ADVANCES IN FINANCIAL MACHINE LEARNING

BY MARCOS LÓPEZ DE PRADO

Contents

TABLE 1.1 Overview of the Challenges Addressed by Every Chapter

Part	Chapter	Fin. data	Software	Hardware	Math	Meta-Strat	Overfitting
1	2	X	X				
1	3	X	X				
1	4	X	X				
1	5	X	X		X		
2	6		X				
2	7		X			X	X
2	8		X			X	
2	9		X			X	
3	10		X			X	
3	11		X		X		X
3	12		X		X		X
3	13		X		X		X
3	14		X		X		X
3	15		X		X		X
3	16		X		X	X	X
4	17	X	X		X		
4	18	X	X		X		
4	19	X	X				
5	20		X	X	X		
5	21		X	X	X		
5	22		X	X	X		

TABLE 1.2 Common Pitfalls in Financial ML

#	Category	Pitfall	Solution	Chapter
1	Epistemological	The Sisyphus paradigm	The meta-strategy paradigm	1
2	Epistemological	Research through backtesting	Feature importance analysis	8
3	Data processing	Chronological sampling	The volume clock	2
4	Data processing	Integer differentiation	Fractional differentiation	5
5	Classification	Fixed-time horizon labeling	The triple-barrier method	3
6	Classification	Learning side and size simultaneously	Meta-labeling	3
7	Classification	Weighting of non-IID samples	Uniqueness weighting; sequential bootstrapping	4
8	Evaluation	Cross-validation leakage	Purging and embargoing	7, 9
9	Evaluation	Walk-forward (historical) backtesting	Combinatorial purged cross-validation	11, 12
10	Evaluation	Backtest overfitting	Backtesting on synthetic data; the deflated Sharpe ratio	10–16

TABLE 2.1 The Four Essential Types of Financial Data

Fundamental Data	Market Data	Analytics	Alternative Data
• Assets • Liabilities • Sales • Costs/earnings • Macro variables • . . .	• Price/yield/implied volatility • Volume • Dividend/coupons • Open interest • Quotes/cancellations • Aggressor side • . . .	• Analyst recommendations • Credit ratings • Earnings expectations • News sentiment • . . .	• Satellite/CCTV images • Google searches • Twitter/chats • Metadata • . . .

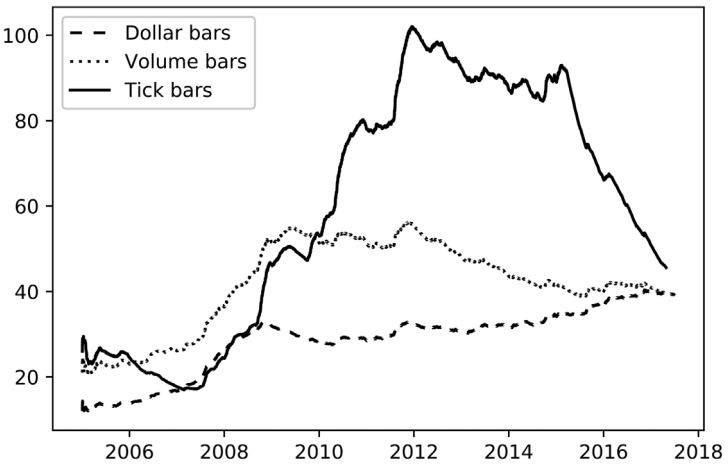


FIGURE 2.1 Average daily frequency of tick, volume, and dollar bars

$$b_t = \begin{cases} b_{t-1} & \text{if } \Delta p_t = 0 \\ \dfrac{|\Delta p_t|}{\Delta p_t} & \text{if } \Delta p_t \neq 0 \end{cases}$$

Equation 1

$$\theta_T = \sum_{t=1}^{T} b_t$$

Equation 2

$$T^* = \arg\min_{T} \left\{ |\theta_T| \geq \mathrm{E}_0 [T] \left| 2\mathrm{P} [b_t = 1] - 1 \right| \right\}$$

Equation 3

$$\theta_T = \sum_{t=1}^{T} b_t v_t$$

Equation 4

$$\mathrm{E}_0[\theta_T] = \mathrm{E}_0\left[\sum_{t|b_t=1}^{T} v_t\right] - \mathrm{E}_0\left[\sum_{t|b_t=-1}^{T} v_t\right] = \mathrm{E}_0[T](\mathrm{P}[b_t = 1]\mathrm{E}_0[v_t|b_t = 1] - \mathrm{P}[b_t = -1]\mathrm{E}_0[v_t|b_t = -1])$$

Equation 5

$$v^+ = \mathrm{P}[b_t = 1]\mathrm{E}_0[v_t|b_t = 1], \quad v^- = \mathrm{P}[b_t = -1]\mathrm{E}_0[v_t|b_t = -1]$$

Equation 6

$$\mathrm{E}_0[T]^{-1}E_0[\textstyle\sum_t v_t] = \mathrm{E}_0[v_t] = v^+ + v^-$$

Equation 7

$$\mathrm{E}_0[\theta_T] = \mathrm{E}_0[T](v^+ - v^-) = \mathrm{E}_0[T](2v^+ - \mathrm{E}_0[v_t])$$

Equation 8

$$T^* = \arg\min_T \{|\theta_T| \geq \mathrm{E}_0[T]|2v^+ - \mathrm{E}_0[v_t]|\}$$

Equation 9

$$\theta_T = \max\left\{\sum_{t|b_t=1}^{T} b_t, - \sum_{t|b_t=-1}^{T} b_t\right\}$$

Equation 10

$$\mathrm{E}_0[\theta_T] = \mathrm{E}_0[T]\max\{\mathrm{P}[b_t = 1],\ 1 - \mathrm{P}[b_t = 1]\}$$

Equation 11

$$T^* = \arg\min_T\{\theta_T \geq \mathrm{E}_0[T]\max\{\mathrm{P}[b_t = 1], 1 - \mathrm{P}[b_t = 1]\}\}$$

Equation 12

$$\theta_T = \max\left\{\sum_{t|b_t=1}^{T} b_t v_t, -\sum_{t|b_t=-1}^{T} b_t v_t\right\}$$

Equation 13

$$\mathrm{E}_0[\theta_T] = \mathrm{E}_0[T]\max\{\mathrm{P}[b_t = 1]\mathrm{E}_0[v_t|b_t = 1], (1 - \mathrm{P}[b_t = 1])\mathrm{E}_0[v_t|b_t = -1]\}$$

Equation 14

$$T^* = \arg\min_T\{\theta_T \geq \mathrm{E}_0[T]\max\{\mathrm{P}[b_t = 1]\mathrm{E}_0[v_t|b_t = 1], (1 - \mathrm{P}[b_t = 1])\mathrm{E}_0[v_t|b_t = -1]\}\}$$

Equation 15

$$\max\{\mathrm{P}[b_t = 1]\mathrm{E}_0[v_t|b_t = 1],(1 - \mathrm{P}[b_t = 1])\mathrm{E}_0[v_t|b_t = -1]\}$$

Expression 1

$$h_{i,t} = \begin{cases} \dfrac{\omega_{i,t} K_t}{o_{i,t+1} \varphi_{i,t} \sum_{i=1}^{I} |\omega_{i,t}|} & \text{if } t \in B \\ h_{i,t-1} & \text{otherwise} \end{cases}$$

$$\delta_{i,t} = \begin{cases} p_{i,t} - o_{i,t} & \text{if } (t-1) \in B \\ \Delta p_{i,t} & \text{otherwise} \end{cases}$$

$$K_t = K_{t-1} + \sum_{i=1}^{I} h_{i,t-1} \varphi_{i,t} \left(\delta_{i,t} + d_{i,t} \right)$$

Equations 16, 17, and 18

$$\omega_{i,t} \left(\sum_{i=1}^{I} |\omega_{i,t}| \right)^{-1}$$

Expression 2

1. **Rebalance costs:** The variable cost $\{c_t\}$ associated with the allocation rebalance is $c_t = \sum_{i=1}^{I} (|h_{i,t-1}| p_{i,t} + |h_{i,t}| o_{i,t+1}) \varphi_{i,t} \tau_i$, $\forall t \in B$. We do not embed c_t in K_t, or shorting the spread will generate fictitious profits when the allocation is rebalanced. In your code, you can treat $\{c_t\}$ as a (negative) dividend.
2. **Bid-ask spread:** The cost $\{\tilde{c}_t\}$ of buying or selling one unit of this virtual ETF is $\tilde{c}_t = \sum_{i=1}^{I} |h_{i,t-1}| p_{i,t} \varphi_{i,t} \tau_i$. When a unit is bought or sold, the strategy must charge this cost $\tilde{c}_t$, which is the equivalent to crossing the bid-ask spread of this virtual ETF.
3. **Volume:** The volume traded $\{v_t\}$ is determined by the least active member in the basket. Let $v_{i,t}$ be the volume traded by instrument i over bar t. The number of tradeable basket units is $v_t = \min_i \left\{ \frac{v_{i,t}}{|h_{i,t-1}|} \right\}$.

Equations 19, 20, and 21

$$\sigma^2 = \omega' V \omega = \omega' W \Lambda W' \omega = \beta' \Lambda \beta = (\Lambda^{1/2}\beta)'(\Lambda^{1/2}\beta)$$

Equation 22

$$\sigma^2 = \sum_{n=1}^{N} \beta_n^2 \Lambda_{n,n}$$

Equation 23

$$R_n = \beta_n^2 \Lambda_{n,n} \sigma^{-2} = [W'\omega]_n^2 \Lambda_{n,n} \sigma^{-2}$$

Equation 24

$$\beta = \left\{ \sigma \sqrt{\frac{R_n}{\Lambda_{n,n}}} \right\}_{n=1,\ldots,N}$$

Equation 25

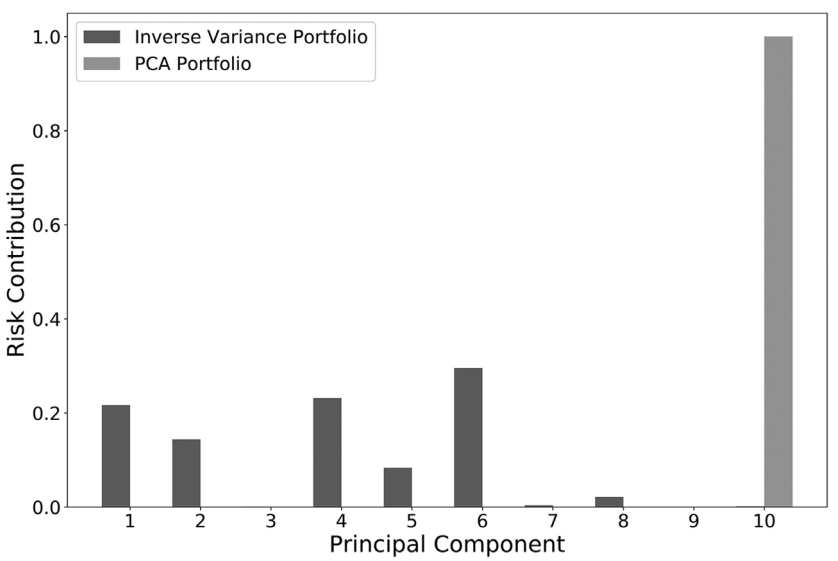


FIGURE 2.2 Contribution to risk per principal component

SNIPPET 2.1 PCA WEIGHTS FROM A RISK DISTRIBUTION R

```
def pcaWeights(cov,riskDist=None,riskTarget=1.):
    # Following the riskAlloc distribution, match riskTarget
    eVal,eVec=np.linalg.eigh(cov) # must be Hermitian
    indices=eVal.argsort()[::-1] # arguments for sorting eVal desc
    eVal,eVec=eVal[indices],eVec[:,indices]
    if riskDist is None:
        riskDist=np.zeros(cov.shape[0])
        riskDist[-1]=1.
    loads=riskTarget*(riskDist/eVal)**.5
    wghts=np.dot(eVec,np.reshape(loads,(-1,1)))
    #ctr=(loads/riskTarget)**2*eVal # verify riskDist
    return wghts
```

SNIPPET 2.2 FORM A GAPS SERIES, DETRACT IT FROM PRICES

```
def getRolledSeries(pathIn,key):
    series=pd.read_hdf(pathIn,key='bars/ES_10k')
    series['Time']=pd.to_datetime(series['Time'],format='%Y%m%d%H%M%S%f')
    series=series.set_index('Time')
    gaps=rollGaps(series)
    for fld in ['Close','VWAP']:series[fld]-=gaps
    return series
#-------------------------------------
def rollGaps(series,dictio={'Instrument':'FUT_CUR_GEN_TICKER','Open':'PX_OPEN', \
    'Close':'PX_LAST'},matchEnd=True):
    # Compute gaps at each roll, between previous close and next open
    rollDates=series[dictio['Instrument']].drop_duplicates(keep='first').index
    gaps=series[dictio['Close']]*0
    iloc=list(series.index)
    iloc=[iloc.index(i)-1 for i in rollDates] # index of days prior to roll
    gaps.loc[rollDates[1:]]=series[dictio['Open']].loc[rollDates[1:]]- \
        series[dictio['Close']].iloc[iloc[1:]].values
    gaps=gaps.cumsum()
    if matchEnd:gaps-=gaps.iloc[-1] # roll backward
    return gaps
```

SNIPPET 2.3 NON-NEGATIVE ROLLED PRICE SERIES

```
raw=pd.read_csv(filePath,index_col=0,parse_dates=True)
gaps=rollGaps(raw,dictio={'Instrument':'Symbol','Open':'Open','Close':'Close'})
rolled=raw.copy(deep=True)
for fld in ['Open','Close']:rolled[fld]-=gaps
rolled['Returns']=rolled['Close'].diff()/raw['Close'].shift(1)
rolled['rPrices']=(1+rolled['Returns']).cumprod()
```

$$S_t = \max\left\{0, S_{t-1} + y_t - \mathrm{E}_{t-1}\left[y_t\right]\right\}$$

Equation 26

$$S_t \geq h \Leftrightarrow \exists \tau \in [1, t] \left| \sum_{i=\tau}^{t} \left(y_i - \mathrm{E}_{i-1}\left[y_t\right]\right) \geq h \right.$$

Equation 27

$$S_t^+ = \max\left\{0, S_{t-1}^+ + y_t - \mathrm{E}_{t-1}\left[y_t\right]\right\},\ S_0^+ = 0$$

$$S_t^- = \min\left\{0, S_{t-1}^- + y_t - \mathrm{E}_{t-1}\left[y_t\right]\right\},\ S_0^- = 0$$

$$S_t = \max\left\{S_t^+, -S_t^-\right\}$$

Equation 28 and 29a & b

SNIPPET 2.4 THE SYMMETRIC CUSUM FILTER

```
def getTEvents(gRaw,h):
    tEvents,sPos,sNeg=[],0,0
    diff=gRaw.diff()
    for i in diff.index[1:]:
        sPos,sNeg=max(0,sPos+diff.loc[i]),min(0,sNeg+diff.loc[i])
        if sNeg<-h:
            sNeg=0;tEvents.append(i)
        elif sPos>h:
            sPos=0;tEvents.append(i)
    return pd.DatetimeIndex(tEvents)
```

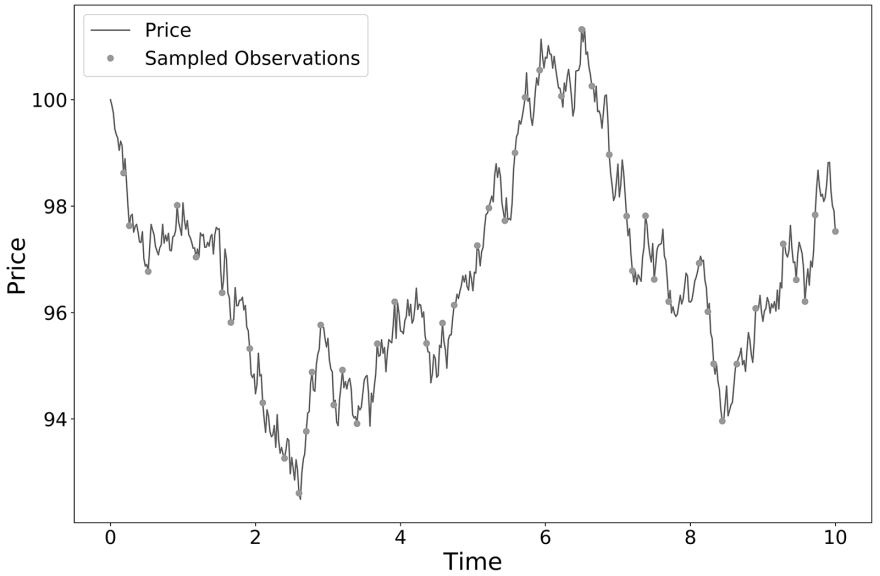


FIGURE 2.3 CUSUM sampling of a price series

$$y_i = \begin{cases} -1 & \text{if } r_{t_{i,0},t_{i,0}+h} < -\tau \\ 0 & \text{if } |r_{t_{i,0},t_{i,0}+h}| \leq \tau \\ 1 & \text{if } r_{t_{i,0},t_{i,0}+h} > \tau \end{cases}$$

Equation 30

$$r_{t_{i,0},t_{i,0}+h} = \frac{p_{t_{i,0}+h}}{p_{t_{i,0}}} - 1$$

Equation 31

SNIPPET 3.1 DAILY VOLATILITY ESTIMATES

```
def getDailyVol(close,span0=100):
    # daily vol, reindexed to close
    df0=close.index.searchsorted(close.index-pd.Timedelta(days=1))
    df0=df0[df0>0]
    df0=pd.Series(close.index[df0-1], index=close.index[close.shape[0]-df0.shape[0]:])
    df0=close.loc[df0.index]/close.loc[df0.values].values-1 # daily returns
    df0=df0.ewm(span=span0).std()
    return df0
```

SNIPPET 3.2 TRIPLE-BARRIER LABELING METHOD

```
def applyPtSlOnT1(close,events,ptSl,molecule):
    # apply stop loss/profit taking, if it takes place before t1 (end of event)
    events_=events.loc[molecule]
    out=events_[['t1']].copy(deep=True)
if ptSl[0]>0:pt=ptSl[0]*events_['trgt']
else:pt=pd.Series(index=events.index) # NaNs
if ptSl[1]>0:sl=-ptSl[1]*events_['trgt']
else:sl=pd.Series(index=events.index) # NaNs
for loc,t1 in events_['t1'].fillna(close.index[-1]).iteritems():
    df0=close[loc:t1] # path prices
    df0=(df0/close[loc]-1)*events_.at[loc,'side'] # path returns
    out.loc[loc,'sl']=df0[df0<sl[loc]].index.min() # earliest stop loss.
    out.loc[loc,'pt']=df0[df0>pt[loc]].index.min() # earliest profit taking.
return out
```

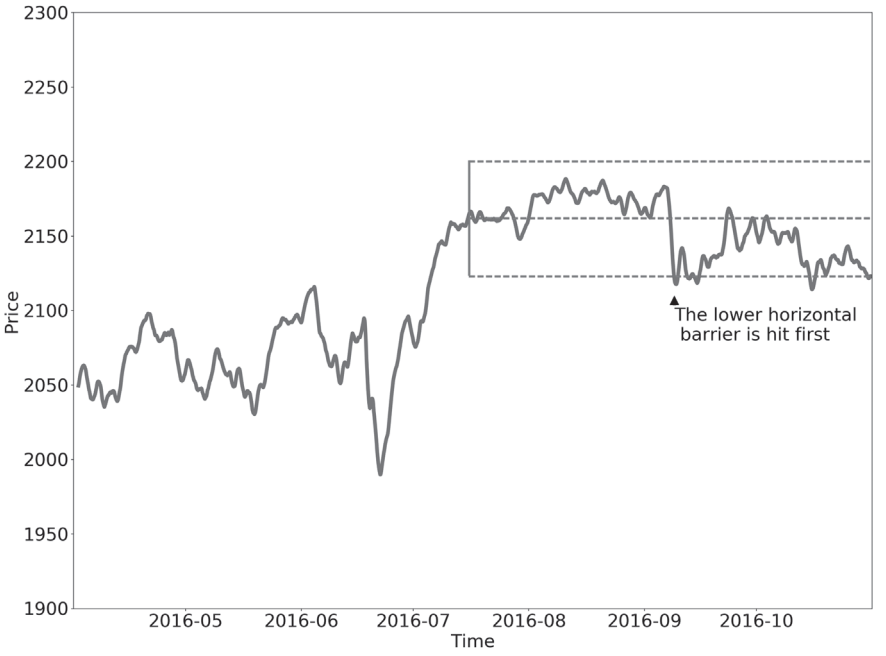


FIGURE 3.1 Two alternative configurations of the triple-barrier method

SNIPPET 3.3 GETTING THE TIME OF FIRST TOUCH

```
def getEvents(close,tEvents,ptSl,trgt,minRet,numThreads,t1=False):
    #1) get target
    trgt=trgt.loc[tEvents]
    trgt=trgt[trgt>minRet] # minRet
    #2) get t1 (max holding period)
    if t1 is False:t1=pd.Series(pd.NaT,index=tEvents)
    #3) form events object, apply stop loss on t1
    side_=pd.Series(1.,index=trgt.index)
    events=pd.concat({'t1':t1,'trgt':trgt,'side':side_}, \
        axis=1).dropna(subset=['trgt'])
    df0=mpPandasObj(func=applyPtSlOnT1,pdObj=('molecule',events.index), \
        numThreads=numThreads,close=close,events=events,ptSl=[ptSl,ptSl])
    events['t1']=df0.dropna(how='all').min(axis=1) # pd.min ignores nan
    events=events.drop('side',axis=1)
    return events
```

SNIPPET 3.4 ADDING A VERTICAL BARRIER

```
t1=close.index.searchsorted(tEvents+pd.Timedelta(days=numDays))
t1=t1[t1<close.shape[0]]
t1=pd.Series(close.index[t1],index=tEvents[:t1.shape[0]]) # NaNs at end
```

SNIPPET 3.5 LABELING FOR SIDE AND SIZE

```
def getBins(events,close):
    #1) prices aligned with events
    events_=events.dropna(subset=['t1'])
    px=events_.index.union(events_['t1'].values).drop_duplicates()
    px=close.reindex(px,method='bfill')
    #2) create out object
    out=pd.DataFrame(index=events_.index)
    out['ret']=px.loc[events_['t1'].values].values/px.loc[events_.index]-1
    out['bin']=np.sign(out['ret'])
    return out
```

SNIPPET 3.6 EXPANDING `getEvents` TO INCORPORATE META-LABELING

```
def getEvents(close,tEvents,ptSl,trgt,minRet,numThreads,t1=False,side=None):
    #1) get target
    trgt=trgt.loc[tEvents]
    trgt=trgt[trgt>minRet] # minRet
    #2) get t1 (max holding period)
    if t1 is False:t1=pd.Series(pd.NaT,index=tEvents)
    #3) form events object, apply stop loss on t1
    if side is None:side_,ptSl_=pd.Series(1.,index=trgt.index),[ptSl[0],ptSl[0]]
    else:side_,ptSl_=side.loc[trgt.index],ptSl[:2]
    events=pd.concat({'t1':t1,'trgt':trgt,'side':side_}, \
        axis=1).dropna(subset=['trgt'])
    df0=mpPandasObj(func=applyPtSlOnT1,pdObj=('molecule',events.index), \
        numThreads=numThreads,close=inst['Close'],events=events,ptSl=ptSl_)
        events['t1']=df0.dropna(how='all').min(axis=1) # pd.min ignores nan
        if side is None:events=events.drop('side',axis=1)
        return events
```

