels)*

Time-insensitive Set	Description(i = *level index*)
$v_2 = \{(P_i^{ask} - P_i^{bid}), (P_i^{ask} + P_i^{bid})/2\}_{i=1}^{n},$	*bid-ask spreads and mid-prices*
$v_3 = \{P_n^{ask} - P_1^{ask}, P_1^{bid} - P_n^{bid}, \lvert P_{i+1}^{ask} - P_i^{ask}\rvert, \lvert P_{i+1}^{bid} - P_i^{bid}\rvert\}_{i=1}^{n},$	*price differences*
$v_4 = \{\frac{1}{n}\sum_{i=1}^{n} P_i^{ask},\ \frac{1}{n}\sum_{i=1}^{n} P_i^{bid},\ \frac{1}{n}\sum_{i=1}^{n} V_i^{ask},\ \frac{1}{n}\sum_{i=1}^{n} V_i^{bid}\},$	*mean prices and volumes*
$v_5 = \{\sum_{i=1}^{n}(P_i^{ask} - P_i^{bid}),\ \sum_{i=1}^{n}(V_i^{ask} - V_i^{bid})\},$	*accumulated differences*

Time-sensitive Set	Description(i = *level index*)
$v_6 = \{dP_i^{ask}/dt,\ dP_i^{bid}/dt,\ dV_i^{ask}/dt,\ dV_i^{bid}/dt\}_{i=1}^{n},$	*price and volume derivatives*
$v_7 = \{\lambda_{\Delta t}^{la},\ \lambda_{\Delta t}^{lb},\ \lambda_{\Delta t}^{ma},\ \lambda_{\Delta t}^{mb},\ \lambda_{\Delta t}^{ca},\ \lambda_{\Delta t}^{cb}\ \}$	*average intensity of each type*
$v_8 = \{\mathbf{1}_{\{\lambda_{\Delta t}^{la} > \lambda_{\Delta T}^{la}\}},\ \mathbf{1}_{\{\lambda_{\Delta t}^{lb} > \lambda_{\Delta T}^{lb}\}},\ \mathbf{1}_{\{\lambda_{\Delta t}^{ma} > \lambda_{\Delta T}^{ma}\}},\ \mathbf{1}_{\{\lambda_{\Delta t}^{mb} > \lambda_{\Delta T}^{mb}\}}\},$	*relative intensity indicators*
$v_9 = \{d\lambda^{ma}/dt,\ d\lambda^{lb}/dt,\ d\lambda^{mb}/dt,\ d\lambda^{la}/dt\},$	*accelerations(market/limit)*