SNIPPET 3.7 EXPANDING `getBins` TO INCORPORATE META-LABELING

```
def getBins(events,close):
    '''
    Compute event's outcome (including side information, if provided).
    events is a DataFrame where:
    —events.index is event's starttime
    —events['t1'] is event's endtime
    —events['trgt'] is event's target
    —events['side'] (optional) implies the algo's position side
    Case 1: ('side' not in events): bin in (-1,1) <—label by price action
    Case 2: ('side' in events): bin in (0,1) <—label by pnl (meta-labeling)
    '''
    #1) prices aligned with events
    events_=events.dropna(subset=['t1'])
    px=events_.index.union(events_['t1'].values).drop_duplicates()
    px=close.reindex(px,method='bfill')
    #2) create out object
    out=pd.DataFrame(index=events_.index)
    out['ret']=px.loc[events_['t1'].values].values/px.loc[events_.index]-1
    if 'side' in events_:out['ret']*=events_['side'] # meta-labeling
    out['bin']=np.sign(out['ret'])
    if 'side' in events_:out.loc[out['ret']<=0,'bin']=0 # meta-labeling
    return out
```

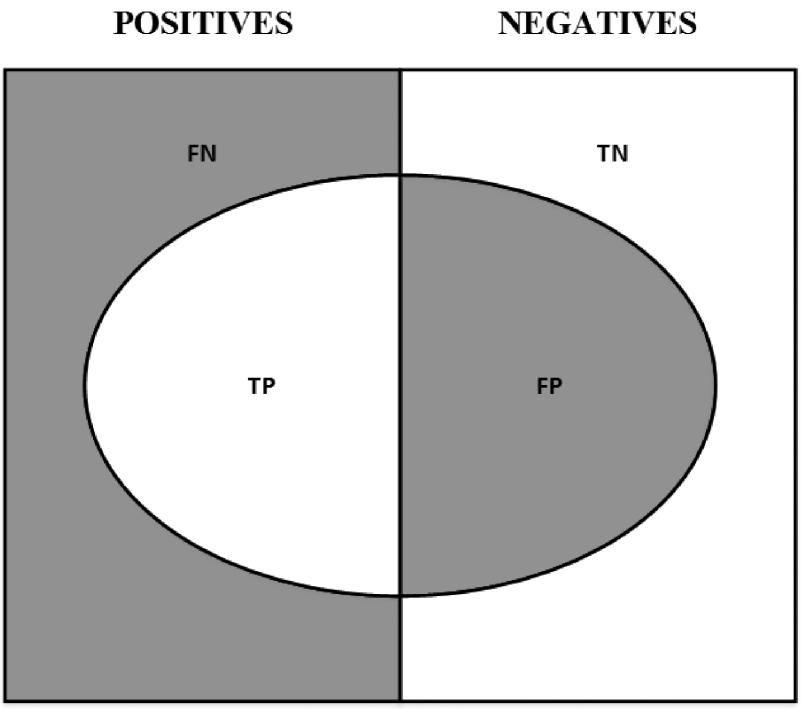


FIGURE 3.2 A visualization of the "confusion matrix"

SNIPPET 3.8 DROPPING UNDER-POPULATED LABELS

```
def dropLabels(events,minPtc=.05):
    # apply weights, drop labels with insufficient examples
    while True:
        df0=events['bin'].value_counts(normalize=True)
        if df0.min()>minPct or df0.shape[0]<3:break
        print 'dropped label',df0.argmin(),df0.min()
        events=events[events['bin']!=df0.argmin()]
    return events
```

SNIPPET 4.1 ESTIMATING THE UNIQUENESS OF A LABEL

```
def mpNumCoEvents(closeIdx,t1,molecule):
    '''
    Compute the number of concurrent events per bar.
    +molecule[0] is the date of the first event on which the weight will be computed
    +molecule[-1] is the date of the last event on which the weight will be computed
    Any event that starts before t1[molecule].max() impacts the count.
    '''
    #1) find events that span the period [molecule[0],molecule[-1]]
    t1=t1.fillna(closeIdx[-1]) # unclosed events still must impact other weights
    t1=t1[t1>=molecule[0]] # events that end at or after molecule[0]
    t1=t1.loc[:t1[molecule].max()] # events that start at or before t1[molecule].max()
    #2) count events spanning a bar
    iloc=closeIdx.searchsorted(np.array([t1.index[0],t1.max()]))
    count=pd.Series(0,index=closeIdx[iloc[0]:iloc[1]+1])
    for tIn,tOut in t1.iteritems():count.loc[tIn:tOut]+=1.
    return count.loc[molecule[0]:t1[molecule].max()]
```

$$\bar{u}_i = \left(\sum_{t=1}^{T} u_{t,i}\right)\left(\sum_{t=1}^{T} 1_{t,i}\right)^{-1}$$

Equation 32

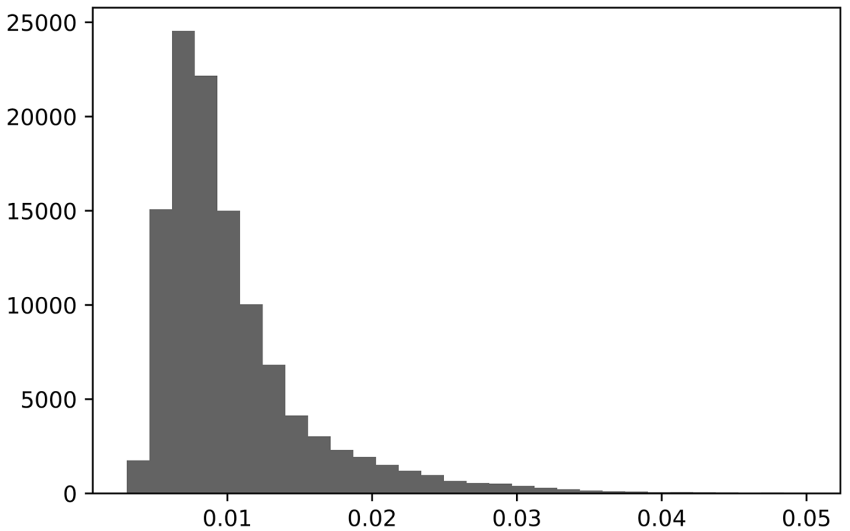


FIGURE 4.1 Histogram of uniqueness values

SNIPPET 4.2 ESTIMATING THE AVERAGE UNIQUENESS OF A LABEL

```
def mpSampleTW(t1,numCoEvents,molecule):
    # Derive average uniqueness over the event's lifespan
    wght=pd.Series(index=molecule)
    for tIn,tOut in t1.loc[wght.index].iteritems():
        wght.loc[tIn]=(1./numCoEvents.loc[tIn:tOut]).mean()
    return wght
#———————————————————————————————————
numCoEvents=mpPandasObj(mpNumCoEvents,('molecule',events.index),numThreads, \
        closeIdx=close.index,t1=events['t1'])
numCoEvents=numCoEvents.loc[~numCoEvents.index.duplicated(keep='last')]
numCoEvents=numCoEvents.reindex(close.index).fillna(0)
out['tW']=mpPandasObj(mpSampleTW,('molecule',events.index),numThreads, \
        t1=events['t1'],numCoEvents=numCoEvents)
```

$$\delta_j^{(2)} = \bar{u}_j^{(2)} \left(\sum_{k=1}^{I} \bar{u}_k^{(2)} \right)^{-1}$$

Equation 33

SNIPPET 4.3 BUILD AN INDICATOR MATRIX

```
import pandas as pd,numpy as np
#———————————————————————————————
def getIndMatrix(barIx,t1):
    # Get indicator matrix
    indM=pd.DataFrame(0,index=barIx,columns=range(t1.shape[0]))
    for i,(t0,t1) in enumerate(t1.iteritems()):indM.loc[t0:t1,i]=1.
    return indM
```

SNIPPET 4.4 COMPUTE AVERAGE UNIQUENESS

```
def getAvgUniqueness(indM):
    # Average uniqueness from indicator matrix
    c=indM.sum(axis=1) # concurrency
    u=indM.div(c,axis=0) # uniqueness
    avgU=u[u>0].mean() # average uniqueness
    return avgU
```

SNIPPET 4.5 RETURN SAMPLE FROM SEQUENTIAL BOOTSTRAP

```
def seqBootstrap(indM,sLength=None):
    # Generate a sample via sequential bootstrap
    if sLength is None:sLength=indM.shape[1]
    phi=[]
    while len(phi)<sLength:
        avgU=pd.Series()
        for i in indM:
            indM_=indM[phi+[i]] # reduce indM
            avgU.loc[i]=getAvgUniqueness(indM_).iloc[-1]
        prob=avgU/avgU.sum() # draw prob
        phi+=[np.random.choice(indM.columns,p=prob)]
    return phi
```

$$\{1_{t,i}\} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix 1

SNIPPET 4.6 EXAMPLE OF SEQUENTIAL BOOTSTRAP

```
def main():
    t1=pd.Series([2,3,5],index=[0,2,4]) # t0,t1 for each feature obs
    barIx=range(t1.max()+1) # index of bars
    indM=getIndMatrix(barIx,t1)
    phi=np.random.choice(indM.columns,size=indM.shape[1])
    print phi
    print 'Standard uniqueness:',getAvgUniqueness(indM[phi]).mean()
    phi=seqBootstrap(indM)
    print phi
    print 'Sequential uniqueness:',getAvgUniqueness(indM[phi]).mean()
    return
```

SNIPPET 4.7 GENERATING A RANDOM T1 SERIES

```
def getRndT1(numObs,numBars,maxH):
    # random t1 Series
    t1=pd.Series()
    for i in xrange(numObs):
        ix=np.random.randint(0,numBars)
        val=ix+np.random.randint(1,maxH)
        t1.loc[ix]=val
    return t1.sort_index()
```

SNIPPET 4.8 UNIQUENESS FROM STANDARD AND SEQUENTIAL BOOTSTRAPS

```
def auxMC(numObs,numBars,maxH):
    # Parallelized auxiliary function
    t1=getRndT1(numObs,numBars,maxH)
    barIx=range(t1.max()+1)
    indM=getIndMatrix(barIx,t1)
    phi=np.random.choice(indM.columns,size=indM.shape[1])
    stdU=getAvgUniqueness(indM[phi]).mean()
    phi=seqBootstrap(indM)
    seqU=getAvgUniqueness(indM[phi]).mean()
    return {'stdU':stdU,'seqU':seqU}
```

SNIPPET 4.9 MULTI-THREADED MONTE CARLO

```
import pandas as pd,numpy as np
from mpEngine import processJobs,processJobs_
#-------------------------------------
def mainMC(numObs=10,numBars=100,maxH=5,numIters=1E6,numThreads=24):
    # Monte Carlo experiments
    jobs=[]
    for i in xrange(int(numIters)):
        job={'func':auxMC,'numObs':numObs,'numBars':numBars,'maxH':maxH}
        jobs.append(job)
  if numThreads==1:out=processJobs_(jobs)
  else:out=processJobs(jobs,numThreads=numThreads)
  print pd.DataFrame(out).describe()
  return
```

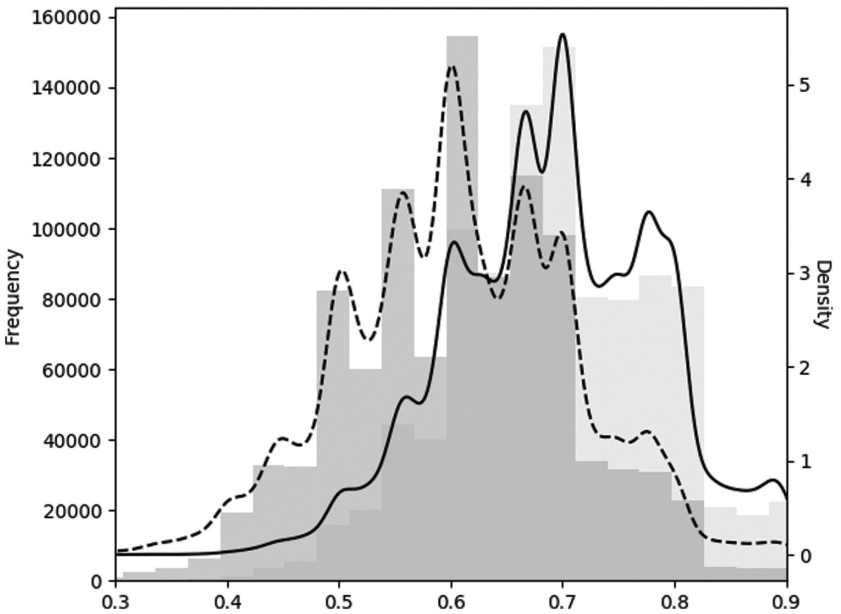


FIGURE 4.2 Monte Carlo experiment of standard vs. sequential bootstraps

When labels are a function of the return sign ($\{-1, 1\}$ for standard labeling or $\{0, 1\}$ for meta-labeling), the sample weights can be defined in terms of the sum of the attributed returns over the event's lifespan, $\left[t_{i,0}, t_{i,1}\right]$,

$$\tilde{w}_i = \left| \sum_{t=t_{i,0}}^{t_{i,1}} \frac{r_{t-1,t}}{c_t} \right|$$

$$w_i = \tilde{w}_i I \left(\sum_{j=1}^{I} \tilde{w}_j \right)^{-1}$$

Function 1

hence $\sum_{i=1}^{I} w_i = I$. We have scaled these weights to add up to I, since libraries (including sklearn) usually define algorithmic parameters assuming a default weight of 1.

SNIPPET 4.10 DETERMINATION OF SAMPLE WEIGHT BY ABSOLUTE RETURN ATTRIBUTION

```
def mpSampleW(t1,numCoEvents,close,molecule):
    # Derive sample weight by return attribution
    ret=np.log(close).diff() # log-returns, so that they are additive
    wght=pd.Series(index=molecule)
    for tIn,tOut in t1.loc[wght.index].iteritems():
        wght.loc[tIn]=(ret.loc[tIn:tOut]/numCoEvents.loc[tIn:tOut]).sum()
    return wght.abs()
#———————————————————————————————————
out['w']=mpPandasObj(mpSampleW,('molecule',events.index),numThreads, \
        t1=events['t1'],numCoEvents=numCoEvents,close=close)
out['w']*=out.shape[0]/out['w'].sum()
```

4.7 TIME DECAY

Markets are adaptive systems (Lo [2017]). As markets evolve, older examples are less relevant than the newer ones. Consequently, we would typically like sample weights to decay as new observations arrive. Let $d[x] \geq 0, \forall x \in \left[0, \sum_{i=1}^{I} \bar{u}_i\right]$ be the time-decay factors that will multiply the sample weights derived in the previous section. The final weight has no decay, $d\left[\sum_{i=1}^{I} \bar{u}_i\right] = 1$, and all other weights will be adjusted relative to that. Let $c \in (-1, 1]$ be a user-defined parameter that determines the decay function as follows: For $c \in [0, 1]$, then $d[1] = c$, with linear decay; for $c \in (-1, 0)$, then $d\left[-c\sum_{i=1}^{I} \bar{u}_i\right] = 0$, with linear decay between $\left[-c\sum_{i=1}^{I} \bar{u}_i, \sum_{i=1}^{I} \bar{u}_i\right]$ and $d[x] = 0 \; \forall x \leq -c\sum_{i=1}^{I} \bar{u}_i$. For a linear piecewise function $d = \max\{0, a + bx\}$, such requirements are met by the following boundary conditions:

1. $d = a + b\sum_{i=1}^{I} \bar{u}_i = 1 \Rightarrow a = 1 - b\sum_{i=1}^{I} \bar{u}_i$.
2. Contingent on c:
   (a) $d = a + b0 = c \Rightarrow b = (1 - c)\left(\sum_{i=1}^{I} \bar{u}_i\right)^{-1}, \forall c \in [0, 1]$
   (b) $d = a - bc\sum_{i=1}^{I} \bar{u}_i = 0 \Rightarrow b = \left[(c + 1)\sum_{i=1}^{I} \bar{u}_i\right]^{-1}, \forall c \in (-1, 0)$

Snippet 4.11 implements this form of time-decay factors. Note that time is not meant to be chronological. In this implementation, decay takes place according to cumulative uniqueness, $x \in \left[0, \sum_{i=1}^{I} \bar{u}_i\right]$, because a chronological decay would reduce weights too fast in the presence of redundant observations.

SNIPPET 4.11 IMPLEMENTATION OF TIME-DECAY FACTORS

```
def getTimeDecay(tW,clfLastW=1.):
    # apply piecewise-linear decay to observed uniqueness (tW)
    # newest observation gets weight=1, oldest observation gets weight=clfLastW
    clfW=tW.sort_index().cumsum()
    if clfLastW>=0:slope=(1.-clfLastW)/clfW.iloc[-1]
    else:slope=1./((clfLastW+1)*clfW.iloc[-1])
    const=1.-slope*clfW.iloc[-1]
    clfW=const+slope*clfW
    clfW[clfW<0]=0
    print const,slope
    return clfW
```

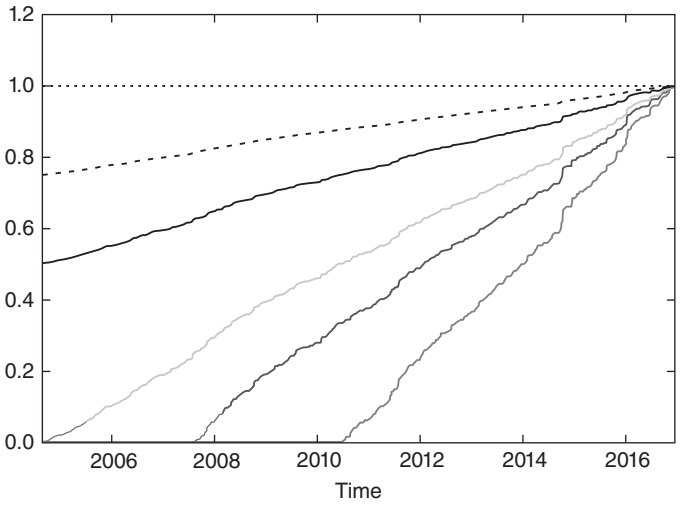


FIGURE 4.3 Piecewise-linear time-decay factors

5.4 THE METHOD

Consider the backshift operator, B, applied to a matrix of real-valued features $\{X_t\}$, where $B^k X_t = X_{t-k}$ for any integer $k \geq 0$. For example, $(1-B)^2 = 1 - 2B + B^2$, where $B^2 X_t = X_{t-2}$, so that $(1-B)^2 X_t = X_t - 2X_{t-1} + X_{t-2}$. Note that $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$, for n a positive integer. For a real number d, $(1+x)^d = \sum_{k=0}^{\infty} \binom{d}{k} x^k$, the binomial series.

In a fractional model, the exponent d is allowed to be a real number, with the following formal binomial series expansion:

$$\begin{aligned}(1-B)^d &= \sum_{k=0}^{\infty} \binom{d}{k} (-B)^k = \sum_{k=0}^{\infty} \frac{\prod_{i=0}^{k-1}(d-i)}{k!}(-B)^k \\ &= \sum_{k=0}^{\infty} (-B)^k \prod_{i=0}^{k-1} \frac{d-i}{k-i} \\ &= 1 - dB + \frac{d(d-1)}{2!}B^2 - \frac{d(d-1)(d-2)}{3!}B^3 + \cdots\end{aligned}$$

5.4.1 Long Memory

Let us see how a real (non-integer) positive d preserves memory. This arithmetic series consists of a dot product

$$\tilde{X}_t = \sum_{k=0}^{\infty} \omega_k X_{t-k}$$

with weights ω

$$\omega = \left\{1, -d, \frac{d(d-1)}{2!}, -\frac{d(d-1)(d-2)}{3!}, \ldots, (-1)^k \prod_{i=0}^{k-1} \frac{d-i}{k!}, \ldots\right\}$$

and values X

$$X = \left\{X_t, X_{t-1}, X_{t-2}, X_{t-3}, \ldots, X_{t-k}, \ldots\right\}$$

When d is a positive integer number, $\prod_{i=0}^{k-1} \frac{d-i}{k!} = 0, \forall k > d$, and memory beyond that point is cancelled. For example, $d = 1$ is used to compute returns, where $\prod_{i=0}^{k-1} \frac{d-i}{k!} = 0, \forall k > 1$, and $\omega = \{1, -1, 0, 0, \ldots\}$.

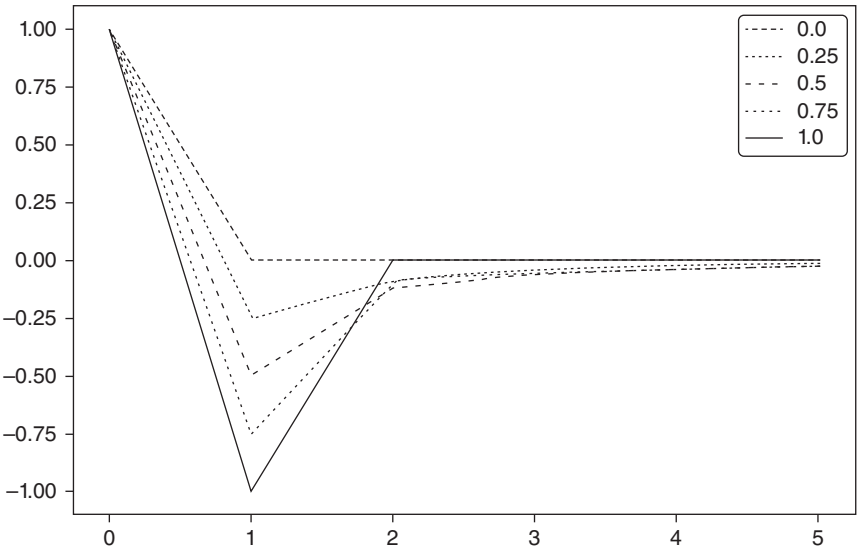


FIGURE 5.1 ω_k (y-axis) as k increases (x-axis). Each line is associated with a particular value of $d \in [0,1]$, in 0.1 increments.

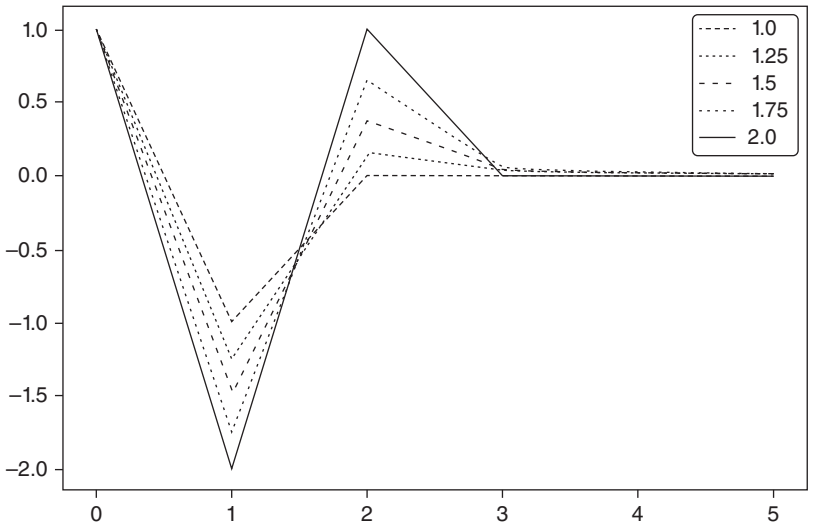


FIGURE 5.2 ω_k (y-axis) as k increases (x-axis). Each line is associated with a particular value of $d \in [1,2]$, in 0.1 increments.

SNIPPET 5.1 WEIGHTING FUNCTION

```
def getWeights(d,size):
    # thres>0 drops insignificant weights
    w=[1.]
    for k in range(1,size):
        w_=-w[-1]/k*(d-k+1)
        w.append(w_)
    w=np.array(w[::-1]).reshape(-1,1)
    return w
#---------------------------------------
def plotWeights(dRange,nPlots,size):
    w=pd.DataFrame()
    for d in np.linspace(dRange[0],dRange[1],nPlots):
        w_=getWeights(d,size=size)
        w_=pd.DataFrame(w_,index=range(w_.shape[0])[::-1],columns=[d])
        w=w.join(w_,how='outer')
    ax=w.plot()
    ax.legend(loc='upper left');mpl.show()
    return
#---------------------------------------
if __name__=='__main__':
    plotWeights(dRange=[0,1],nPlots=11,size=6)
    plotWeights(dRange=[1,2],nPlots=11,size=6)
```

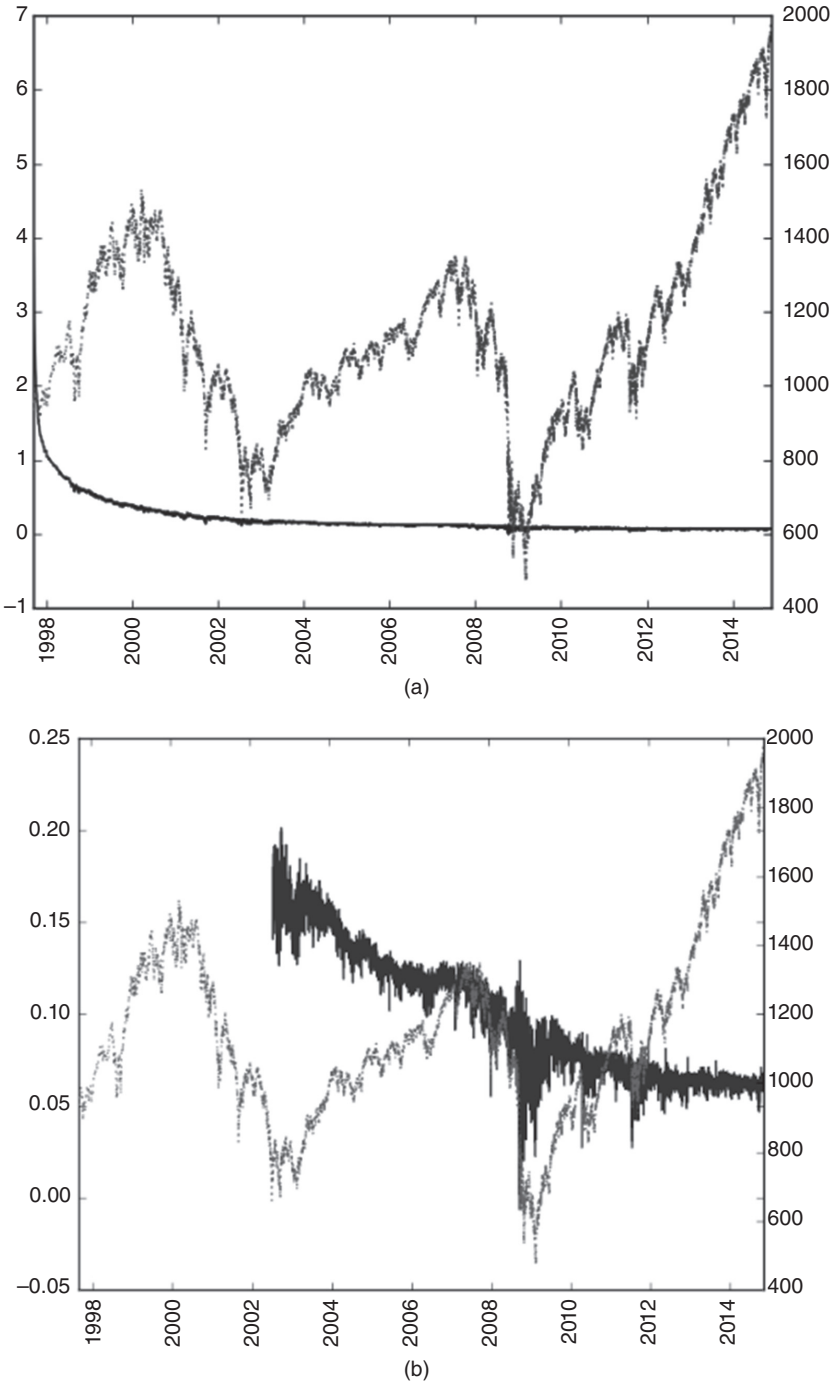


FIGURE 5.3 Fractional differentiation without controlling for weight loss (top plot) and after controlling for weight loss with an expanding window (bottom plot)

SNIPPET 5.2 STANDARD FRACDIFF (EXPANDING WINDOW)

```
def fracDiff(series,d,thres=.01):
    '''
    Increasing width window, with treatment of NaNs
    Note 1: For thres=1, nothing is skipped.
    Note 2: d can be any positive fractional, not necessarily bounded [0,1].
    '''
    #1) Compute weights for the longest series
    w=getWeights(d,series.shape[0])
    #2) Determine initial calcs to be skipped based on weight-loss threshold
    w_=np.cumsum(abs(w))
    w_/=w_[-1]
    skip=w_[w_>thres].shape[0]
    #3) Apply weights to values
    df={}
    for name in series.columns:
        seriesF,df_=series[[name]].fillna(method='ffill').dropna(),pd.Series()
        for iloc in range(skip,seriesF.shape[0]):
            loc=seriesF.index[iloc]
            if not np.isfinite(series.loc[loc,name]):continue # exclude NAs
            df_[loc]=np.dot(w[-(iloc+1):,:].T,seriesF.loc[:loc])[0,0]
        df[name]=df_.copy(deep=True)
    df=pd.concat(df,axis=1)
    return df
```

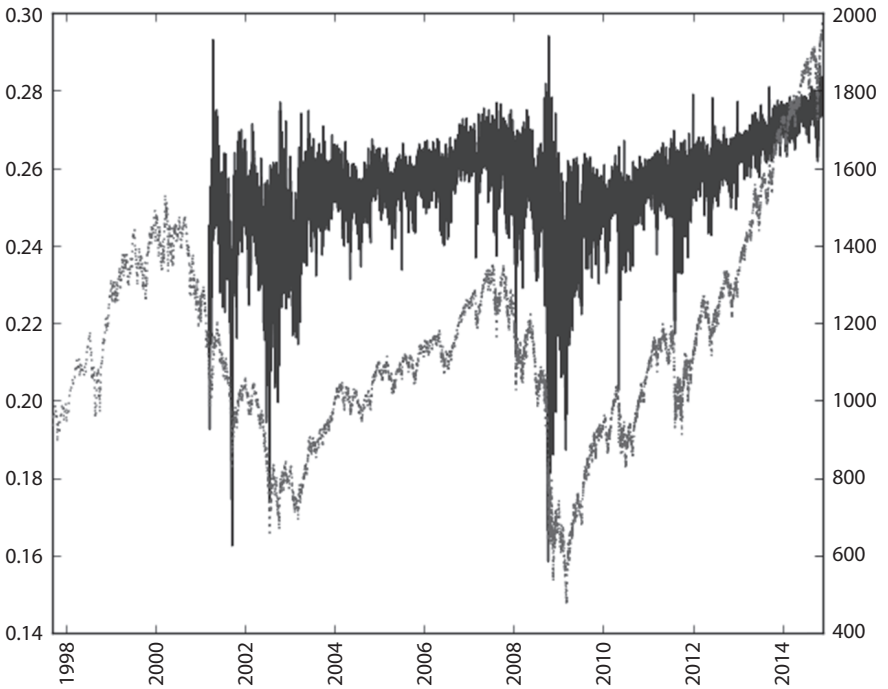


FIGURE 5.4 Fractional differentiation after controlling for weight loss with a fixed-width window

SNIPPET 5.3 THE NEW FIXED-WIDTH WINDOW FRACDIFF METHOD

```
def fracDiff_FFD(series,d,thres=1e-5):
    '''
    Constant width window (new solution)
    Note 1: thres determines the cut-off weight for the window
    Note 2: d can be any positive fractional, not necessarily bounded [0,1].
    '''
    #1) Compute weights for the longest series
    w=getWeights_FFD(d,thres)
    width=len(w)-1
    #2) Apply weights to values
    df={}
  for name in series.columns:
      seriesF,df_=series[[name]].fillna(method='ffill').dropna(),pd.Series()
      for iloc1 in range(width,seriesF.shape[0]):
          loc0,loc1=seriesF.index[iloc1-width],seriesF.index[iloc1]
          if not np.isfinite(series.loc[loc1,name]):continue # exclude NAs
          df_[loc1]=np.dot(w.T,seriesF.loc[loc0:loc1])[0,0]
      df[name]=df_.copy(deep=True)
  df=pd.concat(df,axis=1)
  return df
```

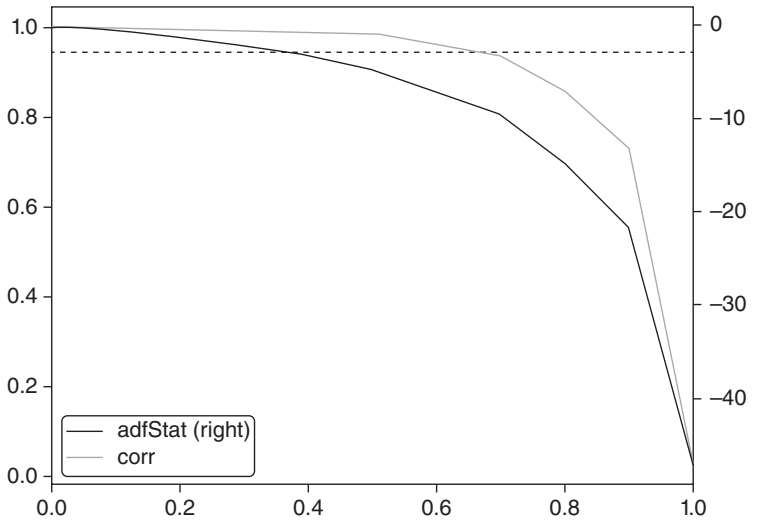


FIGURE 5.5 ADF statistic as a function of d, on E-mini S&P 500 futures log-prices

SNIPPET 5.4 FINDING THE MINIMUM *D* VALUE THAT PASSES THE ADF TEST

```
def plotMinFFD():
    from statsmodels.tsa.stattools import adfuller
    path,instName='./','ES1_Index_Method12'
    out=pd.DataFrame(columns=['adfStat','pVal','lags','nObs','95% conf','corr'])
    df0=pd.read_csv(path+instName+'.csv',index_col=0,parse_dates=True)
    for d in np.linspace(0,1,11):
        df1=np.log(df0[['Close']]).resample('1D').last() # downcast to daily obs
        df2=fracDiff_FFD(df1,d,thres=.01)
        corr=np.corrcoef(df1.loc[df2.index,'Close'],df2['Close'])[0,1]
        df2=adfuller(df2['Close'],maxlag=1,regression='c',autolag=None)
        out.loc[d]=list(df2[:4])+[df2[4]['5%']]+[corr] # with critical value
    out.to_csv(path+instName+'_testMinFFD.csv')
    out[['adfStat','corr']].plot(secondary_y='adfStat')
    mpl.axhline(out['95% conf'].mean(),linewidth=1,color='r',linestyle='dotted')
    mpl.savefig(path+instName+'_testMinFFD.png')
    return
```

TABLE 5.1 ADF Statistic on FFD(d) for Some of the Most Liquid Futures Contracts

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
AD1 Curncy	−1.7253	−1.8665	−2.2801	−2.9743	−3.9590	−5.4450	−7.7387	−10.3412	−15.7255	−22.5170	−43.8281
BO1 Comdty	−0.7039	−1.0021	−1.5848	−2.4038	−3.4284	−4.8916	−7.0604	−9.5089	−14.4065	−20.4393	−38.0683
BP1 Curncy	−1.0573	−1.4963	−2.3223	−3.4641	−4.8976	−6.9157	−9.8833	−13.1575	−19.4238	−26.6320	−43.3284
BTS1 Comdty	−1.7987	−2.1428	−2.7600	−3.7019	−4.8522	−6.2412	−7.8115	−9.4645	−11.0334	−12.4470	−13.6410
BZ1 Index	−1.6569	−1.8766	−2.3948	−3.2145	−4.2821	−5.9431	−8.3329	−10.9046	−15.7006	−20.7224	−29.9510
C 1 Comdty	−1.7870	−2.1273	−2.9539	−4.1642	−5.7307	−7.9577	−11.1798	−14.6946	−20.9925	−27.6602	−39.3576
CC1 Comdty	−2.3743	−2.9503	−4.1694	−5.8997	−8.0868	−10.9871	−14.8206	−18.6154	−24.1738	−29.0285	−34.8580
CD1 Curncy	−1.6304	−2.0557	−2.7284	−3.8380	−5.2341	−7.3172	−10.3738	−13.8263	−20.2897	−27.6242	−43.6794
CF1 Index	−1.5539	−1.9387	−2.7421	−3.9235	−5.5085	−7.7585	−11.0571	−14.6829	−21.4877	−28.9810	−44.5059
CL1 Comdty	−0.3795	−0.7164	−1.3359	−2.2018	−3.2603	−4.7499	−6.9504	−9.4531	−14.4936	−20.8392	−41.1169
CN1 Comdty	−0.8798	−0.8711	−1.1020	−1.4626	−1.9732	−2.7508	−3.9217	−5.2944	−8.4257	−12.7300	−42.1411
CO1 Comdty	−0.5124	−0.8468	−1.4247	−2.2402	−3.2566	−4.7022	−6.8601	−9.2836	−14.1511	−20.2313	−39.2207
CT1 Comdty	−1.7604	−2.0728	−2.7529	−3.7853	−5.1397	−7.1123	−10.0137	−13.1851	−19.0603	−25.4513	−37.5703
DM1 Index	−0.1929	−0.5718	−1.2414	−2.1127	−3.1765	−4.6695	−6.8852	−9.4219	−14.6726	−21.5411	−49.2663
DU1 Comdty	−0.3365	−0.4572	−0.7647	−1.1447	−1.6132	−2.2759	−3.3389	−4.5689	−7.2101	−10.9025	−42.9012
DX1 Curncy	−1.5768	−1.9458	−2.7358	−3.8423	−5.3101	−7.3507	−10.3569	−13.6451	−19.5832	−25.8907	−37.2623
EC1 Comdty	−0.2727	−0.6650	−1.3359	−2.2112	−3.3112	−4.8320	−7.0777	−9.6299	−14.8258	−21.4634	−44.6452
EC1 Curncy	−1.4733	−1.9344	−2.8507	−4.1588	−5.8240	−8.1834	−11.6278	−15.4095	−22.4317	−30.1482	−45.6373
ED1 Comdty	−0.4084	−0.5350	−0.7948	−1.1772	−1.6633	−2.3818	−3.4601	−4.7041	−7.4373	−11.3175	−46.4487
EE1 Curncy	−1.2100	−1.6378	−2.4216	−3.5470	−4.9821	−7.0166	−9.9962	−13.2920	−19.5047	−26.5158	−41.4672
EO1 Comdty	−0.7903	−0.8917	−1.0551	−1.3465	−1.7302	−2.3500	−3.3068	−4.5136	−7.0157	−10.6463	−45.2100

EO1 Index	−0.6561	−1.0567	−1.7409	−2.6774	−3.8543	−5.5096	−7.9133	−10.5674	−15.6442	−21.3066	−35.1397
ER1 Comdty	−0.1970	−0.3442	−0.6334	−1.0363	−1.5327	−2.2378	−3.2819	−4.4647	−7.1031	−10.7389	−40.0407
ES1 Index	−0.3387	−0.7206	−1.3324	−2.2252	−3.2733	−4.7976	−7.0436	−9.6095	−14.8624	−21.6177	−46.9114
FA1 Index	−0.5292	−0.8526	−1.4250	−2.2359	−3.2500	−4.6902	−6.8272	−9.2410	−14.1664	−20.3733	−41.9705
FC1 Comdty	−1.8846	−2.1853	−2.8808	−3.8546	−5.1483	−7.0226	−9.6889	−12.5679	−17.8160	−23.0530	−31.6503
FV1 Comdty	−0.7257	−0.8515	−1.0596	−1.4304	−1.8312	−2.5302	−3.6296	−4.9499	−7.8292	−12.0467	−49.1508
G 1 Comdty	0.2326	0.0026	−0.4686	−1.0590	−1.7453	−2.6761	−4.0336	−5.5624	−8.8575	−13.3277	−42.9177
GC1 Comdty	−2.2221	−2.3544	−2.7467	−3.4140	−4.4861	−6.0632	−8.4803	−11.2152	−16.7111	−23.1750	−39.0715
GX1 Index	−1.5418	−1.7749	−2.4666	−3.4417	−4.7321	−6.6155	−9.3667	−12.5240	−18.6291	−25.8116	−43.3610
HG1 Comdty	−1.7372	−2.1495	−2.8323	−3.9090	−5.3257	−7.3805	−10.4121	−13.7669	−19.8902	−26.5819	−39.3267
HI1 Index	−1.8289	−2.0432	−2.6203	−3.5233	−4.7514	−6.5743	−9.2733	−12.3722	−18.5308	−25.9762	−45.3396
HO1 Comdty	−1.6024	−1.9941	−2.6619	−3.7131	−5.1772	−7.2468	−10.3326	−13.6745	−19.9728	−26.9772	−40.9824
IB1 Index	−2.3912	−2.8254	−3.5813	−4.8774	−6.5884	−9.0665	−12.7381	−16.6706	−23.6752	−30.7986	−43.0687
IK1 Comdty	−1.7373	−2.3000	−2.7764	−3.7101	−4.8686	−6.3504	−8.2195	−9.8636	−11.7882	−13.3983	−14.8391
IR1 Comdty	−2.0622	−2.4188	−3.1736	−4.3178	−5.8119	−7.9816	−11.2102	−14.7956	−21.6158	−29.4555	−46.2683
JA1 Comdty	−2.4701	−2.7292	−3.3925	−4.4658	−5.9236	−8.0270	−11.2082	−14.7198	−21.2681	−28.4380	−42.1937
JB1 Comdty	−0.2081	−0.4319	−0.8490	−1.4289	−2.1160	−3.0932	−4.5740	−6.3061	−9.9454	−15.0151	−47.6037
JE1 Curncy	−0.9268	−1.2078	−1.7565	−2.5398	−3.5545	−5.0270	−7.2096	−9.6808	−14.6271	−20.7168	−37.6954
JG1 Comdty	−1.7468	−1.8071	−2.0654	−2.5447	−3.2237	−4.3418	−6.0690	−8.0537	−12.3908	−18.1881	−44.2884
JO1 Comdty	−3.0052	−3.3099	−4.2639	−5.7291	−7.5686	−10.1683	−13.7068	−17.3054	−22.7853	−27.7011	−33.4658
JY1 Curncy	−1.2616	−1.5891	−2.2042	−3.1407	−4.3715	−6.1600	−8.8261	−11.8449	−17.8275	−25.0700	−44.8394
KC1 Comdty	−0.7786	−1.1172	−1.7723	−2.7185	−3.8875	−5.5651	−8.0217	−10.7422	−15.9423	−21.8651	−35.3354
L 1 Comdty	−0.0805	−0.2228	−0.6144	−1.0751	−1.6335	−2.4186	−3.5676	−4.8749	−7.7528	−11.7669	−44.0349

At a 95% confidence level, the ADF test's critical value is −2.8623. All of the log-price series achieve stationarity at $d < 0.6$, and the great majority are stationary at $d < 0.3$.

Consider a training set of observations $\{x_i\}_{i=1,\ldots,n}$ and real-valued outcomes $\{y_i\}_{i=1,\ldots,n}$. Suppose a function $f[x]$ exists, such that $y = f[x] + \varepsilon$, where ε is white noise with $\mathrm{E}[\varepsilon_i] = 0$ and $\mathrm{E}[\varepsilon_i^2] = \sigma_\varepsilon^2$. We would like to estimate the function $\hat{f}[x]$ that best fits $f[x]$, in the sense of making the variance of the estimation error $\mathrm{E}[(y_i - \hat{f}[x_i])^2]$ minimal (the mean squared error cannot be zero, because of the noise represented by σ_ε^2). This mean-squared error can be decomposed as

$$\mathrm{E}\left[(y_i - \hat{f}[x_i])^2\right] = \left(\underbrace{\mathrm{E}[\hat{f}[x_i] - f[x_i]]}_{bias}\right)^2 + \underbrace{\mathrm{V}[\hat{f}[x_i]]}_{variance} + \underbrace{\sigma_\varepsilon^2}_{noise}$$

Equation 34

An ensemble method is a method that combines a set of weak learners, all based on the same learning algorithm, in order to create a (stronger) learner that performs better than any of the individual ones. Ensemble methods help reduce bias and/or variance.

6.3.1 Variance Reduction

Bagging's main advantage is that it reduces forecasts' variance, hence helping address overfitting. The variance of the bagged prediction ($\varphi_i[c]$) is a function of the number of bagged estimators (N), the average variance of a single estimator's prediction ($\bar{\sigma}$), and the average correlation among their forecasts ($\bar{\rho}$):

$$\begin{aligned}
\mathrm{V}\left[\frac{1}{N}\sum_{i=1}^{N}\varphi_i[c]\right] &= \frac{1}{N^2}\sum_{i=1}^{N}\left(\sum_{j=1}^{N}\sigma_{i,j}\right) = \frac{1}{N^2}\sum_{i=1}^{N}\left(\sigma_i^2 + \sum_{j\neq i}^{N}\sigma_i\sigma_j\rho_{i,j}\right) \\
&= \frac{1}{N^2}\sum_{i=1}^{N}\left(\bar{\sigma}^2 + \underbrace{\sum_{j\neq i}^{N}\bar{\sigma}^2\bar{\rho}}_{\substack{=(N-1)\bar{\sigma}^2\bar{\rho} \\ \text{for a fixed } i}}\right) = \frac{\bar{\sigma}^2 + (N-1)\bar{\sigma}^2\bar{\rho}}{N} \\
&= \bar{\sigma}^2\left(\bar{\rho} + \frac{1-\bar{\rho}}{N}\right)
\end{aligned}$$

where $\sigma_{i,j}$ is the covariance of predictions by estimators i,j; $\sum_{i=1}^{N}\bar{\sigma}^2 = \sum_{i=1}^{N}\sigma_i^2 \Leftrightarrow \bar{\sigma}^2 = N^{-1}\sum_{i=1}^{N}\sigma_i^2$; and $\sum_{j\neq i}^{N}\bar{\sigma}^2\bar{\rho} = \sum_{j\neq i}^{N}\sigma_i\sigma_j\rho_{i,j} \Leftrightarrow \bar{\rho} = (\bar{\sigma}^2 N(N-1))^{-1}\sum_{j\neq i}^{N}\sigma_i\sigma_j\rho_{i,j}$.

The equation above shows that bagging is only effective to the extent that $\bar{\rho} < 1$; as $\bar{\rho} \to 1 \Rightarrow \mathrm{V}[\frac{1}{N}\sum_{i=1}^{N}\varphi_i[c]] \to \bar{\sigma}^2$. One of the goals of sequential bootstrapping (Chapter 4) is to produce samples as independent as possible, thereby reducing $\bar{\rho}$, which should lower the variance of bagging classifiers. Figure 6.1 plots the standard deviation of the bagged prediction as a function of $N \in [5, 30]$, $\bar{\rho} \in [0, 1]$ and $\bar{\sigma} = 1$.