Table 2: Feature vector sets

Figure 3.1: features

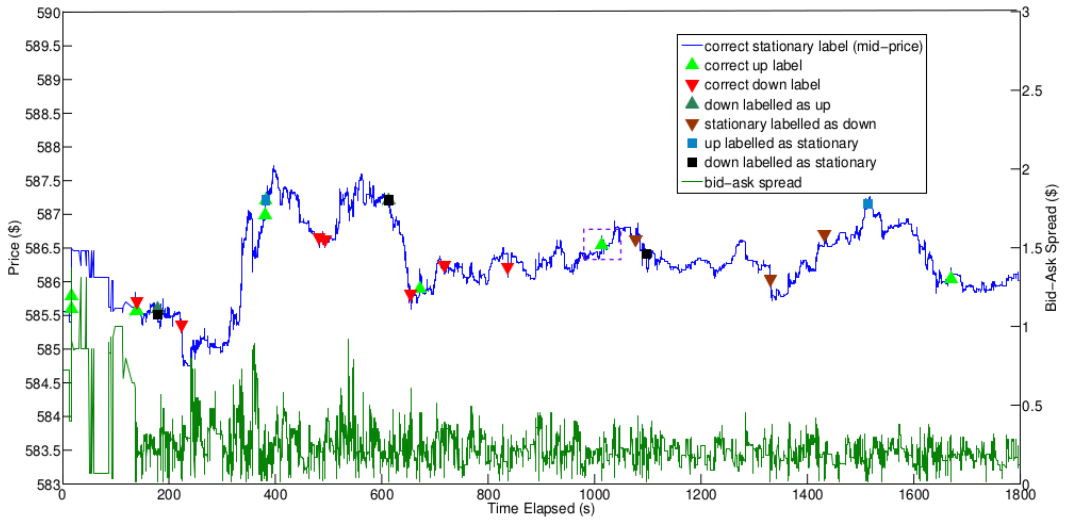


Figure 3.2: ts

AAPL: Mid-price

	Label	$\mathcal{F}_0$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\Delta_{(1,2)}$	$\Delta_{(0,1)}$	$\Delta_{(0,2)}$	$\Delta_{(0,3)}$	Avg.
P(%)	U (↑)	83.6	89.5	85.3	84.7	-4.2	5.9	1.7	1.1	2.9
	D (↓)	75.2	88.0	84.9	84.3	-3.1	12.8	9.7	9.1	10.5
	S (–)	79.8	90.0	80.1	81.9	-9.9	10.2	0.3	2.1	4.2
R(%)	U (↑)	76.5	89.5	77.4	76.5	-12.1	13.0	0.9	0.0	4.6
	D (↓)	80.5	88.0	82.9	83.2	-5.1	7.5	2.4	2.7	4.2
	S (–)	81.0	90.0	83.8	82.4	-6.2	9.0	2.8	1.4	4.4
F_1(%)	U (↑)	79.9	89.5	81.4	80.6	-8.1	9.6	1.5	0.7	3.9
	D (↓)	77.8	88.0	83.9	83.4	-4.1	10.2	6.1	5.6	7.3
	S (–)	80.4	90.0	82.0	82.2	-8.0	9.6	1.6	1.8	4.3

Figure 3.3: validation

3.6 The Bayes Classifier

- The Bayes Classifier is the theoretically-ideal classifier.
- We know a-priori the probability distributions for each class $P(X|Y = j)$.
- We assign each observation to the most likely class, given its predictor values.
- For a given data point $\mathbf{x}_i$ we choose the label j for which maximises:

$$Pr(Y = j|X = \mathbf{x}_i) \tag{3.63}$$

3.6.1 Bayes Rule

$$P(A|B) = \frac{P(B|A)}{P(A)} \tag{3.64}$$

- For a binary classifier with mutually exclusive labels $Y = 0$ or $Y = 1$:

$$Pr(Y = 1|X) = \frac{Pr(X|Y = 1)Pr(Y = 1)}{Pr(X)} \tag{3.65}$$

$$\tag{3.66}$$

- Under the assumption that predictors are independent we can use:

$$P(Y|X) = P(x_1|Y) \times P(x_2|Y) \times \ldots \times P(x_n|Y) \times P(Y) \tag{3.67}$$

- The above assumption is (sometimes) "naive", and gives us the name "Naive Bayes Classifier".