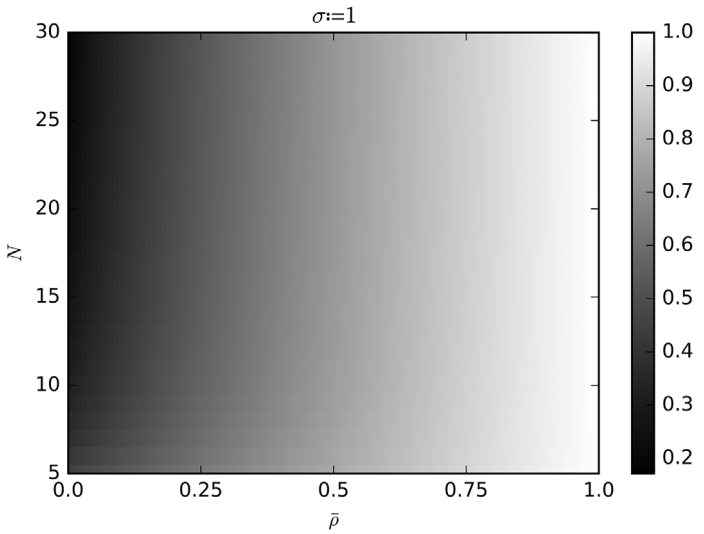


FIGURE 6.1 Standard deviation of the bagged prediction

$$\mathrm{P}\left[X > \frac{N}{k}\right] = 1 - \mathrm{P}\left[X \leq \frac{N}{k}\right] = 1 - \sum_{i=0}^{\left\lfloor N/k \right\rfloor} \binom{N}{i} p^i (1-p)^{N-i}$$

Equation 35

SNIPPET 6.1 ACCURACY OF THE BAGGING CLASSIFIER

```
from scipy.misc import comb
N,p,k=100,1./3,3.
p_=0
for i in xrange(0,int(N/k)+1):
    p_+=comb(N,i)*p**i*(1-p)**(N-i)
print p,1-p_
```

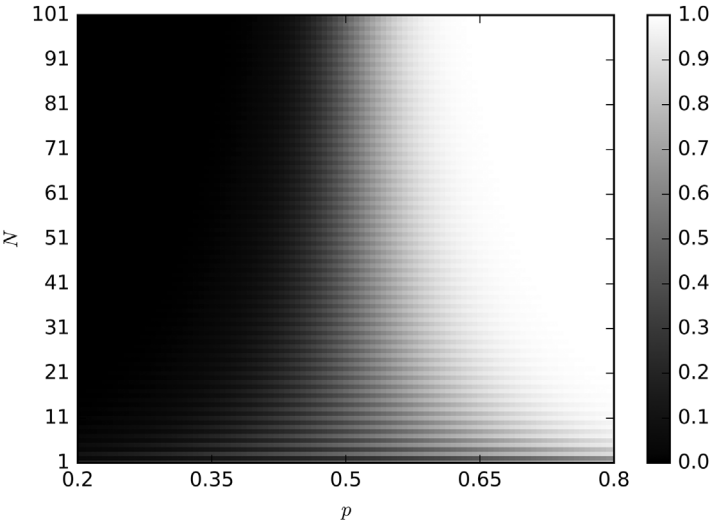


FIGURE 6.2 Accuracy of a bagging classifier as a function of the individual estimator's accuracy (P), the number of estimators (N), and $k = 2$

SNIPPET 6.2 THREE WAYS OF SETTING UP AN RF

```
clf0=RandomForestClassifier(n_estimators=1000,class_weight='balanced_subsample',
    criterion='entropy')
clf1=DecisionTreeClassifier(criterion='entropy',max_features='auto',
    class_weight='balanced')
clf1=BaggingClassifier(base_estimator=clf1,n_estimators=1000,max_samples=avgU)
clf2=RandomForestClassifier(n_estimators=1,criterion='entropy',bootstrap=False,
    class_weight='balanced_subsample')
clf2=BaggingClassifier(base_estimator=clf2,n_estimators=1000,max_samples=avgU,
    max_features=1.)
```

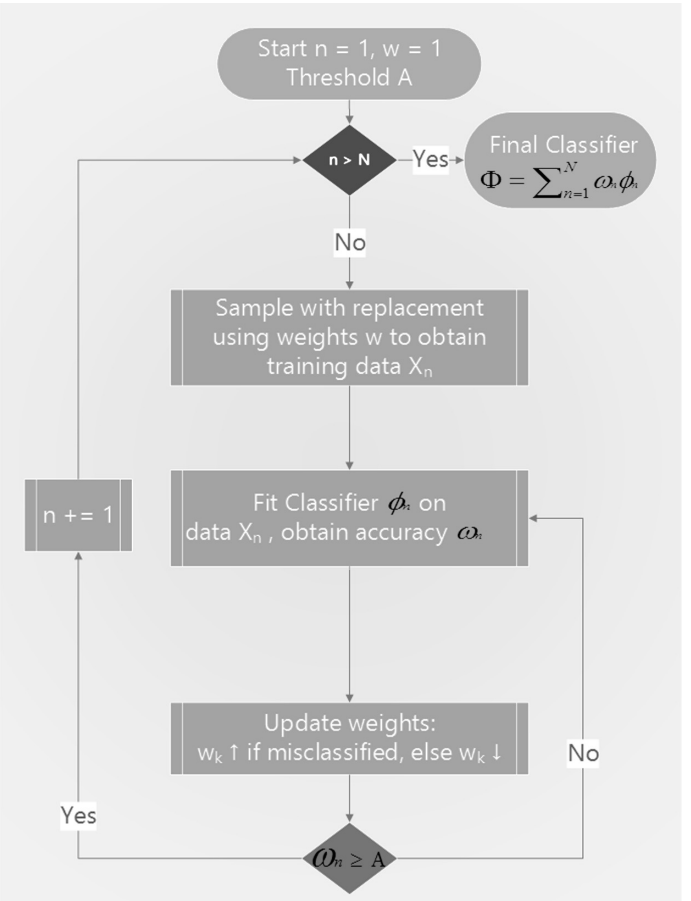


FIGURE 6.3 AdaBoost decision flow

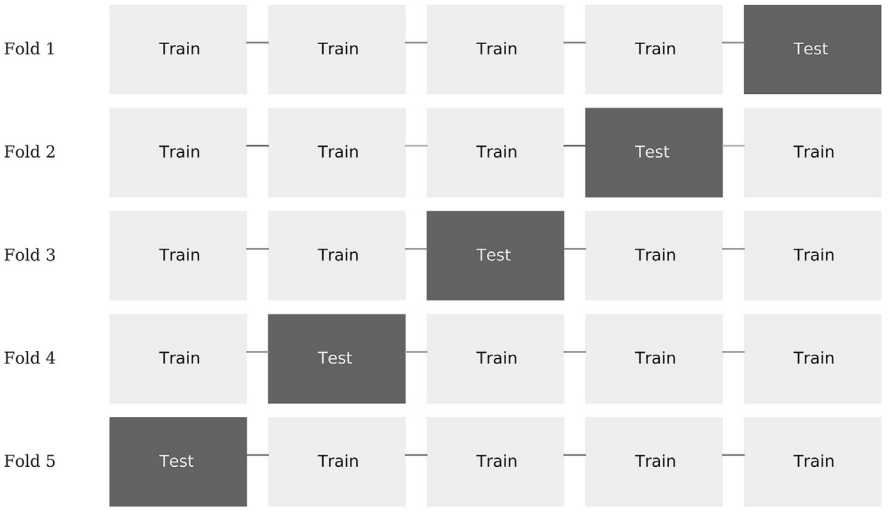


FIGURE 7.1 Train/test splits in a 5-fold CV scheme

SNIPPET 7.1 PURGING OBSERVATION IN THE TRAINING SET

```
def getTrainTimes(t1,testTimes):
    '''
    Given testTimes, find the times of the training observations.
    —t1.index: Time when the observation started.
    —t1.value: Time when the observation ended.
    —testTimes: Times of testing observations.
    '''
    trn=t1.copy(deep=True)
    for i,j in testTimes.iteritems():
        df0=trn[(i<=trn.index)&(trn.index<=j)].index # train starts within test
        df1=trn[(i<=trn)&(trn<=j)].index # train ends within test
        df2=trn[(trn.index<=i)&(j<=trn)].index # train envelops test
        trn=trn.drop(df0.union(df1).union(df2))
    return trn
```

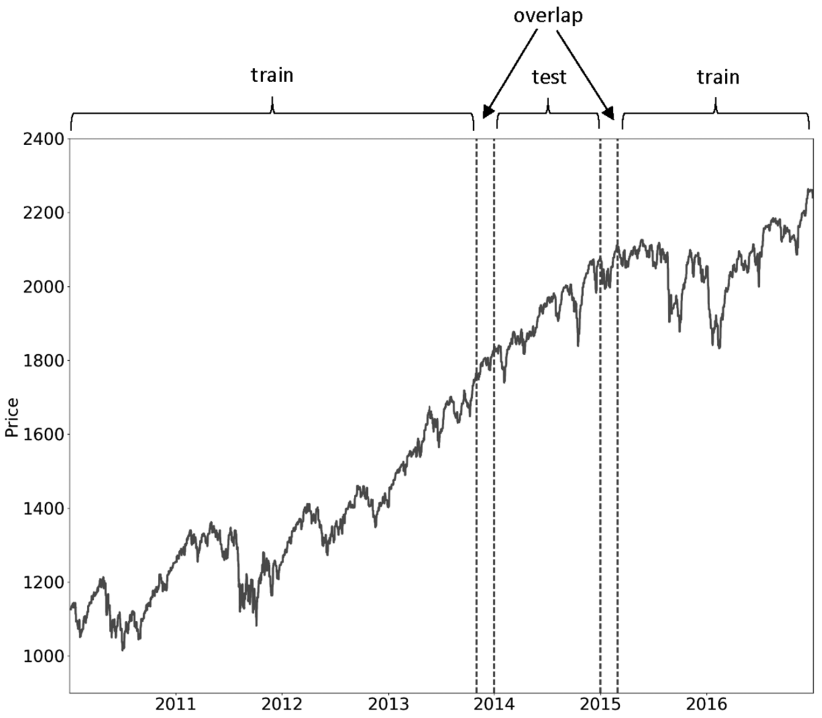


FIGURE 7.2 Purging overlap in the training set

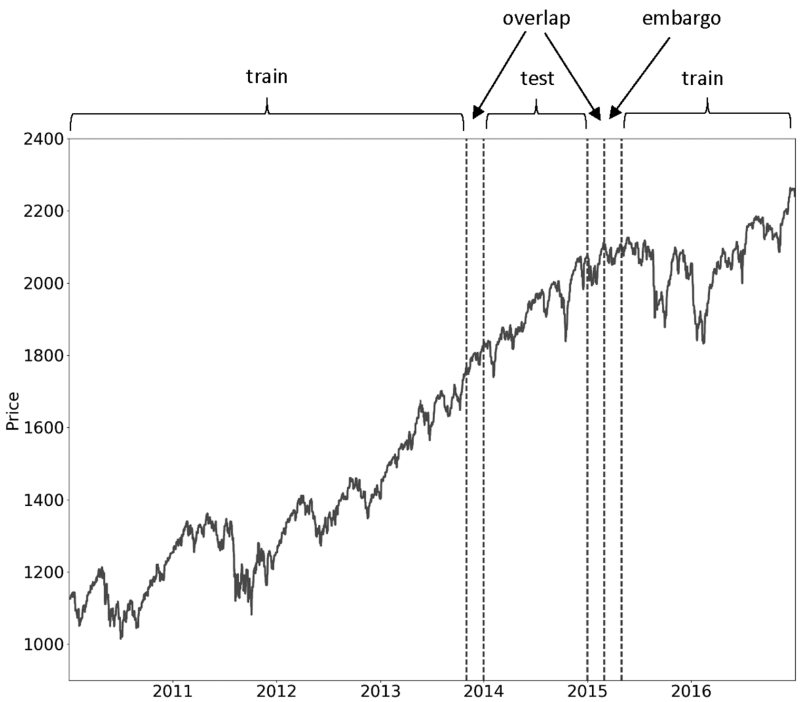


FIGURE 7.3 Embargo of post-test train observations

SNIPPET 7.2 EMBARGO ON TRAINING OBSERVATIONS

```
def getEmbargoTimes(times,pctEmbargo):
    # Get embargo time for each bar
    step=int(times.shape[0]*pctEmbargo)
    if step==0:
        mbrg=pd.Series(times,index=times)
    else:
        mbrg=pd.Series(times[step:],index=times[:-step])
        mbrg=mbrg.append(pd.Series(times[-1],index=times[-step:]))
    return mbrg
#———————————————————————————————————
testTimes=pd.Series(mbrg[dt1],index=[dt0]) # include embargo before purge
trainTimes=getTrainTimes(t1,testTimes)
testTimes=t1.loc[dt0:dt1].index
```

SNIPPET 7.3 CROSS-VALIDATION CLASS WHEN OBSERVATIONS OVERLAP

```
class PurgedKFold(_BaseKFold):
    '''
    Extend KFold class to work with labels that span intervals
    The train is purged of observations overlapping test-label intervals
    Test set is assumed contiguous (shuffle=False), w/o training samples in between
    '''
    def __init__(self,n_splits=3,t1=None,pctEmbargo=0.):
        if not isinstance(t1,pd.Series):
            raise ValueError('Label Through Dates must be a pd.Series')
        super(PurgedKFold,self).__init__(n_splits,shuffle=False,random_state=None)
        self.t1=t1
        self.pctEmbargo=pctEmbargo
    def split(self,X,y=None,groups=None):
        if (X.index==self.t1.index).sum()!=len(self.t1):
            raise ValueError('X and ThruDateValues must have the same index')
        indices=np.arange(X.shape[0])
        mbrg=int(X.shape[0]*self.pctEmbargo)
        test_starts=[(i[0],i[-1]+1) for i in \
            np.array_split(np.arange(X.shape[0]),self.n_splits)]
        for i,j in test_starts:
            t0=self.t1.index[i] # start of test set
            test_indices=indices[i:j]
            maxT1Idx=self.t1.index.searchsorted(self.t1[test_indices].max())
            train_indices=self.t1.index.searchsorted(self.t1[self.t1<=t0].index)
            if maxT1Idx<X.shape[0]: # right train (with embargo)
                train_indices=np.concatenate((train_indices,indices[maxT1Idx+mbrg:]))
            yield train_indices,test_indices
```

SNIPPET 7.4 USING THE `PurgedKFold` CLASS

```
def cvScore(clf,X,y,sample_weight,scoring='neg_log_loss',t1=None,cv=None,cvGen=None,
            pctEmbargo=None):
    if scoring not in ['neg_log_loss','accuracy']:
        raise Exception('wrong scoring method.')
    from sklearn.metrics import log_loss,accuracy_score
    from clfSequential import PurgedKFold
    if cvGen is None:
        cvGen=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged
    score=[]
    for train,test in cvGen.split(X=X):
        fit=clf.fit(X=X.iloc[train,:],y=y.iloc[train],
                    sample_weight=sample_weight.iloc[train].values)
        if scoring=='neg_log_loss':
            prob=fit.predict_proba(X.iloc[test,:])
            score_=-log_loss(y.iloc[test],prob,
                sample_weight=sample_weight.iloc[test].values,labels=clf.classes_)
        else:
            pred=fit.predict(X.iloc[test,:])
            score_=accuracy_score(y.iloc[test],pred,sample_weight= \
                sample_weight.iloc[test].values)
        score.append(score_)
    return np.array(score)
```

SNIPPET 8.2 MDI FEATURE IMPORTANCE

```
def featImpMDI(fit,featNames):
    # feat importance based on IS mean impurity reduction
    df0={i:tree.feature_importances_ for i,tree in enumerate(fit.estimators_)}
    df0=pd.DataFrame.from_dict(df0,orient='index')
    df0.columns=featNames
    df0=df0.replace(0,np.nan) # because max_features=1
  imp=pd.concat({'mean':df0.mean(),'std':df0.std()*df0.shape[0]**-.5},axis=1)
  imp/=imp['mean'].sum()
  return imp
```

SNIPPET 8.3 MDA FEATURE IMPORTANCE

```
def featImpMDA(clf,X,y,cv,sample_weight,t1,pctEmbargo,scoring='neg_log_loss'):
    # feat importance based on OOS score reduction
    if scoring not in ['neg_log_loss','accuracy']:
        raise Exception('wrong scoring method.')
    from sklearn.metrics import log_loss,accuracy_score
    cvGen=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged cv
    scr0,scr1=pd.Series(),pd.DataFrame(columns=X.columns)
    for i,(train,test) in enumerate(cvGen.split(X=X)):
        X0,y0,w0=X.iloc[train,:],y.iloc[train],sample_weight.iloc[train]
        X1,y1,w1=X.iloc[test,:],y.iloc[test],sample_weight.iloc[test]
        fit=clf.fit(X=X0,y=y0,sample_weight=w0.values)
        if scoring=='neg_log_loss':
            prob=fit.predict_proba(X1)
            scr0.loc[i]=-log_loss(y1,prob,sample_weight=w1.values,
                                  labels=clf.classes_)
        else:
            pred=fit.predict(X1)
            scr0.loc[i]=accuracy_score(y1,pred,sample_weight=w1.values)
        for j in X.columns:
            X1_=X1.copy(deep=True)
            np.random.shuffle(X1_[j].values) # permutation of a single column
            if scoring=='neg_log_loss':
                prob=fit.predict_proba(X1_)
                scr1.loc[i,j]=-log_loss(y1,prob,sample_weight=w1.values,
                                        labels=clf.classes_)
            else:
                pred=fit.predict(X1_)
                scr1.loc[i,j]=accuracy_score(y1,pred,sample_weight=w1.values)
    imp=(-scr1).add(scr0,axis=0)
    if scoring=='neg_log_loss':imp=imp/-scr1
    else:imp=imp/(1.-scr1)
    imp=pd.concat({'mean':imp.mean(),'std':imp.std()*imp.shape[0]**-.5},axis=1)
    return imp,scr0.mean()
```

SNIPPET 8.4 IMPLEMENTATION OF SFI

```
def auxFeatImpSFI(featNames,clf,trnsX,cont,scoring,cvGen):
    imp=pd.DataFrame(columns=['mean','std'])
    for featName in featNames:
        df0=cvScore(clf,X=trnsX[[featName]],y=cont['bin'],sample_weight=cont['w'],
                    scoring=scoring,cvGen=cvGen)
        imp.loc[featName,'mean']=df0.mean()
        imp.loc[featName,'std']=df0.std()*df0.shape[0]**-.5
    return imp
```

SNIPPET 8.5 COMPUTATION OF ORTHOGONAL FEATURES

```
def get_eVec(dot,varThres):
    # compute eVec from dot prod matrix, reduce dimension
    eVal,eVec=np.linalg.eigh(dot)
    idx=eVal.argsort()[::-1] # arguments for sorting eVal desc
    eVal,eVec=eVal[idx],eVec[:,idx]
    #2) only positive eVals
    eVal=pd.Series(eVal,index=['PC_'+str(i+1) for i in range(eVal.shape[0])])
    eVec=pd.DataFrame(eVec,index=dot.index,columns=eVal.index)
    eVec=eVec.loc[:,eVal.index]
    #3) reduce dimension, form PCs
    cumVar=eVal.cumsum()/eVal.sum()
    dim=cumVar.values.searchsorted(varThres)
    eVal,eVec=eVal.iloc[:dim+1],eVec.iloc[:,:dim+1]
    return eVal,eVec
#---------------------------------------------------------------
def orthoFeats(dfX,varThres=.95):
    # Given a dataframe dfX of features, compute orthofeatures dfP
    dfZ=dfX.sub(dfX.mean(),axis=1).div(dfX.std(),axis=1) # standardize
    dot=pd.DataFrame(np.dot(dfZ.T,dfZ),index=dfX.columns,columns=dfX.columns)
    eVal,eVec=get_eVec(dot,varThres)
    dfP=np.dot(dfZ,eVec)
    return dfP
```

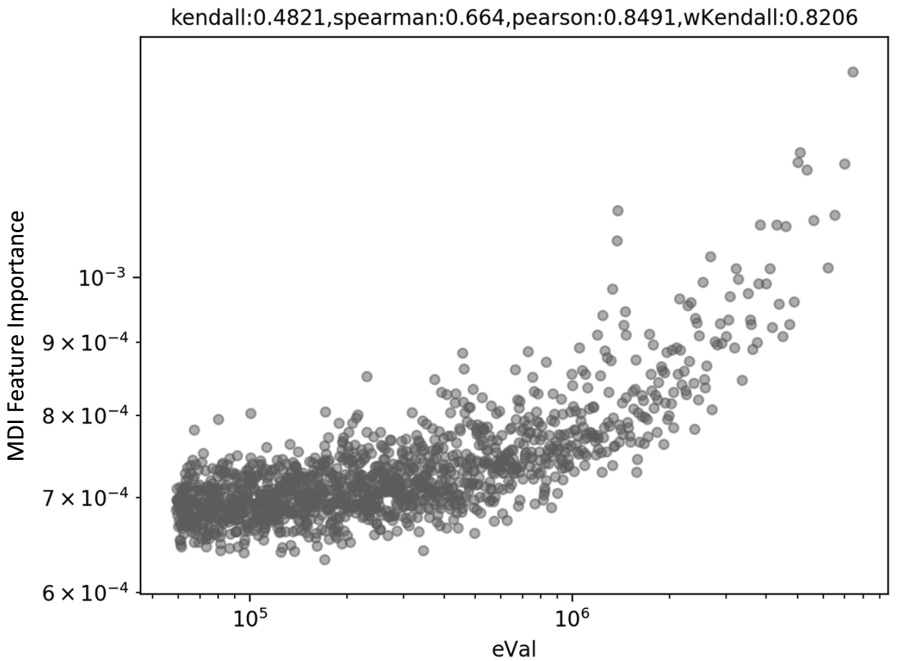


FIGURE 8.1 Scatter plot of eigenvalues (x-axis) and MDI levels (y-axis) in log-log scale

SNIPPET 8.6 COMPUTATION OF WEIGHTED KENDALL'S TAU BETWEEN FEATURE IMPORTANCE AND INVERSE PCA RANKING

```
>>> import numpy as np
>>> from scipy.stats import weightedtau
>>> featImp=np.array([.55,.33,.07,.05]) # feature importance
>>> pcRank=np.array([1,2,4,3]) # PCA rank
>>> weightedtau(featImp,pcRank**-1.)[0]
```

SNIPPET 8.7 CREATING A SYNTHETIC DATASET

```
def getTestData(n_features=40,n_informative=10,n_redundant=10,n_samples=10000):
    # generate a random dataset for a classification problem
    from sklearn.datasets import make_classification
    trnsX,cont=make_classification(n_samples=n_samples,n_features=n_features,
        n_informative=n_informative,n_redundant=n_redundant,random_state=0,
        shuffle=False)
    df0=pd.DatetimeIndex(periods=n_samples,freq=pd.tseries.offsets.BDay(),
                         end=pd.datetime.today())
    trnsX,cont=pd.DataFrame(trnsX,index=df0),
                           pd.Series(cont,index=df0).to_frame('bin')
    df0=['I_'+str(i) for i in xrange(n_informative)]+
        ['R_'+str(i) for i in xrange(n_redundant)]
    df0+=['N_'+str(i) for i in xrange(n_features-len(df0))]
    trnsX.columns=df0
    cont['w']=1./cont.shape[0]
    cont['t1']=pd.Series(cont.index,index=cont.index)
    return trnsX,cont
```