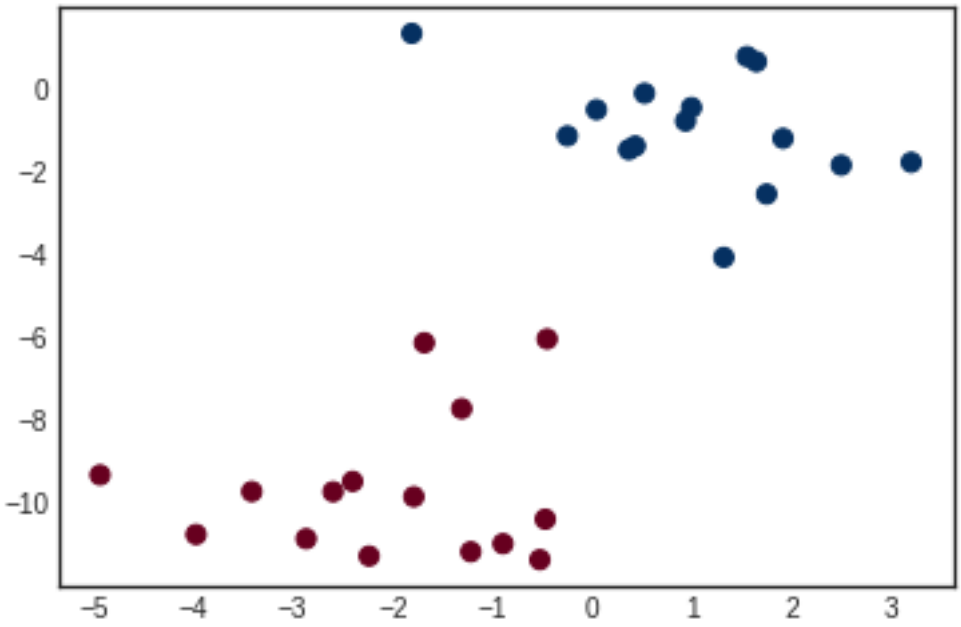
3.6.2 The Bayes Decision Boundary

- For a binary classification problem with mutually exclusive labels $Y = 0$ or $Y = 1$ we use the Bayes decision boundary:
- For a binary classification task with two labels $Pr(Y = 1) = 1 - Pr(Y = 0)$.
- Therefore the decision boundary is $\{\mathbf{x}_i \in \mathbf{X} : P(Y = 1|X = \mathbf{x}_i) > 0.5\}$.

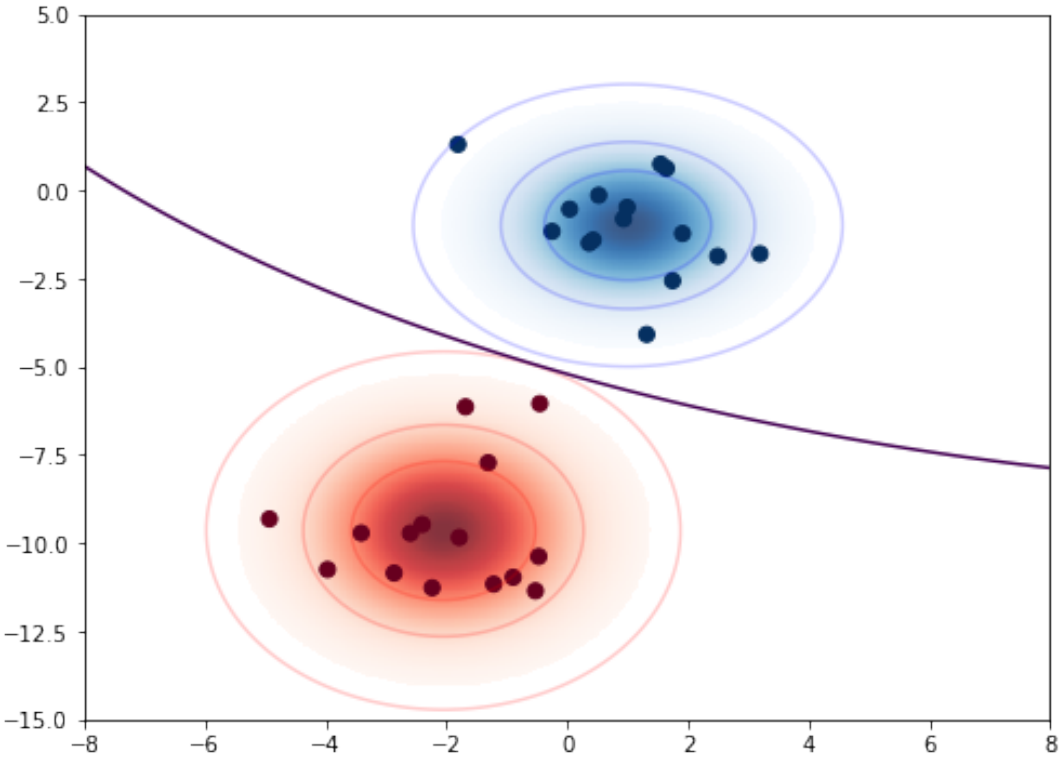
$$Pr(Y = 1|X) > 0.5 \tag{3.68}$$

- To illustrate, we first generate data using known conditional probabilities:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(30, 2, centers=2, random_state=2, cluster_std
    ↪ =1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```



3.6.3 Plotting the decision boundary



3.6.4 Naive Gaussian-Bayes Classifier

- We can make assumptions about the model for each label.

- In a Gaussian Naive Bayes classification we *assume* that data for each feature are i.i.d. distributed from a Gaussian distribution:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma^2 y}\right) \tag{3.69}$$

- The mean and variance can be estimated from sample mean and sample variance.
- In scikit-learn:

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y);
```

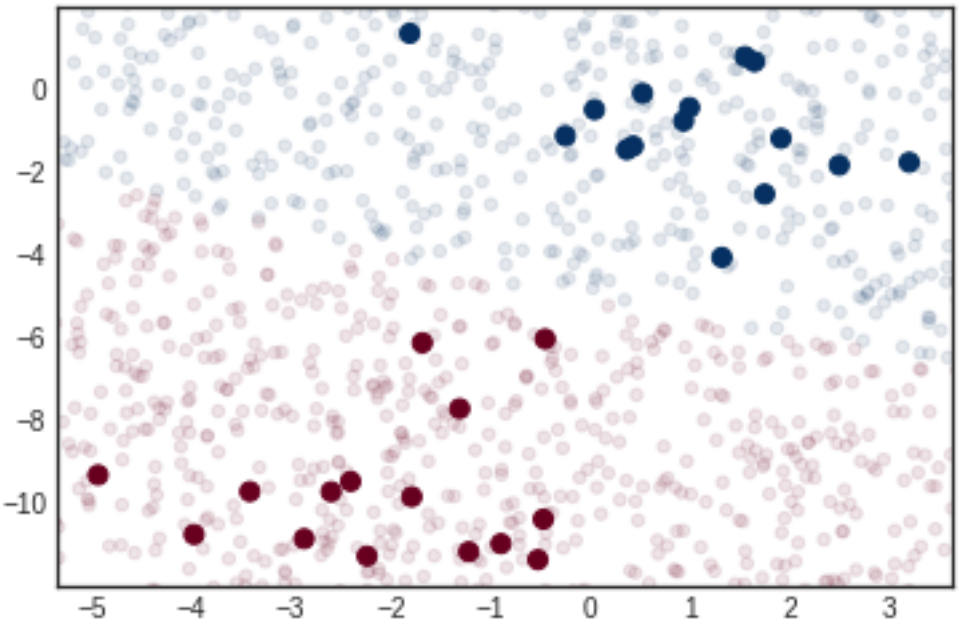
3.6.4.1 Predictions

- We can now predict the label for new observations:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)
```

3.6.4.2 Decision boundary

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu',
    ↪ alpha=0.1)
plt.axis(lim);
```



3.6.4.3 Predicted posterior probabilities