SNIPPET 8.8 CALLING FEATURE IMPORTANCE FOR ANY METHOD

```
def featImportance(trnsX,cont,n_estimators=1000,cv=10,max_samples=1.,numThreads=24,
                   pctEmbargo=0,scoring='accuracy',method='SFI',minWLeaf=0.,**kargs):
    # feature importance from a random forest
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import BaggingClassifier
    from mpEngine import mpPandasObj
    n_jobs=(-1 if numThreads>1 else 1) # run 1 thread with ht_helper in dirac1
    #1) prepare classifier,cv. max_features=1, to prevent masking
    clf=DecisionTreeClassifier(criterion='entropy',max_features=1,
        class_weight='balanced',min_weight_fraction_leaf=minWLeaf)
    clf=BaggingClassifier(base_estimator=clf,n_estimators=n_estimators,
        max_features=1.,max_samples=max_samples,oob_score=True,n_jobs=n_jobs)
    fit=clf.fit(X=trnsX,y=cont['bin'],sample_weight=cont['w'].values)
    oob=fit.oob_score_
    if method=='MDI':
        imp=featImpMDI(fit,featNames=trnsX.columns)
        oos=cvScore(clf,X=trnsX,y=cont['bin'],cv=cv,sample_weight=cont['w'],
                    t1=cont['t1'],pctEmbargo=pctEmbargo,scoring=scoring).mean()
    elif method=='MDA':
        imp,oos=featImpMDA(clf,X=trnsX,y=cont['bin'],cv=cv,sample_weight=cont['w'],
                           t1=cont['t1'],pctEmbargo=pctEmbargo,scoring=scoring)
    elif method=='SFI':
        cvGen=PurgedKFold(n_splits=cv,t1=cont['t1'],pctEmbargo=pctEmbargo)
        oos=cvScore(clf,X=trnsX,y=cont['bin'],sample_weight=cont['w'],scoring=scoring,
            cvGen=cvGen).mean()
        clf.n_jobs=1 # paralellize auxFeatImpSFI rather than clf
        imp=mpPandasObj(auxFeatImpSFI,('featNames',trnsX.columns),numThreads,
            clf=clf,trnsX=trnsX,cont=cont,scoring=scoring,cvGen=cvGen)
    return imp,oob,oos
```

SNIPPET 8.9 CALLING ALL COMPONENTS

```
def testFunc(n_features=40,n_informative=10,n_redundant=10,n_estimators=1000,
             n_samples=10000,cv=10):
    # test the performance of the feat importance functions on artificial data
    # Nr noise features = n_features—n_informative—n_redundant
    trnsX,cont=getTestData(n_features,n_informative,n_redundant,n_samples)
    dict0={'minWLeaf':[0.],'scoring':['accuracy'],'method':['MDI','MDA','SFI'],
        'max_samples':[1.]}
    jobs,out=(dict(izip(dict0,i)) for i in product(*dict0.values())),[]
    kargs={'pathOut':'./testFunc/','n_estimators':n_estimators,
        'tag':'testFunc','cv':cv}
    for job in jobs:
```

```
        job['simNum']=job['method']+'_'+job['scoring']+'_'+'%.2f'%job['minWLeaf']+ \
                      '_'+str(job['max_samples'])
        print job['simNum']
        kargs.update(job)
        imp,oob,oos=featImportance(trnsX=trnsX,cont=cont,**kargs)
        plotFeatImportance(imp=imp,oob=oob,oos=oos,**kargs)
        df0=imp[['mean']]/imp['mean'].abs().sum()
        df0['type']=[i[0] for i in df0.index]
        df0=df0.groupby('type')['mean'].sum().to_dict()
        df0.update({'oob':oob,'oos':oos});df0.update(job)
        out.append(df0)
    out=pd.DataFrame(out).sort_values(['method','scoring','minWLeaf','max_samples'])
    out=out['method','scoring','minWLeaf','max_samples','I','R','N','oob','oos']
    out.to_csv(kargs['pathOut']+'stats.csv')
    return
```

SNIPPET 8.10 FEATURE IMPORTANCE PLOTTING FUNCTION

```
def plotFeatImportance(pathOut,imp,oob,oos,method,tag=0,simNum=0,**kargs):
    # plot mean imp bars with std
    mpl.figure(figsize=(10,imp.shape[0]/5.))
    imp=imp.sort_values('mean',ascending=True)
    ax=imp['mean'].plot(kind='barh',color='b',alpha=.25,xerr=imp['std'],
                        error_kw={'ecolor':'r'})
    if method=='MDI':
        mpl.xlim([0,imp.sum(axis=1).max()])
        mpl.axvline(1./imp.shape[0],linewidth=1,color='r',linestyle='dotted')
    ax.get_yaxis().set_visible(False)
    for i,j in zip(ax.patches,imp.index):ax.text(i.get_width()/2,
                   i.get_y()+i.get_height()/2,j,ha='center',va='center',
                   color='black')
    mpl.title('tag='+tag+' | simNum='+str(simNum)+' | oob='+str(round(oob,4))+
              ' | oos='+str(round(oos,4)))
    mpl.savefig(pathOut+'featImportance_'+str(simNum)+'.png',dpi=100)
    mpl.clf();mpl.close()
    return
```

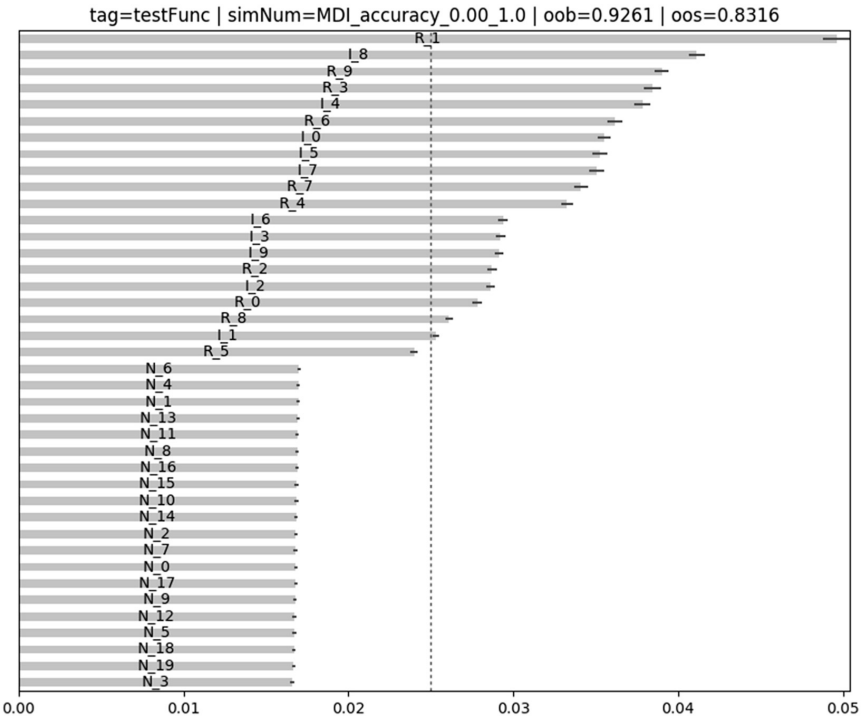


FIGURE 8.2 MDI feature importance computed on a synthetic dataset

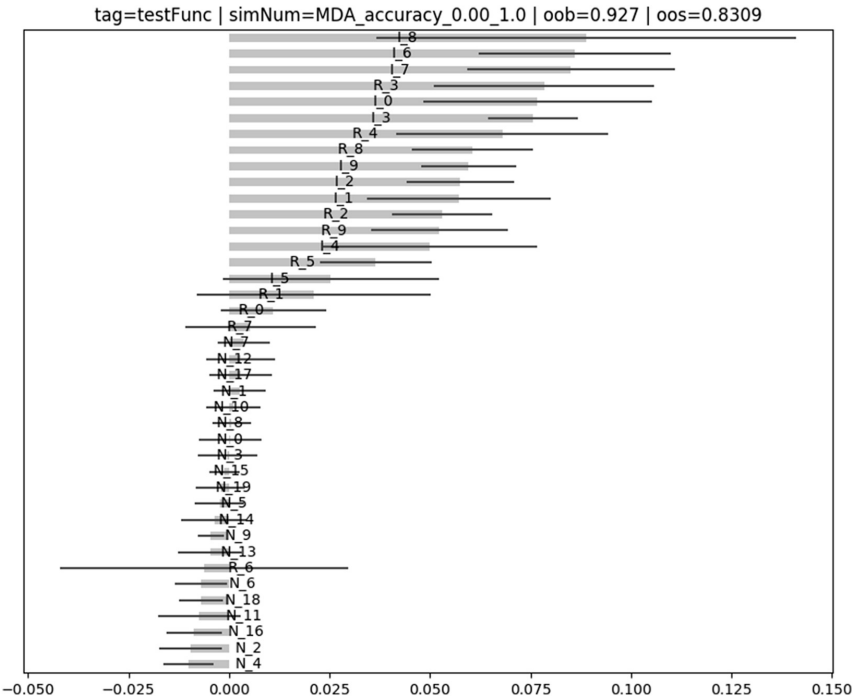


FIGURE 8.3 MDA feature importance computed on a synthetic dataset

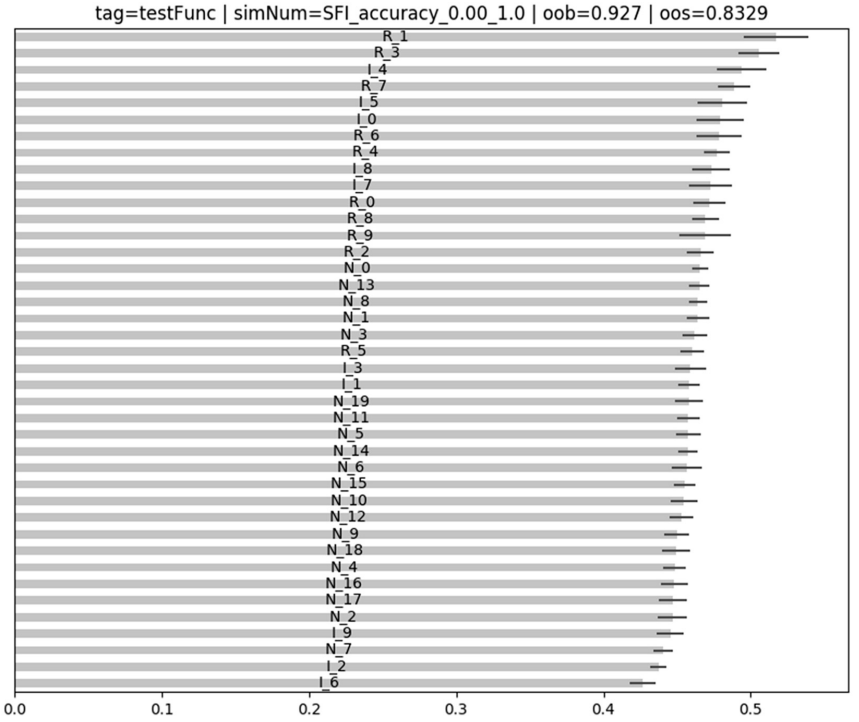


FIGURE 8.4 SFI feature importance computed on a synthetic dataset

SNIPPET 9.1 GRID SEARCH WITH PURGED K-FOLD CROSS-VALIDATION

```
def clfHyperFit(feat,lbl,t1,pipe_clf,param_grid,cv=3,bagging=[0,None,1.],
                n_jobs=-1,pctEmbargo=0,**fit_params):
    if set(lbl.values)=={0,1}:scoring='f1' # f1 for meta-labeling
    else:scoring='neg_log_loss' # symmetric towards all cases
    #1) hyperparameter search, on train data
    inner_cv=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged
    gs=GridSearchCV(estimator=pipe_clf,param_grid=param_grid,
        scoring=scoring,cv=inner_cv,n_jobs=n_jobs,iid=False)
    gs=gs.fit(feat,lbl,**fit_params).best_estimator_ # pipeline
    #2) fit validated model on the entirety of the data
    if bagging[1]>0:
        gs=BaggingClassifier(base_estimator=MyPipeline(gs.steps),
            n_estimators=int(bagging[0]),max_samples=float(bagging[1]),
            max_features=float(bagging[2]),n_jobs=n_jobs)
        gs=gs.fit(feat,lbl,sample_weight=fit_params \
            [gs.base_estimator.steps[-1][0]+'__sample_weight'])
        gs=Pipeline([('bag',gs)])
    return gs
```

SNIPPET 9.2 AN ENHANCED PIPELINE CLASS

```
class MyPipeline(Pipeline):
    def fit(self,X,y,sample_weight=None,**fit_params):
        if sample_weight is not None:
            fit_params[self.steps[-1][0]+'__sample_weight']=sample_weight
        return super(MyPipeline,self).fit(X,y,**fit_params)
```

SNIPPET 9.3 RANDOMIZED SEARCH WITH PURGED K-FOLD CV

```
def clfHyperFit(feat,lbl,t1,pipe_clf,param_grid,cv=3,bagging=[0,None,1.],
                rndSearchIter=0,n_jobs=-1,pctEmbargo=0,**fit_params):
    if set(lbl.values)=={0,1}:scoring='f1' # f1 for meta-labeling
    else:scoring='neg_log_loss' # symmetric towards all cases
    #1) hyperparameter search, on train data
    inner_cv=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged
    if rndSearchIter==0:
        gs=GridSearchCV(estimator=pipe_clf,param_grid=param_grid,
            scoring=scoring,cv=inner_cv,n_jobs=n_jobs,iid=False)
    else:
    gs=RandomizedSearchCV(estimator=pipe_clf,param_distributions= \
        param_grid,scoring=scoring,cv=inner_cv,n_jobs=n_jobs,
        iid=False,n_iter=rndSearchIter)
gs=gs.fit(feat,lbl,**fit_params).best_estimator_ # pipeline
#2) fit validated model on the entirety of the data
if bagging[1]>0:
    gs=BaggingClassifier(base_estimator=MyPipeline(gs.steps),
        n_estimators=int(bagging[0]),max_samples=float(bagging[1]),
        max_features=float(bagging[2]),n_jobs=n_jobs)
    gs=gs.fit(feat,lbl,sample_weight=fit_params \
        [gs.base_estimator.steps[-1][0]+'__sample_weight'])
    gs=Pipeline([('bag',gs)])
return gs
```

A random variable x follows a log-uniform distribution between $a > 0$ and $b > a$ if and only if $\log[x] \sim U\left[\log[a], \log[b]\right]$. This distribution has a CDF:

$$F[x] = \begin{cases} \dfrac{\log[x] - \log[a]}{\log[b] - \log[a]} & \text{for } a \leq x \leq b \\ 0 & \text{for } x < a \\ 1 & \text{for } x > b \end{cases}$$

From this, we derive a PDF:

$$f[x] = \begin{cases} \dfrac{1}{x \log\left[b/a\right]} & \text{for } a \leq x \leq b \\ 0 & \text{for } x < a \\ 0 & \text{for } x > b \end{cases}$$

Function 2

Note that the CDF is invariant to the base of the logarithm, since $\frac{\log\left[\frac{x}{a}\right]}{\log\left[\frac{b}{a}\right]} = \frac{\log_c\left[\frac{x}{a}\right]}{\log_c\left[\frac{b}{a}\right]}$ for any base c, thus the random variable is not a function of c. Snippet 9.4 implements (and tests) in `scipy.stats` a random variable where $[a, b] = [1E-3, 1E3]$, hence $\log[x] \sim U\left[\log[1E-3], \log[1E3]\right]$. Figure 9.1 illustrates the uniformity of the samples in log-scale.

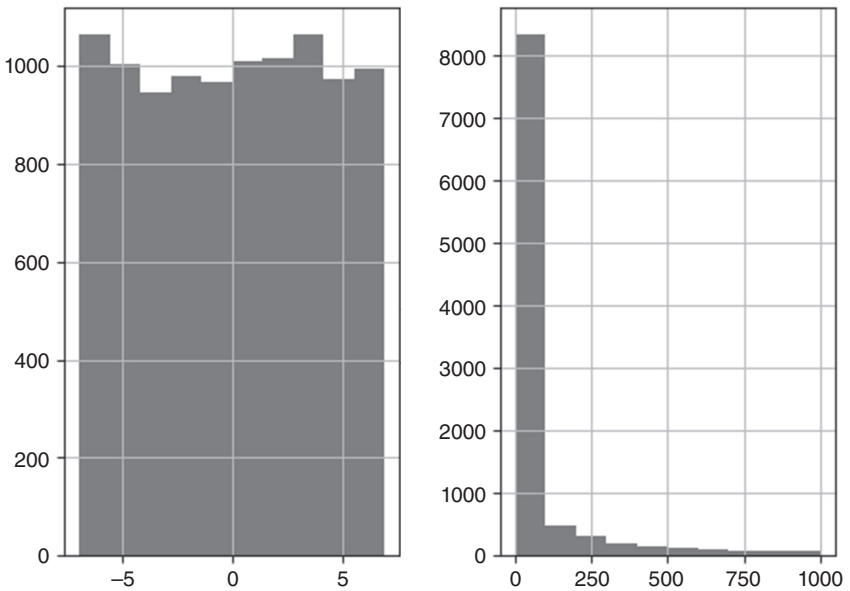


FIGURE 9.1 Result from testing the `logUniform_gen` class

SNIPPET 9.4 THE `logUniform_gen` CLASS

```
import numpy as np,pandas as pd,matplotlib.pyplot as mpl
from scipy.stats import rv_continuous,kstest
#———————————————————————————————
class logUniform_gen(rv_continuous):
    # random numbers log-uniformly distributed between 1 and e
    def _cdf(self,x):
        return np.log(x/self.a)/np.log(self.b/self.a)
def logUniform(a=1,b=np.exp(1)):return logUniform_gen(a=a,b=b,name='logUniform')
#———————————————————————————————
a,b,size=1E-3,1E3,10000
vals=logUniform(a=a,b=b).rvs(size=size)
print kstest(rvs=np.log(vals),cdf='uniform',args=(np.log(a),np.log(b/a)),N=size)
print pd.Series(vals).describe()
mpl.subplot(121)
pd.Series(np.log(vals)).hist()
mpl.subplot(122)
pd.Series(vals).hist()
mpl.show()
```

Investment strategies profit from predicting the right label with high confidence. Gains from good predictions with low confidence will not suffice to offset the losses from bad predictions with high confidence. For this reason, accuracy does not provide a realistic scoring of the classifier's performance. Conversely, log loss[3] (aka cross-entropy loss) computes the log-likelihood of the classifier given the true label, which takes predictions' probabilities into account. Log loss can be estimated as follows:

$$L[Y,P] = -\log[\text{Prob}[Y|P]] = -N^{-1}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} y_{n,k}\log\left[p_{n,k}\right]$$

where

- $p_{n,k}$ is the probability associated with prediction n of label k.
- Y is a 1-of-K binary indicator matrix, such that $y_{n,k} = 1$ when observation n was assigned label k out of K possible labels, and 0 otherwise.

Log 1

[3] http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss.

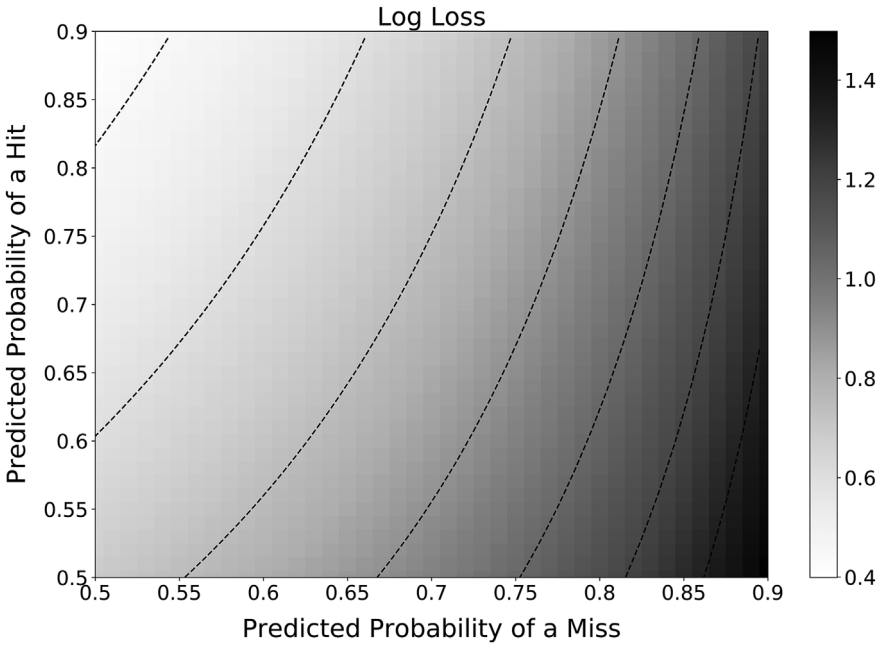


FIGURE 9.2 Log loss as a function of predicted probabilities of hit and miss

We would prefer to size positions in such way that we reserve some cash for the possibility that the trading signal strengthens before it weakens. One option is to compute the series $c_t = c_{t,l} - c_{t,s}$, where $c_{t,l}$ is the number of concurrent long bets at time t, and $c_{t,s}$ is the number of concurrent short bets at time t. This bet concurrency is derived, for each side, similarly to how we computed label concurrency in Chapter 4 (recall the `t1` object, with overlapping time spans). We fit a mixture of two Gaussians on {ct}, applying a method like the one described in Lopez de Prado and Foreman [2014]. Then, the bet size is derived as

$$m_t = \begin{cases} \dfrac{F[c_t] - F[0]}{1 - F[0]} & \text{if } c_t \geq 0 \\ \dfrac{F[c_t] - F[0]}{F[0]} & \text{if } c_t < 0 \end{cases}$$

Equation 36

where $F[x]$ is the CDF of the fitted mixture of two Gaussians for a value x. For example, we could size the bet as 0.9 when the probability of observing a signal of greater value is only 0.1. The stronger the signal, the smaller the probability that the signal becomes even stronger, hence the greater the bet size.

10.3 BET SIZING FROM PREDICTED PROBABILITIES

Let us denote $p\,[x]$ the probability that label x takes place. For two possible outcomes, $x \in \{-1, 1\}$, we would like to test the null hypothesis $H_0 : p\,[x = 1] = \frac{1}{2}$. We compute the test statistic $z = \frac{p[x=1]-\frac{1}{2}}{\sqrt{p[x=1](1-p[x=1])}} = \frac{2p[x=1]-1}{2\sqrt{p[x=1](1-p[x=1])}} \sim Z$, with $z \in (-\infty, +\infty)$ and where Z represents the standard Normal distribution. We derive the bet size as $m = 2Z\,[z] - 1$, where $m \in [-1, 1]$ and $Z\,[.]$ is the CDF of Z.

For more than two possible outcomes, we follow a one-versus-rest method. Let $X = \{-1, \ldots, 0, \ldots, 1\}$ be various labels associated with bet sizes, and $x \in X$ the predicted label. In other words, the label is identified by the bet size associated with it. For each label $i = 1, \ldots, \|X\|$, we estimate a probability p_i, with $\sum_{i=1}^{\|X\|} p_i = 1$. We define $\tilde{p} = \max_i\{p_i\}$ as the probability of x, and we would like to test for $H_0 : \tilde{p} = \frac{1}{\|X\|}$.[2] We compute the test statistic $z = \frac{\tilde{p}-\frac{1}{\|X\|}}{\sqrt{\tilde{p}(1-\tilde{p})}} \sim Z$, with $z \in [0\ , +\infty)$. We derive the bet size as $m = x\underbrace{(2Z\,[z] - 1)}_{\in[0,1]}$, where $m \in [-1, 1]$ and $Z\,[z]$ regulates the size for a prediction x (where the side is implied by x).

Figure 10.1 plots the bet size as a function of test statistic. Snippet 10.1 implements the translation from probabilities to bet size. It handles the possibility that the prediction comes from a meta-labeling estimator, as well from a standard labeling estimator. In step #2, it also averages active bets, and discretizes the final value, which we will explain in the following sections.

[2] Uncertainty is absolute when all outcomes are equally likely.

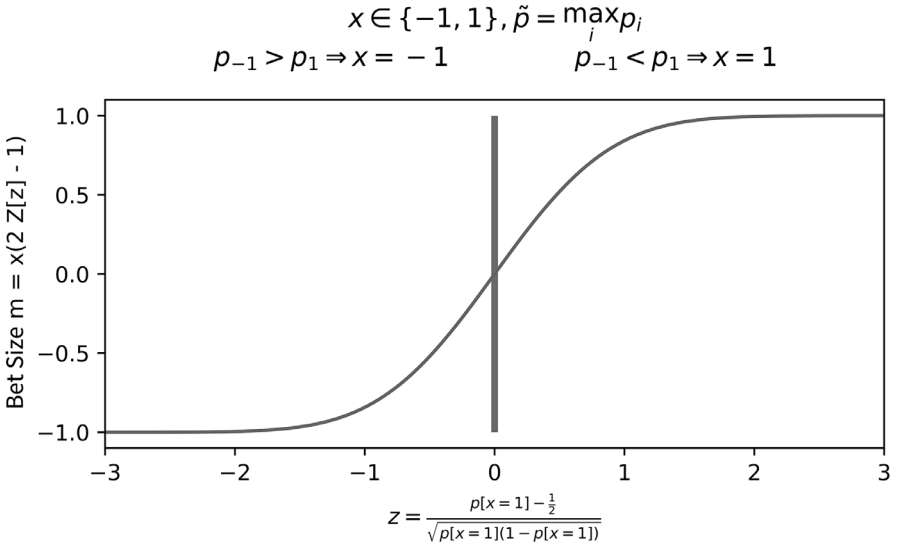


FIGURE 10.1 Bet size from predicted probabilities

SNIPPET 10.1 FROM PROBABILITIES TO BET SIZE

```
def getSignal(events,stepSize,prob,pred,numClasses,numThreads,**kargs):
    # get signals from predictions
    if prob.shape[0]==0:return pd.Series()
    #1) generate signals from multinomial classification (one-vs-rest, OvR)
    signal0=(prob-1./numClasses)/(prob*(1.-prob))**.5 # t-value of OvR
    signal0=pred*(2*norm.cdf(signal0)-1) # signal=side*size
    if 'side' in events:signal0*=events.loc[signal0.index,'side'] # meta-labeling
    #2) compute average signal among those concurrently open
    df0=signal0.to_frame('signal').join(events[['t1']],how='left')
    df0=avgActiveSignals(df0,numThreads)
    signal1=discreteSignal(signal0=df0,stepSize=stepSize)
    return signal1
```

SNIPPET 10.2 BETS ARE AVERAGED AS LONG AS THEY ARE STILL ACTIVE

```
def avgActiveSignals(signals,numThreads):
    # compute the average signal among those active
    #1) time points where signals change (either one starts or one ends)
    tPnts=set(signals['t1'].dropna().values)
    tPnts=tPnts.union(signals.index.values)
    tPnts=list(tPnts);tPnts.sort()
    out=mpPandasObj(mpAvgActiveSignals,('molecule',tPnts),numThreads,signals=signals)
    return out
#---------------------------------------
def mpAvgActiveSignals(signals,molecule):
  '''
  At time loc, average signal among those still active.
  Signal is active if:
      a) issued before or at loc AND
      b) loc before signal's endtime, or endtime is still unknown (NaT).
  '''
  out=pd.Series()
  for loc in molecule:
      df0=(signals.index.values<=loc)&((loc<signals['t1'])|pd.isnull(signals['t1']))
      act=signals[df0].index
      if len(act)>0:out[loc]=signals.loc[act,'signal'].mean()
      else:out[loc]=0 # no signals active at this time
  return out
```

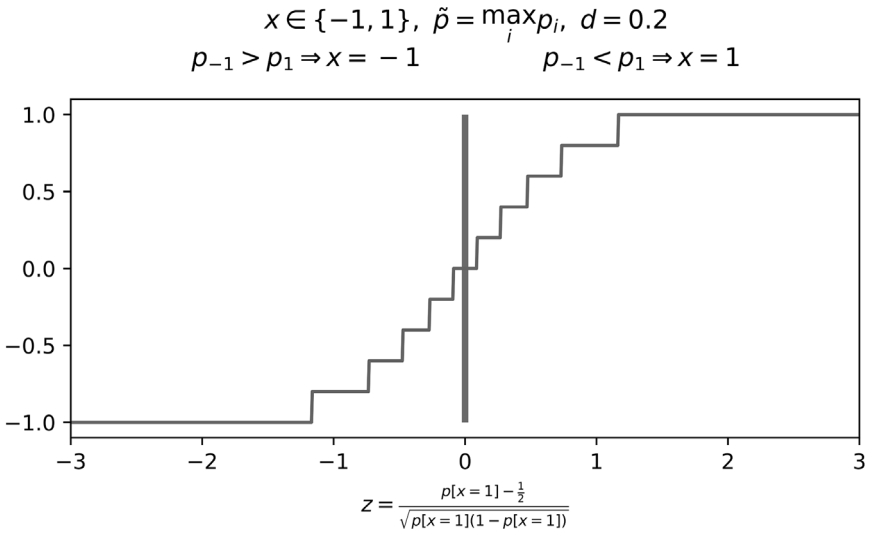


FIGURE 10.2 Discretization of the bet size, d = 0.2

SNIPPET 10.3 SIZE DISCRETIZATION TO PREVENT OVERTRADING

```
def discreteSignal(signal0,stepSize):
    # discretize signal
    signal1=(signal0/stepSize).round()*stepSize # discretize
    signal1[signal1>1]=1 # cap
    signal1[signal1<-1]=-1 # floor
    return signal1
```

10.6 DYNAMIC BET SIZES AND LIMIT PRICES

Recall the triple-barrier labeling method presented in Chapter 3. Bar i is formed at time $t_{i,0}$, at which point we forecast the first barrier that will be touched. That prediction implies a forecasted price, $\mathrm{E}_{t_{i,0}}[p_{t_{i,1}}]$, consistent with the barriers' settings. In the period elapsed until the outcome takes place, $t \in [t_{i,0}, t_{i,1}]$, the price p_t fluctuates and additional forecasts may be formed, $\mathrm{E}_{t_{j,0}}[p_{t_{i,1}}]$, where $j \in [i+1, I]$ and $t_{j,0} \leq t_{i,1}$. In Sections 10.4 and 10.5 we discussed methods for averaging the active bets and

discretizing the bet size as new forecasts are formed. In this section we will introduce an approach to adjust bet sizes as market price p_t and forecast price f_i fluctuate. In the process, we will derive the order's limit price.

Let q_t be the current position, Q the maximum absolute position size, and $\hat{q}_{i,t}$ the target position size associated with forecast f_i, such that

$$\hat{q_{i,t}} = \text{int}[m[\omega, f_i - p_t]Q]$$

$$m[\omega, x] = \frac{x}{\sqrt{\omega + x^2}}$$

where $m[\omega, x]$ is the bet size, $x = f_i - p_t$ is the divergence between the current market price and the forecast, ω is a coefficient that regulates the width of the sigmoid function, and Int $[x]$ is the integer value of x. Note that for a real-valued price divergence x, $-1 < m[\omega, x] < 1$, the integer value $\hat{q}_{i,t}$ is bounded $-Q < \hat{q}_{i,t} < Q$.

The target position size $\hat{q}_{i,t}$ can be dynamically adjusted as p_t changes. In particular, as $p_t \to f_i$ we get $\hat{q}_{i,t} \to 0$, because the algorithm wants to realize the gains. This implies a breakeven limit price $\bar{p}$ for the order size $\hat{q}_{i,t} - q_t$, to avoid realizing losses. In particular,

$$\bar{p} = \frac{1}{|\hat{q}_{i,t} - q_t|} \sum_{j=|q_t + \text{sgn}[\hat{q}_{i,t} - q_t]|}^{|\hat{q}_{i,t}|} L\left[f_i, \omega, \frac{j}{Q}\right]$$

where $L[f_i, \omega, m]$ is the inverse function of $m[\omega, f_i - p_t]$ with respect to p_t,

$$L[f_i, \omega, m] = f_i - m\sqrt{\frac{\omega}{1 - m^2}}$$

We do not need to worry about the case $m^2 = 1$, because $|\hat{q}_{i,t}| < 1$. Since this function is monotonic, the algorithm cannot realize losses as $p_t \to f_i$.

Let us calibrate ω. Given a user-defined pair (x, m^*), such that $x = f_i - p_t$ and $m^* = m[\omega, x]$, the inverse function of $m[\omega, x]$ with respect to ω is

$$\omega = x^2(m^{*-2} - 1)$$

Snippet 10.4 implements the algorithm that computes the dynamic position size and limit prices as a function of p_t and f_i. First, we calibrate the sigmoid function, so that it returns a bet size of $m^* = .95$ for a price divergence of $x = 10$. Second, we compute the target position $\hat{q}_{i,t}$ for a maximum position $Q = 100$, $f_i = 115$ and $p_t = 100$. If you try $f_i = 110$, you will get $\hat{q}_{i,t} = 95$, consistent with the calibration of ω. Third, the limit price for this order of size $\hat{q}_{i,t} - q_t = 97$ is $p_t < 112.3657 < f_i$, which is between the current price and the forecasted price.

SNIPPET 10.4 DYNAMIC POSITION SIZE AND LIMIT PRICE

```
def betSize(w,x):
    return x*(w+x**2)**-.5
#-------------------------------------------------
def getTPos(w,f,mP,maxPos):
    return int(betSize(w,f-mP)*maxPos)
#-------------------------------------------------
def invPrice(f,w,m):
    return f-m*(w/(1-m**2))**.5
#-------------------------------------------------
def limitPrice(tPos,pos,f,w,maxPos):
    sgn=(1 if tPos>=pos else -1)
    lP=0
    for j in xrange(abs(pos+sgn),abs(tPos+1)):
        lP+=invPrice(f,w,j/float(maxPos))
    lP/=tPos-pos
    return lP
#-------------------------------------------------
def getW(x,m):
  # 0<alpha<1
    return x**2*(m**-2-1)
#-------------------------------------------------
def main():
    pos,maxPos,mP,f,wParams=0,100,100,115,{'divergence':10,'m':.95}
    w=getW(wParams['divergence'],wParams['m']) # calibrate w
    tPos=getTPos(w,f,mP,maxPos) # get tPos
    lP=limitPrice(tPos,pos,f,w,maxPos) # limit price for order
    return
#-------------------------------------------------
if __name__=='__main__':main()
```

As an alternative to the sigmoid function, we could have used a power function $\tilde{m}[\omega, x] = \text{sgn}[x]|x|^{\omega}$, where $\omega \geq 0$, $x \in [-1, 1]$, which results in $\tilde{m}[\omega, x] \in [-1, 1]$. This alternative presents the advantages that:

- $\tilde{m}[\omega, -1] = -1, \tilde{m}[\omega, 1] = 1$.
- Curvature can be directly manipulated through ω.
- For $\omega > 1$, the function goes from concave to convex, rather than the other way around, hence the function is almost flat around the inflexion point.

We leave the derivation of the equations for a power function as an exercise. Figure 10.3 plots the bet sizes (y-axis) as a function of price divergence $f - p_t$ (x-axis) for both the sigmoid and power functions.

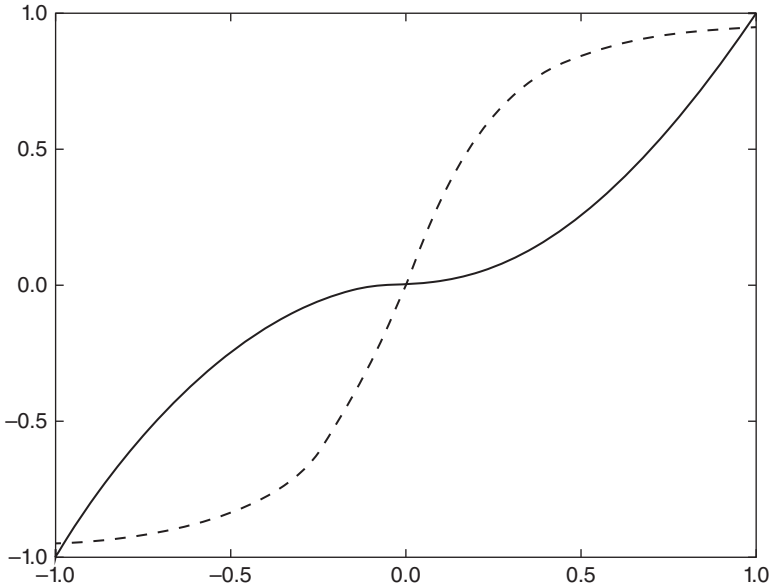


FIGURE 10.3 $f[x] = sgn[x]\,|x|^2$ (concave to convex) and $f[x] = x(.1 + x^2)^{-.5}$ (convex to concave)

Third, we form all combinations C_S of M_s, taken in groups of size $\frac{S}{2}$. This gives a total number of combinations

$$\binom{S}{S/2} = \binom{S-1}{S/2-1}\frac{S}{S/2} = \ldots = \prod_{i=0}^{S/2-1}\frac{S-i}{S/2-i}$$

Equation 37

For instance, if $S = 16$, we will form 12,780 combinations. Each combination $c \in C_S$ is composed of $S/2$ submatrices M_s.

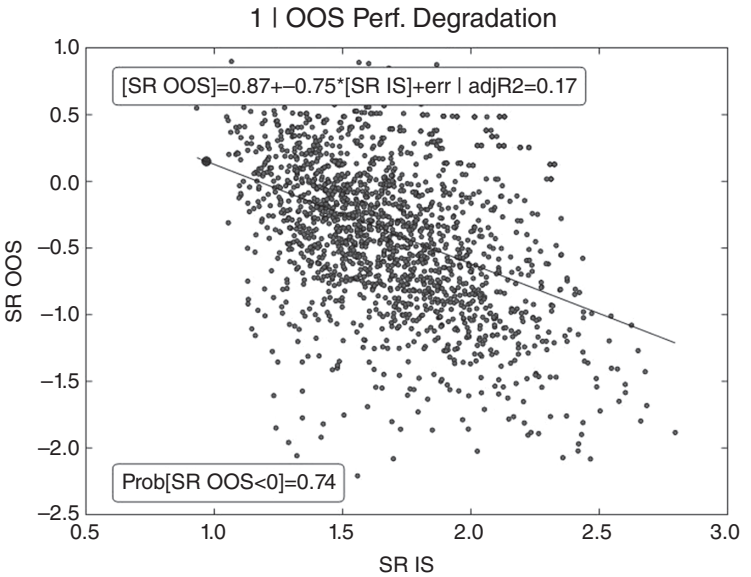


FIGURE 11.1 Best Sharpe ratio in-sample (SR IS) vs Sharpe ratio out-of-sample (SR OOS)

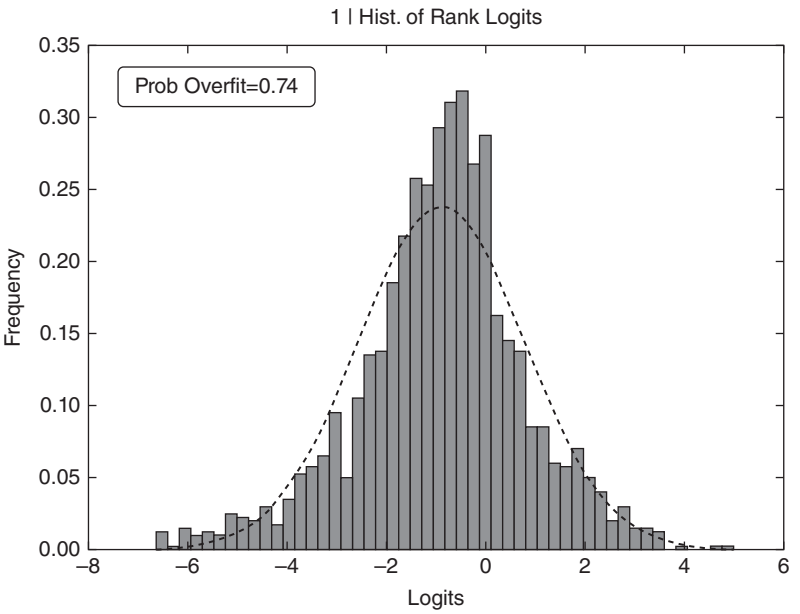


FIGURE 11.2 Probability of backtest overfitting derived from the distribution of logits

The third disadvantage of WF is that the initial decisions are made on a smaller portion of the total sample. Even if a warm-up period is set, most of the information is used by only a small portion of the decisions. Consider a strategy with a warm-up period that uses t_0 observations out of T. This strategy makes half of its decisions $\left(\frac{T-t_0}{2}\right)$ on an average number of datapoints,

$$\left(\frac{T-t_0}{2}\right)^{-1}\left(t_0+\frac{T+t_0}{2}\right)\frac{T-t_0}{4}=\frac{1}{4}T+\frac{3}{4}t_0$$

Equation 38

which is only a $\frac{3}{4}\frac{t_0}{T}+\frac{1}{4}$ fraction of the observations. Although this problem is attenuated by increasing the warm-up period, doing so also reduces the length of the backtest.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Paths
G1	x	x	x	x	x											5
G2	x					x	x	x	x							5
G3		x				x				x	x	x				5
G4			x				x			x			x	x		5
G5				x				x			x		x		x	5
G6					x				x			x		x	x	5

FIGURE 12.1 Paths generated for $\varphi[6, 2] = 5$

12.4.1 Combinatorial Splits

Consider T observations partitioned into N groups without shuffling, where groups $n = 1, \ldots, N-1$ are of size $\lfloor T/N \rfloor$, the Nth group is of size $T - \lfloor T/N \rfloor (N-1)$, and $\lfloor . \rfloor$ is the floor or integer function. For a testing set of size k groups, the number of possible training/testing splits is

$$\binom{N}{N-k} = \frac{\prod_{i=0}^{k-1}(N-i)}{k!}$$

Equation 39

Since each combination involves k tested groups, the total number of tested groups is $k\binom{N}{N-k}$. And since we have computed all possible combinations, these tested groups are uniformly distributed across all N (each group belongs to the same number of training and testing sets). The implication is that from k-sized testing sets on N groups we can backtest a total number of paths $\varphi[N, k]$,

$$\varphi[N,k] = \frac{k}{N}\binom{N}{N-k} = \frac{\prod_{i=1}^{k-1}(N-i)}{(k-1)!}$$

Equation 40

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Paths
G1	1	2	3	4	5											5
G2	1					2	3	4	5							5
G3		1				2				3	4	5				5
G4			1				2			3			4	5		5
G5				1				2			3		4		5	5
G6					1				2			3		4	5	5

FIGURE 12.2 Assignment of testing groups to each of the 5 paths

12.5 HOW COMBINATORIAL PURGED CROSS-VALIDATION ADDRESSES BACKTEST OVERFITTING

Given a sample of IID random variables, $x_i \sim Z$, $i = 1, \ldots, I$, where Z is the standard normal distribution, the expected maximum of that sample can be approximated as

$$\mathrm{E}[\max\{x_i\}_{i=1,\ldots,I}] \approx (1-\gamma) Z^{-1}\left[1-\frac{1}{I}\right] + \gamma Z^{-1}\left[1-\frac{1}{I}e^{-1}\right] \leq \sqrt{2\log[I]}$$

Equation 41

where $Z^{-1}[.]$ is the inverse of the CDF of Z, $\gamma \approx 0.5772156649\cdots$ is the Euler-Mascheroni constant, and $I \gg 1$ (see Bailey et al. [2014] for a proof). Now suppose that a researcher backtests I strategies on an instrument that behaves like a martingale, with Sharpe ratios $\{y_i\}_{i=1,\ldots,I}$, $\mathrm{E}[y_i] = 0$, $\sigma^2[y_i] > 0$, and $\frac{y_i}{\sigma[y_i]} \sim Z$. Even though the true Sharpe ratio is zero, we expect to find one strategy with a Sharpe ratio of

$$\mathrm{E}[\max\{y_i\}_{i=1,\ldots,I}] = \mathrm{E}[\max\{x_i\}_{i=1,\ldots,I}]\sigma[y_i]$$

Equation 42

CV backtests (Section 12.3) address that source of variance by training each classifier on equal and large portions of the dataset. Although CV leads to fewer false discoveries than WF, both approaches still estimate the Sharpe ratio from a single path for a srategy i, y_i, and that estimation may be highly volatile. In contrast, CPCV derives the distribution of Sharpe ratios from a large number of paths, $j = 1, \ldots, \varphi$, with mean $\mathrm{E}[\{y_{i,j}\}_{j=1,\ldots,\varphi}] = \mu_i$ and variance $\sigma^2[\{y_{i,j}\}_{j=1,\ldots,\varphi}] = \sigma_i^2$. The variance of the sample mean of CPCV paths is

$$\sigma^2[\mu_i] = \varphi^{-2}\left(\varphi\sigma_i^2 + \varphi(\varphi-1)\sigma_i^2\bar{\rho}_i\right) = \varphi^{-1}\sigma_i^2\left(1 + (\varphi-1)\bar{\rho}_i\right)$$

Equation 43a

where σ_i^2 is the variance of the Sharpe ratios across paths for strategy i, and $\bar{\rho}_i$ is the average off-diagonal correlation among $\{y_{i,j}\}_{j=1,\ldots,\varphi}$. CPCV leads to fewer false discoveries than CV and WF, because $\bar{\rho}_i < 1$ implies that the variance of the sample mean is lower than the variance of the sample,

$$\varphi^{-1}\sigma_i^2 \leq \sigma^2\left[\mu_i\right] < \sigma_i^2$$

Equation 43b

$$R^* = \arg\max_{R \in \Omega} \{SR_R\}$$
$$SR_R = \frac{\mathrm{E}[\pi_{i,T_i}|R]}{\sigma[\pi_{i,T_i}|R]}$$

Equation 44

Definition 2: Overfit Trading Rule: R^* is overfit if $\mathrm{E}\left[\frac{E\left[\pi_{j,T_j}\middle|R^*\right]}{\sigma\left[\pi_{j,T_j}\middle|R^*\right]}\right] < \mathrm{Me}_\Omega\left[\mathrm{E}\left[\frac{E\left[\pi_{j,T_j}\middle|R\right]}{\sigma\left[\pi_{j,T_j}\middle|R\right]}\right]\right]$, where $j = I+1, \ldots J$ and $\mathrm{Me}_\Omega[.]$ is the median.