```
yprob = model.predict_proba(Xnew)
yprob[-8:].round(2)
```

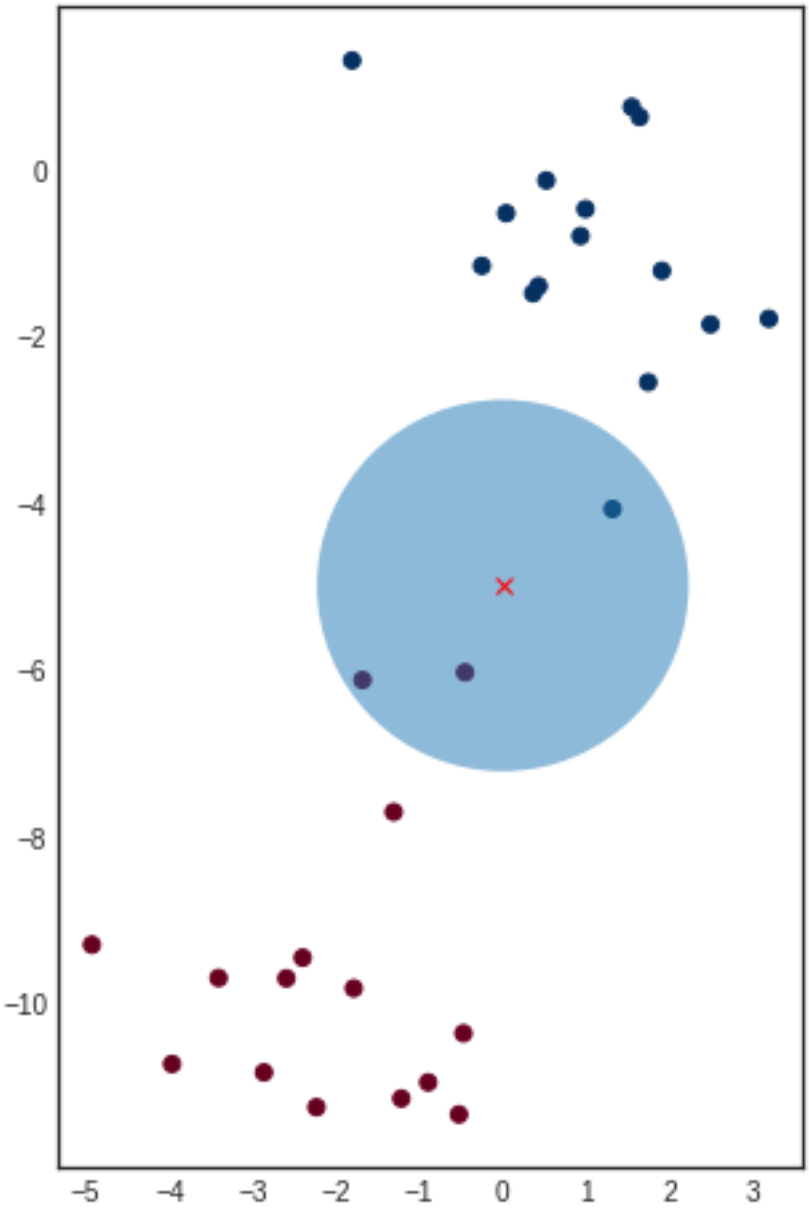
```
array([[1.  , 0.  ],
       [1.  , 0.  ],
       [1.  , 0.  ],
       [1.  , 0.  ],
       [1.  , 0.  ],
       [1.  , 0.  ],
       [0.  , 1.  ],
       [0.98, 0.02]])
```

3.7 K-Nearest Neighbors

- In general we do not know a-priori the actual conditional probabilities.
- Therefore we have to *estimate* them.
- With K-nearest neighbors (KNN) we first identify the K points in the training data that are closest to a given training data x_0, e.g. using an Euclidian distance metric.
- We then estimate the conditional probability as a sample mean:

$$Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in N_0} I(y_i = j) \tag{3.70}$$

3.7.1 KNN example for $K = 3$



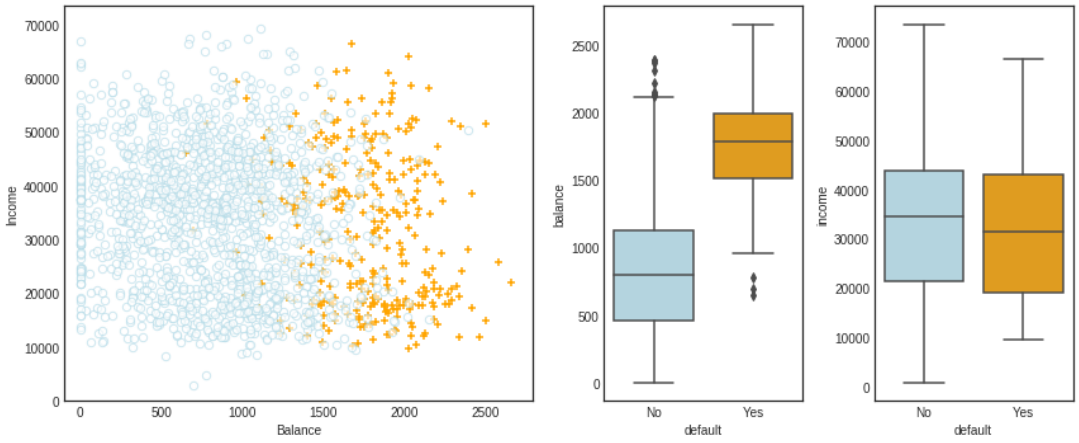
4 Logistic regression

- So far we have modelled dependent variables as deterministic quantities.
- In many applications we want to forecast probabilities, e.g. probability of default.

4.1 Credit data set

	default	student	balance	income
0	No	No	729.526495	44361.625074
1	No	Yes	817.180407	12106.134700
2	No	No	1073.549164	31767.138947
3	No	No	529.250605	35704.493935
4	No	No	785.655883	38463.495879

4.2 Credit data-set: visualisation



4.3 Dummy variables and probabilities

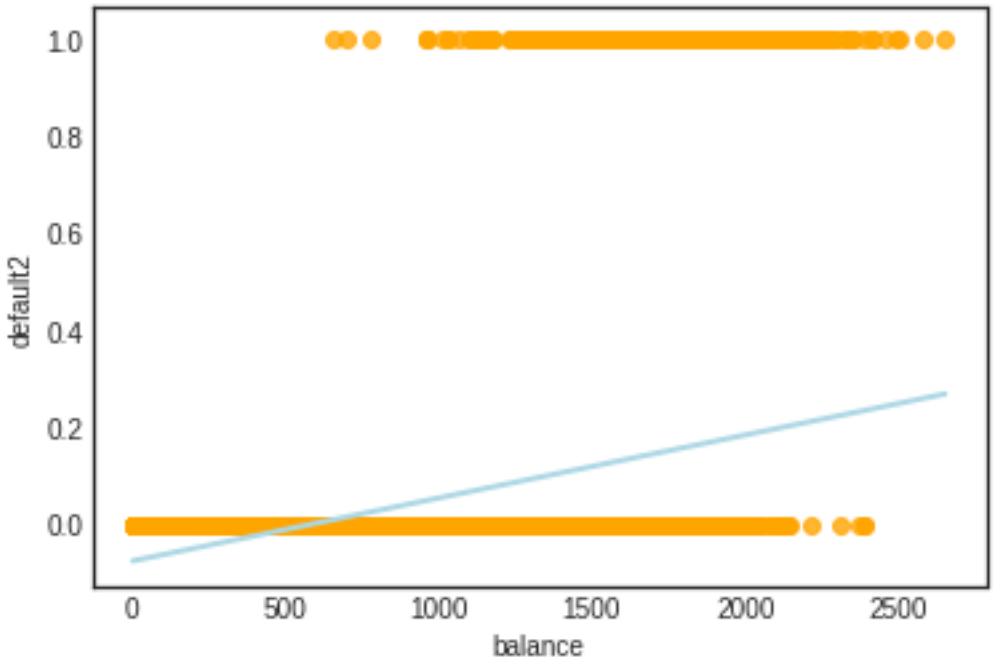
- We could map labels onto integers, but we have to take great care as labels have no meaningful intrinsic ordering.
- For a binary classifier with only two labels we can use dummy variables by transforming the labels onto 0 and 1.
- We can straightforwardly use dummy-variables for predictors.
- We often try to interpret the real-valued response as a *probability*.

4.4 Probabilities are not linear in predictors

- Naively, we might attempt to fit a model of the form:

$$p(X) = \beta_0 + \beta_1 X \tag{4.1}$$

- However, this will give nonsensical results.



4.5 Logistic regression assumptions

- We use a sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{4.2}$$

- for a single training pair $(\mathbf{x}, y)$, we assume:

$$P(Y = 1|\mathbf{X} = \mathbf{x}) = \sigma(z) \tag{4.3}$$

where:

$$z = w_0 + \sum_{i=1}^{m} w_i x_i \tag{4.4}$$

4.6 Vectorized conditional probabilities

$$\begin{aligned} P(Y = 1|\mathbf{X} = \mathbf{x}) &= \sigma(\mathbf{w}^T\mathbf{x}) && (4.5)\\ P(Y = 0|\mathbf{X} = \mathbf{x}) &= 1 - \sigma(\mathbf{w}^T\mathbf{x}) && (4.6)\\ & && (4.7) \end{aligned}$$

If the labels are 0 and 1, then Y is a Bernoulli random variable $Y \sim Ber(p)$ where $p = \sigma(\mathbf{w}^T\mathbf{x})$, therefore:

$$P(Y = y|X = \mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})^y \cdot \left[1 - \sigma(\mathbf{w}^T\mathbf{x})\right]^{(1-y)} \tag{4.8}$$

4.7 Likelihood function

$$
\begin{aligned}
L(\mathbf{w}) &= \prod_{i=1}^{n} P(Y = y_i | X = \mathbf{x}_i) \\
L(\mathbf{w}) &= \prod_{i=1}^{n} \sigma(\mathbf{w}^T \mathbf{x}_i)^y \cdot \left[1 - \sigma(\mathbf{w}^T \mathbf{x}_i)\right]^{(1-y)}
\end{aligned}
$$

4.8 Log-likelihood function

$$
LL(\mathbf{w}) = \sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i)^y \log \left[1 - \sigma(\mathbf{w}^T \mathbf{x}_i)\right]
$$

- To obtain optimal weights we maximise the above, which gives us the maximum likelihood estimate (MLE).
- Provided that the data are not separated, this function is concave and can be solved numerically using gradient ascent.

4.9 Gradient function

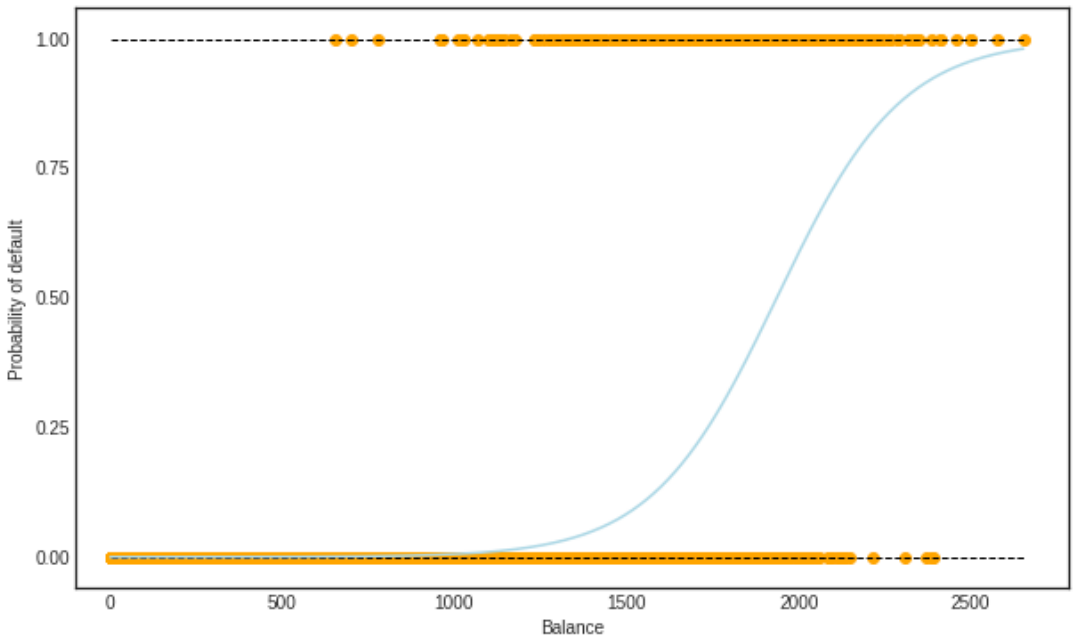
$$
\frac{\partial LL(\mathbf{w})}{\partial w_j} = \sum_{i=1}^{n} \left[y_i - \sigma(\mathbf{w}^T \mathbf{x}_i)\right] x_{ij} \tag{4.9}
$$

4.10 Logistic regression in scikit-learn