13.4 OUR FRAMEWORK

Until now we have not characterized the stochastic process from which observations $\pi_{i,t}$ are drawn. We are interested in finding an optimal trading rule (OTR) for those scenarios where overfitting would be most damaging, such as when $\pi_{i,t}$ exhibits serial correlation. In particular, suppose a discrete Ornstein-Uhlenbeck (O-U) process on prices

$$P_{i,t} = (1-\varphi)\,\mathrm{E}_0[P_{i,T_i}] + \varphi P_{i,t-1} + \sigma\varepsilon_{i,t} \tag{13.2}$$

such that the random shocks are IID distributed $\varepsilon_{i,t} \sim N(0,1)$. The seed value for this process is $P_{i,0}$, the level targeted by opportunity i is $\mathrm{E}_0[P_{i,T_i}]$, and φ determines the speed at which $P_{i,0}$ converges towards $\mathrm{E}_0[P_{i,T_i}]$. Because $\pi_{i,t} = m_i(P_{i,t} - P_{i,0})$, equation (13.2) implies that the performance of opportunity i is characterized by the process

$$\frac{1}{m_i}\pi_{i,t} = (1-\varphi)\mathrm{E}_0[P_{i,T_i}] - P_{i,0} + \varphi P_{i,t-1} + \sigma\varepsilon_{i,t} \tag{13.3}$$

From the proof to Proposition 4 in Bailey and López de Prado [2013], it can be shown that the distribution of the process specified in equation (13.2) is Gaussian with parameters

$$\pi_{i,t} \sim N\left[m_i\left((1-\varphi)\,\mathrm{E}_0[P_{i,T_i}]\sum_{j=0}^{t-1}\varphi^j - P_{i,0}\right), m_i^2\sigma^2\sum_{j=0}^{t-1}\varphi^{2j}\right] \tag{13.4}$$

and a necessary and sufficient condition for its stationarity is that $\varphi \in (-1,1)$. Given a set of input parameters $\{\sigma,\varphi\}$ and initial conditions $\{P_{i,0}, \mathrm{E}_0[P_{i,T_i}]\}$ associated with opportunity i, is there an OTR $R^* := (\underline{\pi},\bar{\pi})$? Similarly, should strategy S predict a profit target $\bar{\pi}$, can we compute the optimal stop-loss $\underline{\pi}$ given the input values $\{\sigma,\varphi\}$? If the answer to these questions is affirmative, no backtest would be needed in order to determine R^*, thus avoiding the problem of overfitting the trading rule. In the next section we will show how to answer these questions experimentally.

13.5.1 The Algorithm

The algorithm consists of five sequential steps.

Step 1: We estimate the input parameters $\{\sigma, \varphi\}$, by linearizing equation (13.2) as:

$$P_{i,t} = \mathrm{E}_0[P_{i,T_i}] + \varphi(P_{i,t-1} - \mathrm{E}_0[P_{i,T_i}]) + \xi_t \tag{13.5}$$

We can then form vectors X and Y by sequencing opportunities:

$$X = \begin{bmatrix} P_{0,0} - \mathrm{E}_0[P_{0,T_0}] \\ P_{0,1} - \mathrm{E}_0[P_{0,T_0}] \\ \cdots \\ P_{0,T-1} - \mathrm{E}_0[P_{0,T_0}] \\ \cdots \\ P_{I,0} - \mathrm{E}_0[P_{I,T_I}] \\ \cdots \\ P_{I,T-1} - \mathrm{E}_0[P_{I,T_I}] \end{bmatrix};\ Y = \begin{bmatrix} P_{0,1} \\ P_{0,2} \\ \cdots \\ P_{0,T} \\ \cdots \\ P_{I,1} \\ \cdots \\ P_{I,T} \end{bmatrix};\ Z = \begin{bmatrix} \mathrm{E}_0[P_{0,T_0}] \\ \mathrm{E}_0[P_{0,T_0}] \\ \cdots \\ \mathrm{E}_0[P_{0,T_0}] \\ \cdots \\ \mathrm{E}_0[P_{I,T_I}] \\ \cdots \\ \mathrm{E}_0[P_{I,T_I}] \end{bmatrix} \tag{13.6}$$

Applying OLS on equation (13.5), we can estimate the original O-U parameters as,

$$\begin{aligned} \hat{\varphi} &= \frac{\mathrm{cov}\,[Y, X]}{\mathrm{cov}\,[X, X]} \\ \hat{\xi}_t &= Y - Z - \hat{\varphi} X \\ \hat{\sigma} &= \sqrt{\mathrm{cov}[\hat{\xi}_t, \hat{\xi}_t]} \end{aligned} \tag{13.7}$$

where $\mathrm{cov}\,[\cdot, \cdot]$ is the covariance operator.

Step 2: We construct a mesh of stop-loss and profit-taking pairs, $(\underline{\pi}, \bar{\pi})$. For example, a Cartesian product of $\underline{\pi} = \{-\frac{1}{2}\sigma, -\sigma, \ldots, -10\sigma\}$ and $\bar{\pi} = \{\frac{1}{2}\sigma, \sigma, \ldots, 10\sigma\}$ give us 20×20 nodes, each constituting an alternative trading rule $R \in \Omega$.

Step 3: We generate a large number of paths (e.g., 100,000) for $\pi_{i,t}$ applying our estimates $\{\hat{\sigma}, \hat{\varphi}\}$. As seed values, we use the observed initial conditions $\{P_{i,0}, \mathrm{E}_0[P_{i,T_i}]\}$ associated with an opportunity i. Because a position cannot be held for an unlimited period of time, we can impose a maximum holding period (e.g., 100 observations) at which point the position is exited even though $\underline{\pi} \leq \pi_{i,100} \leq \bar{\pi}$. This maximum holding period is equivalent to the vertical bar of the triple-barrier method (Chapter 3).[3]

Step 4: We apply the 100,000 paths generated in Step 3 on each node of the 20×20 mesh $(\underline{\pi}, \bar{\pi})$ generated in Step 2. For each node, we apply the stop-loss and profit-taking logic, giving us 100,000 values of π_{i,T_i}. Likewise, for each node we compute the Sharpe ratio associated with that trading rule as described in equation (13.1). See Bailey and López de Prado [2012] for a study of the confidence interval of the Sharpe ratio estimator. This result can be used in three different ways: Step 5a, Step 5b and Step 5c).

Step 5a: We determine the pair $(\underline{\pi}, \bar{\pi})$ within the mesh of trading rules that is optimal, given the input parameters $\{\hat{\sigma}, \hat{\varphi}\}$ and the observed initial conditions $\{P_{i,0}, \mathrm{E}_0[P_{i,T_i}]\}$.

Step 5b: If strategy S provides a profit target $\overline{\pi_i}$ for a particular opportunity i, we can use that information in conjunction with the results in Step 4 to determine the optimal stop-loss, $\underline{\pi_i}$.

Step 5c: If the trader has a maximum stop-loss $\underline{\pi_i}$ imposed by the fund's management for opportunity i, we can use that information in conjunction with the results in Step 4 to determine the optimal profit-taking $\overline{\pi_i}$ within the range of stop-losses $[0, \underline{\pi_i}]$.

Bailey and López de Prado [2013] prove that the half-life of the process in equation (13.2) is $\tau = -\frac{\log[2]}{\log[\varphi]}$, with the requirement that $\varphi \in (0, 1)$. From that result, we can determine the value of φ associated with a certain half-life τ as $\varphi = 2^{-1/\tau}$.

[3] The trading rule R could be characterized as a function of the three barriers, instead of the horizontal ones. That change would have no impact on the procedure. It would merely add one more dimension to the mesh ($20 \times 20 \times 20$). In this chapter we do not consider that setting, because it would make the visualization of the method less intuitive.

SNIPPET 13.1 PYTHON CODE FOR THE DETERMINATION OF OPTIMAL TRADING RULES

```
import numpy as np
from random import gauss
from itertools import product
#————————————————————————————————
def main():
    rPT=rSLm=np.linspace(0,10,21)
    count=0
    for prod_ in product([10,5,0,-5,-10],[5,10,25,50,100]):
        count+=1
        coeffs={'forecast':prod_[0],'hl':prod_[1],'sigma':1}
        output=batch(coeffs,nIter=1e5,maxHP=100,rPT=rPT,rSLm=rSLm)
    return output
```

SNIPPET 13.2 PYTHON CODE FOR THE DETERMINATION OF OPTIMAL TRADING RULES

```
def batch(coeffs,nIter=1e5,maxHP=100,rPT=np.linspace(.5,10,20),
    rSLm=np.linspace(.5,10,20),seed=0):
    phi,output1=2**(-1./coeffs['hl']),[]
    for comb_ in product(rPT,rSLm):
        output2=[]
        for iter_ in range(int(nIter)):
            p,hp,count=seed,0,0
            while True:
                p=(1-phi)*coeffs['forecast']+phi*p+coeffs['sigma']*gauss(0,1)
                cP=p-seed;hp+=1
                if cP>comb_[0] or cP<-comb_[1] or hp>maxHP:
                    output2.append(cP)
                    break
    mean,std=np.mean(output2),np.std(output2)
    print comb_[0],comb_[1],mean,std,mean/std
    output1.append((comb_[0],comb_[1],mean,std,mean/std))
return output1
```

TABLE 13.1 Input Parameter Combinations Used in the Simulations

Figure	Forecast	Half-Life	Sigma	maxHP
16.1	0	5	1	100
16.2	0	10	1	100
16.3	0	25	1	100
16.4	0	50	1	100
16.5	0	100	1	100
16.6	5	5	1	100
16.7	5	10	1	100
16.8	5	25	1	100
16.9	5	50	1	100
16.10	5	100	1	100
16.11	10	5	1	100
16.12	10	10	1	100
16.13	10	25	1	100
16.14	10	50	1	100
16.15	10	100	1	100
16.16	– 5	5	1	100
16.17	– 5	10	1	100
16.18	– 5	25	1	100
16.19	– 5	50	1	100
16.20	– 5	100	1	100
16.21	– 10	5	1	100
16.22	– 10	10	1	100
16.23	– 10	25	1	100
16.24	– 10	50	1	100
16.25	– 10	100	1	100