```
from sklearn.linear_model import LogisticRegression

logistic = skl_lm.LogisticRegression(solver='newton-cg')
logistic.fit(X_train, y)
print(clf)
print('classes: ',clf.classes_)
print('coefficients: ',clf.coef_)
print('intercept :', clf.intercept_)
```

```
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=None, solver='newton-cg', tol=0.0001,
verbose=0,
                   warm_start=False)
classes:  [0 1]
coefficients:  [[0.00549891]]
intercept : [-10.65132393]
```

4.10.1 Probability of default



5 Bibliography

Bianchi, D., & McAlinn, K. (2020). Divide and Conquer: Financial Ratios and Industry Returns Predictability (No. 3136368; SSRN). https://doi.org/10.2139/ssrn.3136368

Bhattacharyya, S., Jha, S., Tharakunnel, K., & Westland, J. C. (2011). Data mining for credit card fraud: A comparative study. Decision Support Systems, 50(3), 602-613.

Doumpos, M., & Zopounidis, C. (2002). Classification problems in finance. Multicriteria Decision Aid Classification Methods, 159-224.

Friedman, J., Hastie, T., & Tibshirani, R. (2001). The elements of statistical learning (Vol. 1, No. 10). New York: Springer series in statistics.

Gilli, M., Maringer, D., & Winker, P. (2008). Applications of heuristics in finance. In Handbook on information technology in finance (pp. 635-653). Springer, Berlin, Heidelberg.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning (Vol. 112, pp. 3-7). New York: Springer.

Kercheval, A. N., & Zhang, Y. (2015). Modelling high-frequency limit order book dynamics with support vector machines. Quantitative Finance, 15(8), 1315-1329.

Murphy, K. P. (2012). Machine learning: a probabilistic perspective. MIT press.

VanderPlas, J. (2016). Python data science handbook: Essential tools for working with data. " O'Reilly Media, Inc.".

5.0.1 Acknowledgements

Some of the materials from the above adapted for these notes under CC BY-SA 4.0.