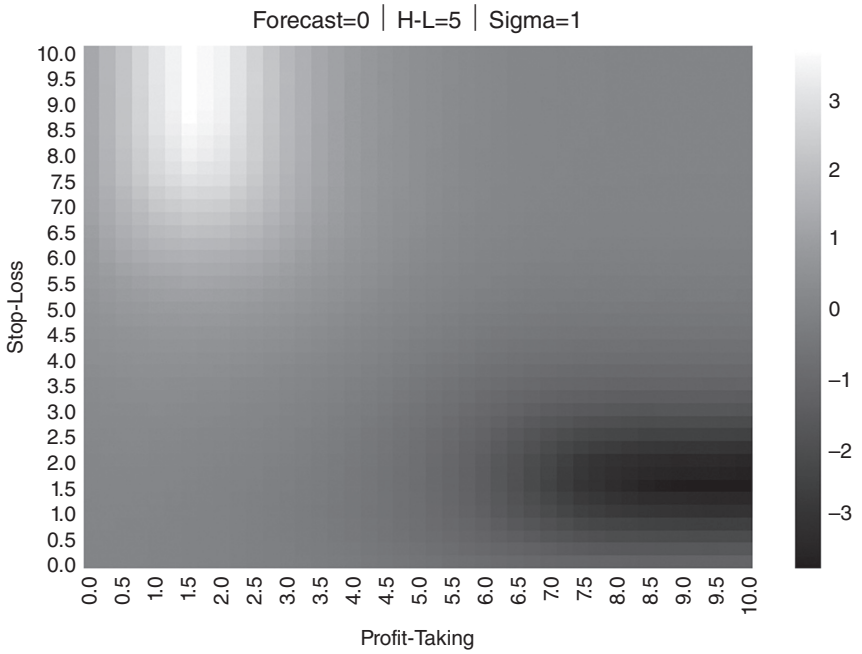


FIGURE 13.1 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{0, 5, 1\}$

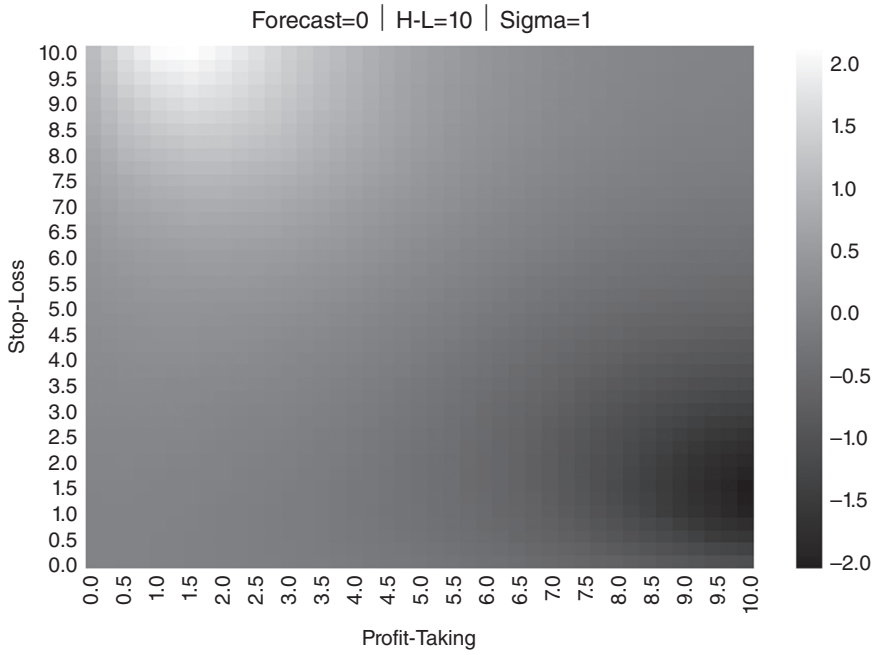


FIGURE 13.2 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{0, 10, 1\}$

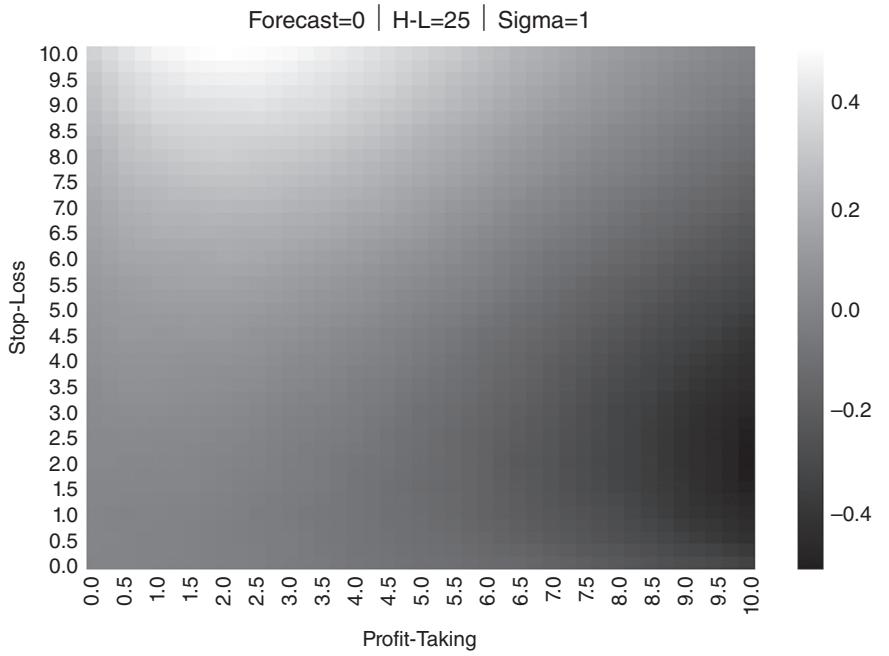


FIGURE 13.3 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{0, 25, 1\}$

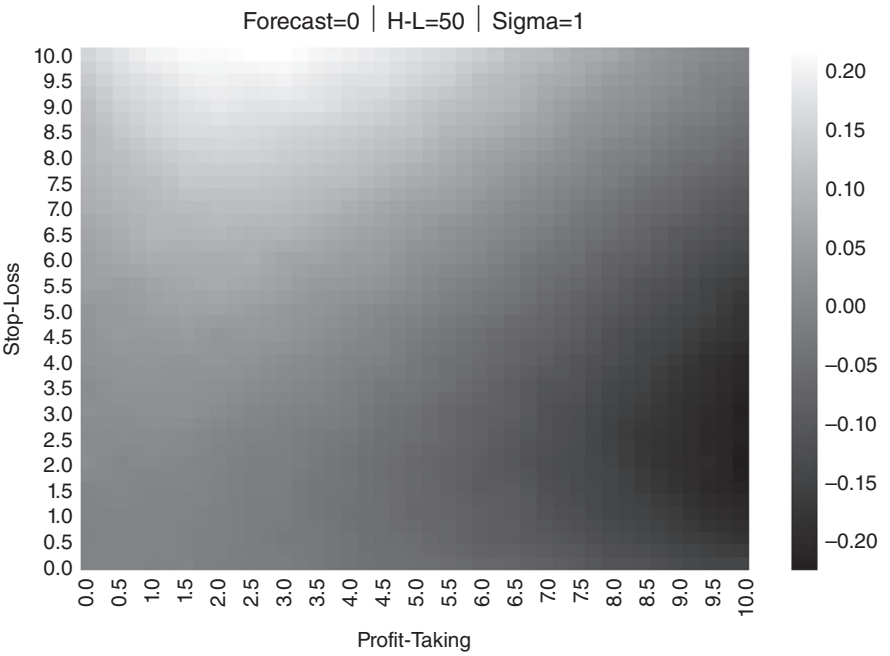


FIGURE 13.4 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{0, 50, 1\}$

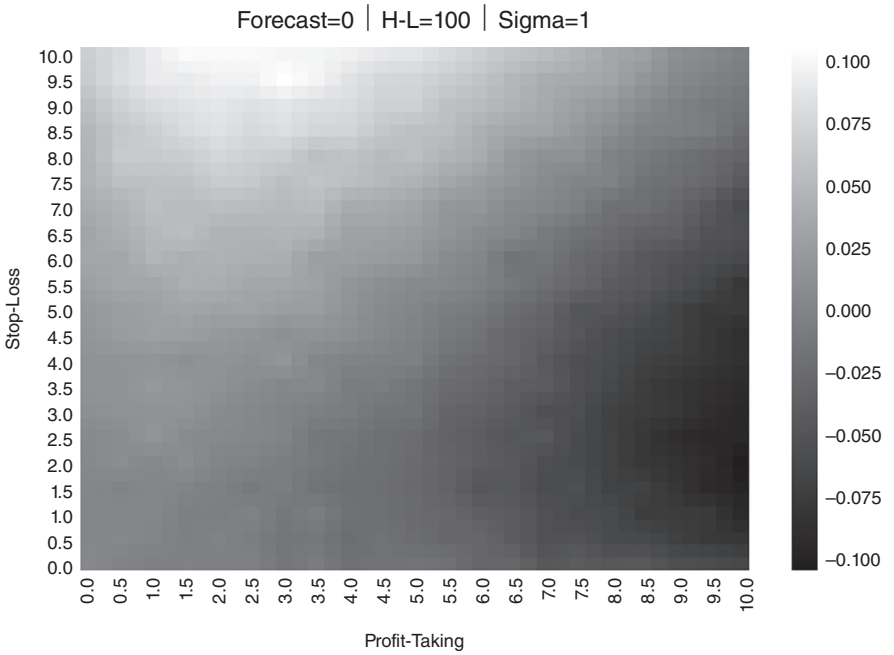


FIGURE 13.5 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{0, 100, 1\}$

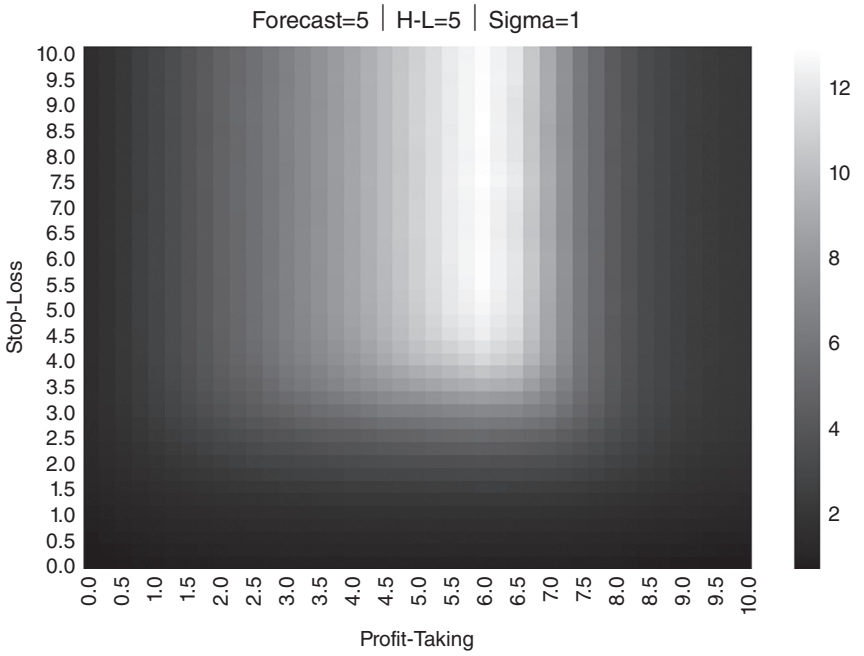


FIGURE 13.6 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{5, 5, 1\}$

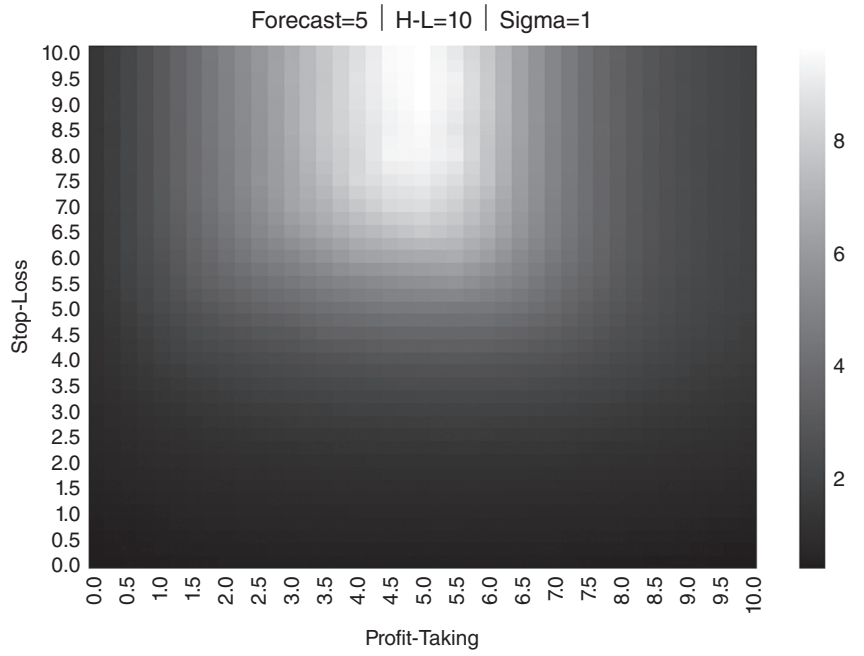


FIGURE 13.7 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{5, 10, 1\}$

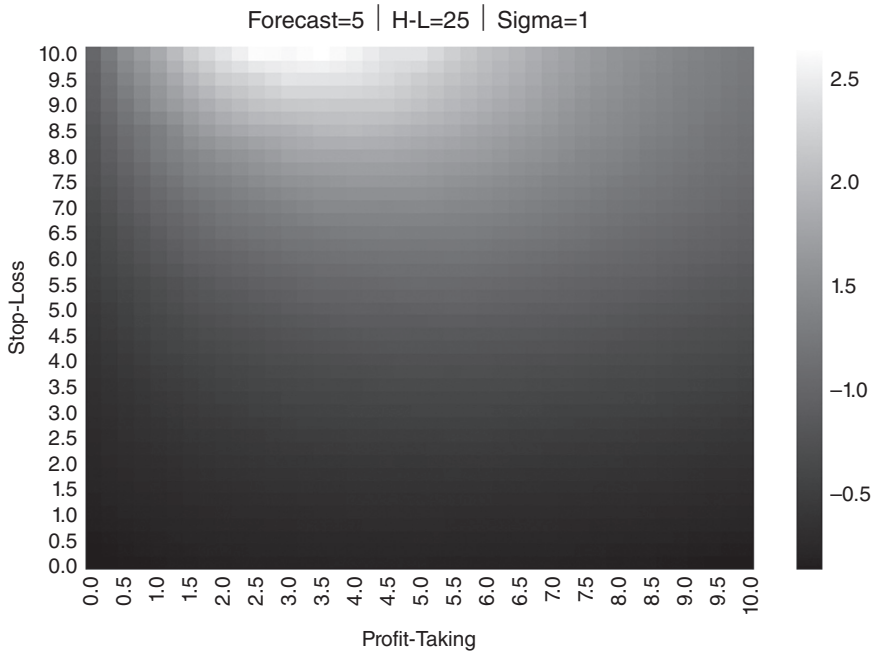


FIGURE 13.8 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{5, 25, 1\}$

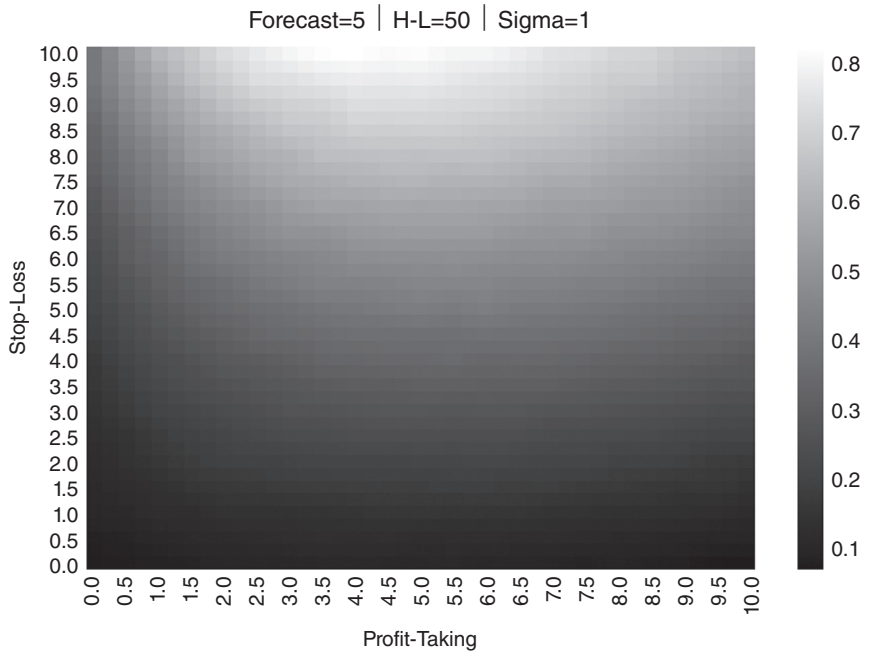


FIGURE 13.9 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{5, 50, 1\}$

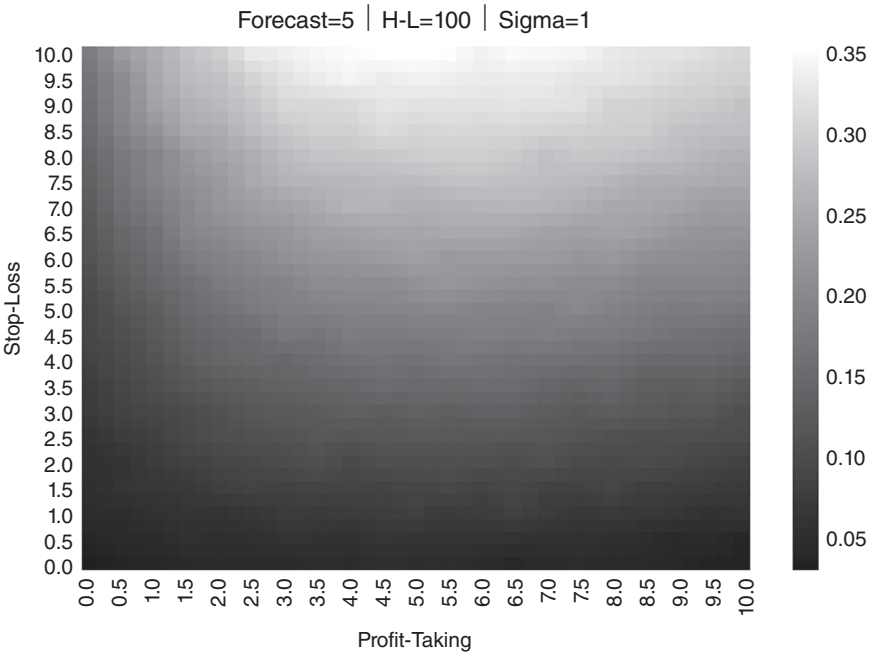


FIGURE 13.10 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{5{,}100{,}1\}$

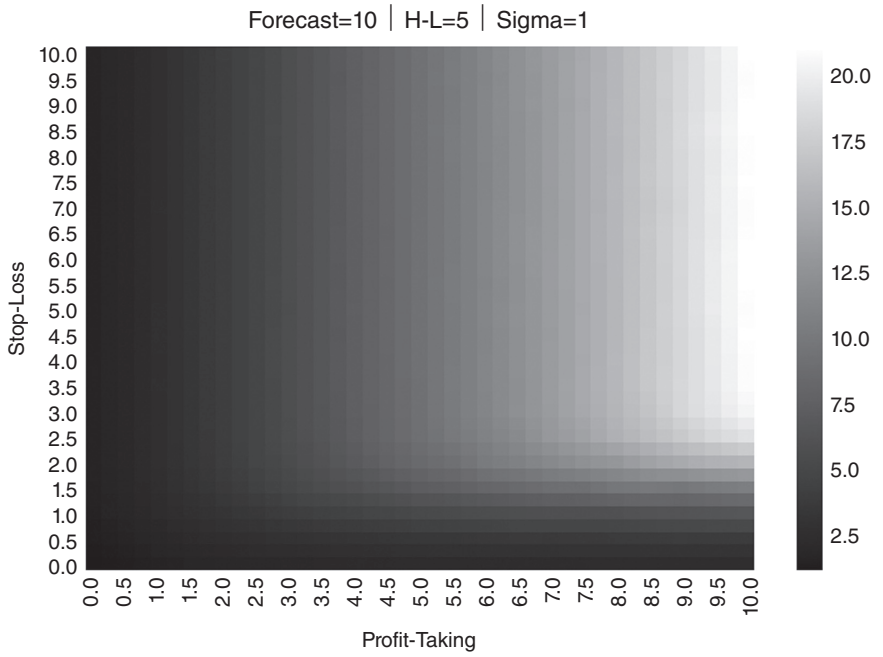


FIGURE 13.11 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{10, 5, 1\}$

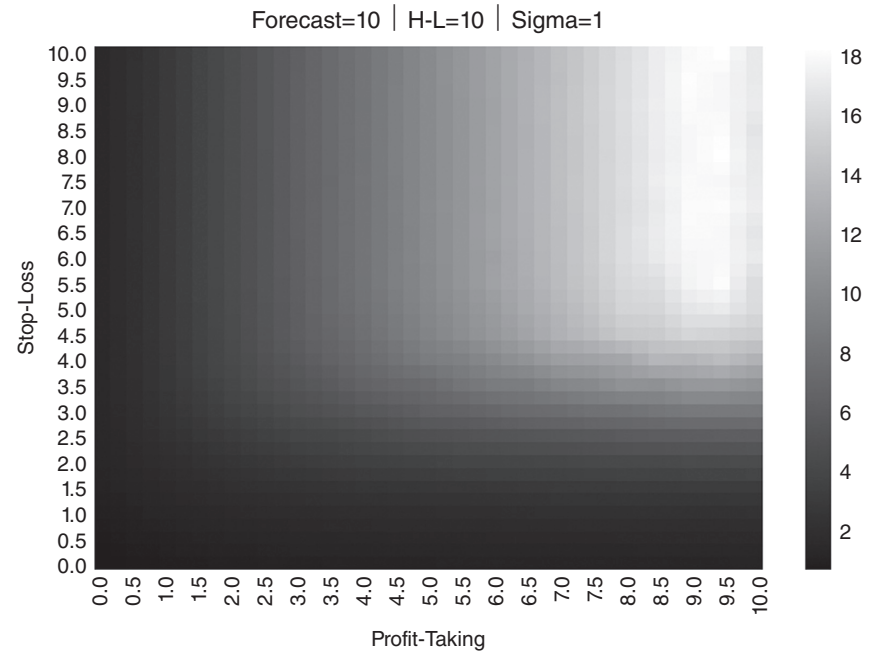


FIGURE 13.12 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{10, 10, 1\}$

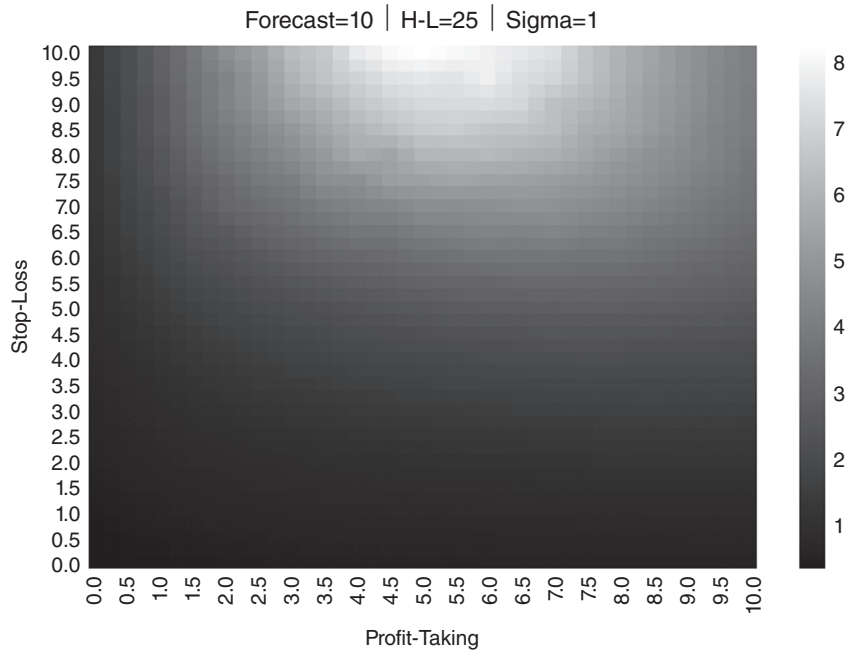


FIGURE 13.13 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{10, 25, 1\}$

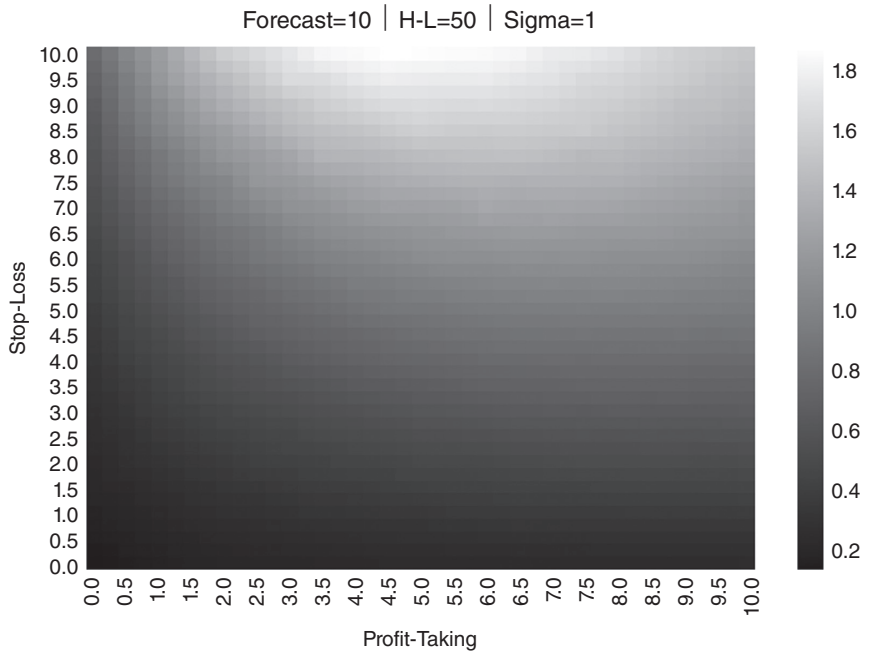


FIGURE 13.14 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{10, 50, 1\}$

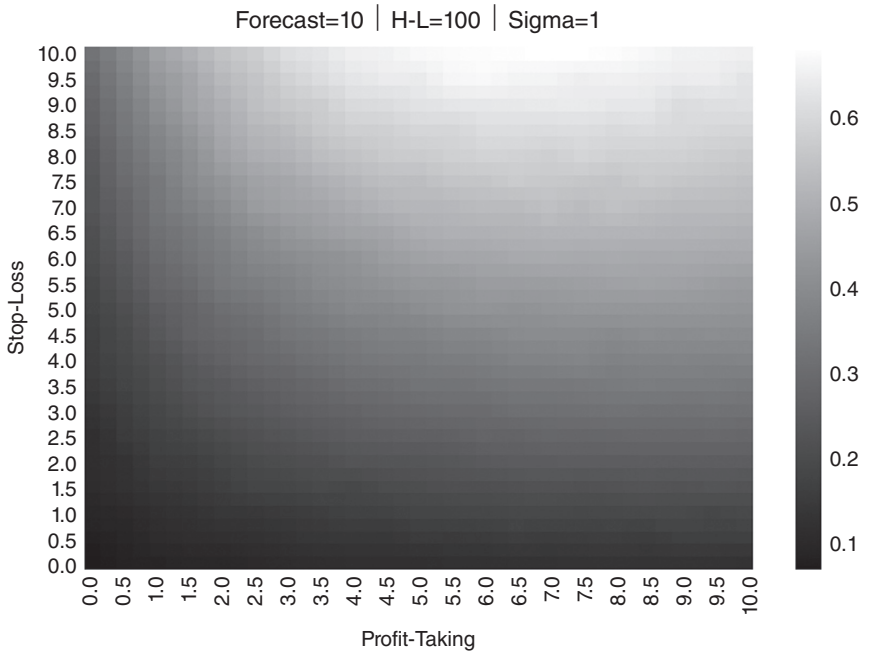


FIGURE 13.15 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{10, 100, 1\}$

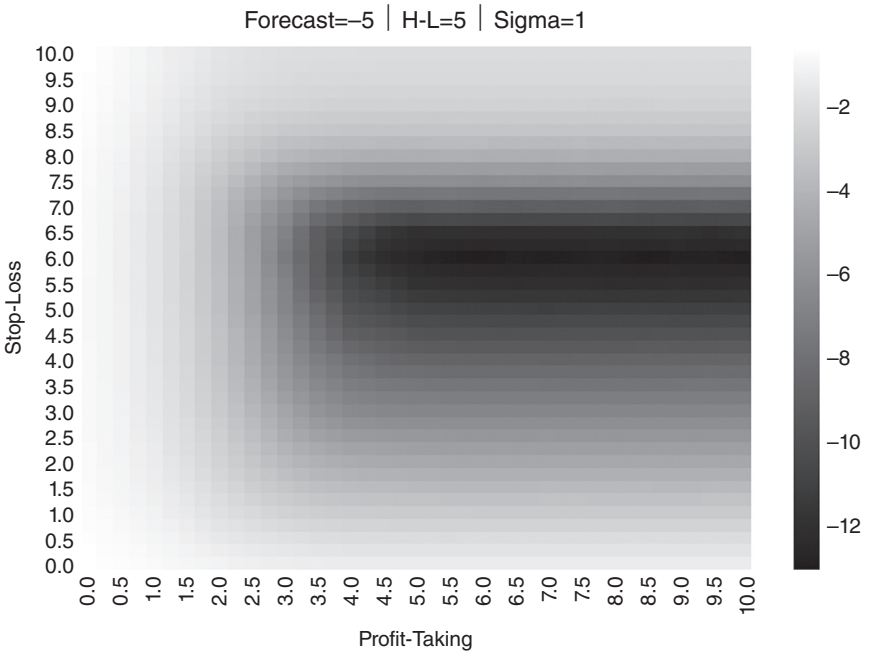


FIGURE 13.16 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-5, 5, 1\}$

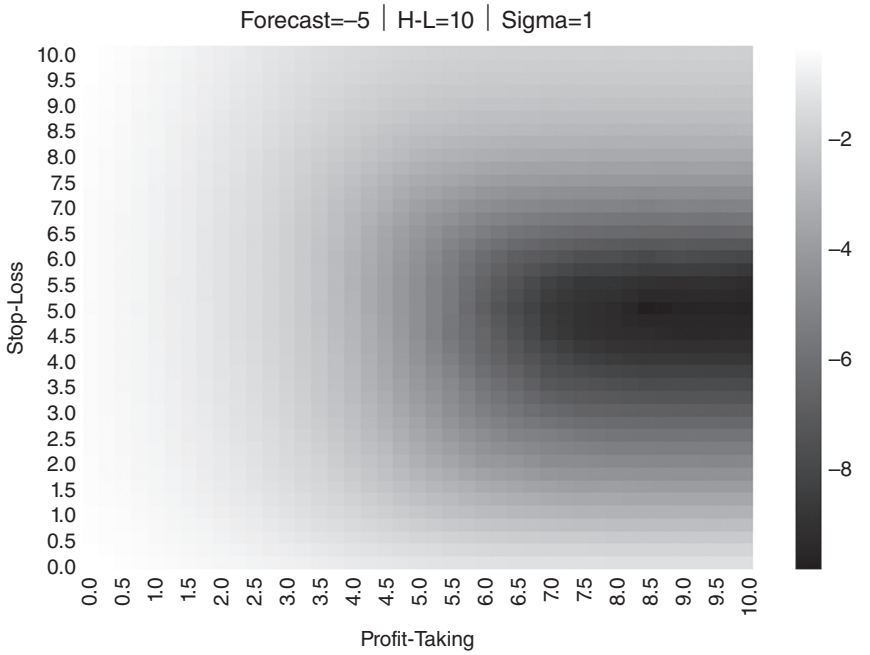


FIGURE 13.17 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-5, 10, 1\}$

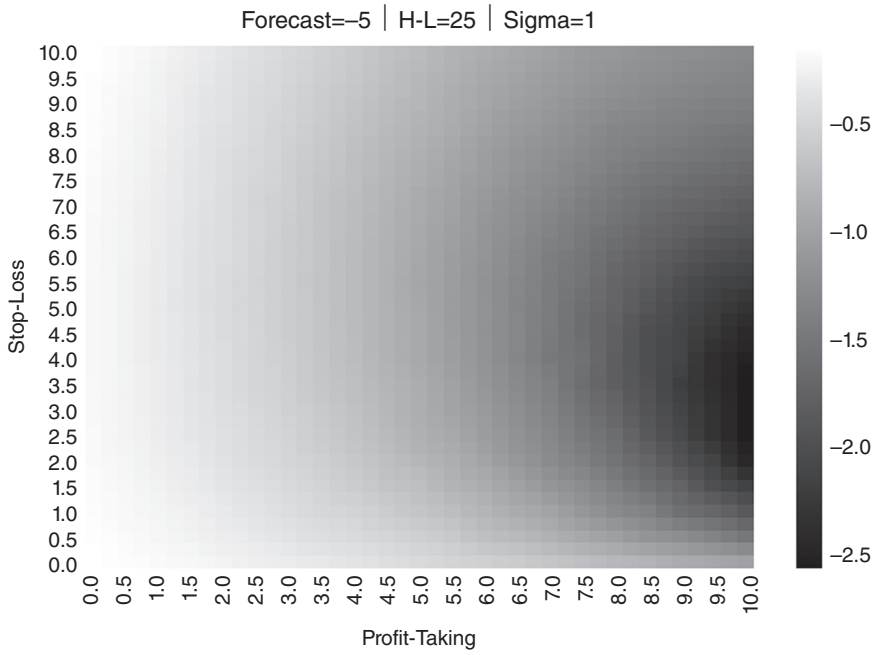


FIGURE 13.18 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-5, 25, 1\}$

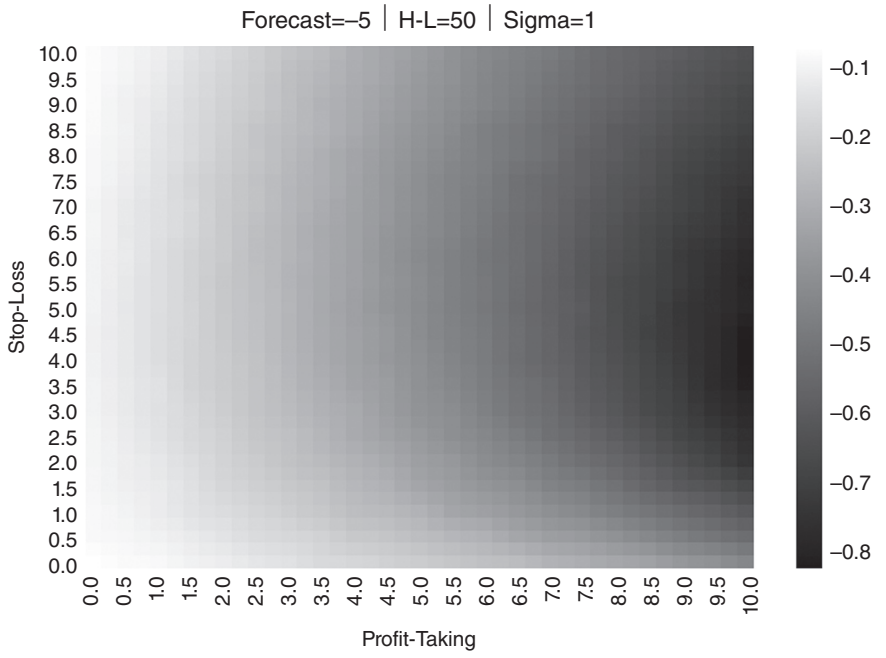


FIGURE 13.19 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-5, 50, 1\}$

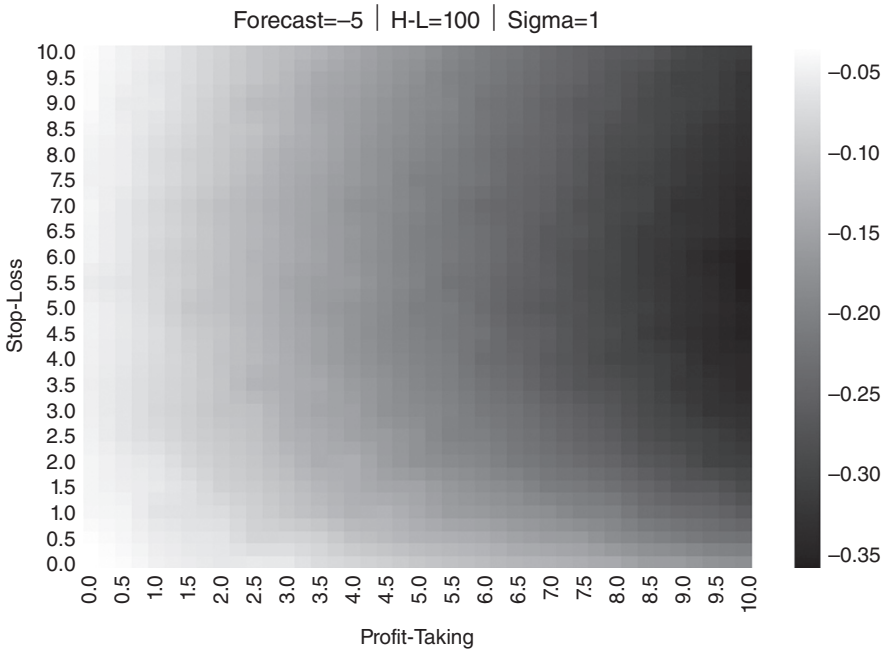


FIGURE 13.20 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-5, 100, 1\}$

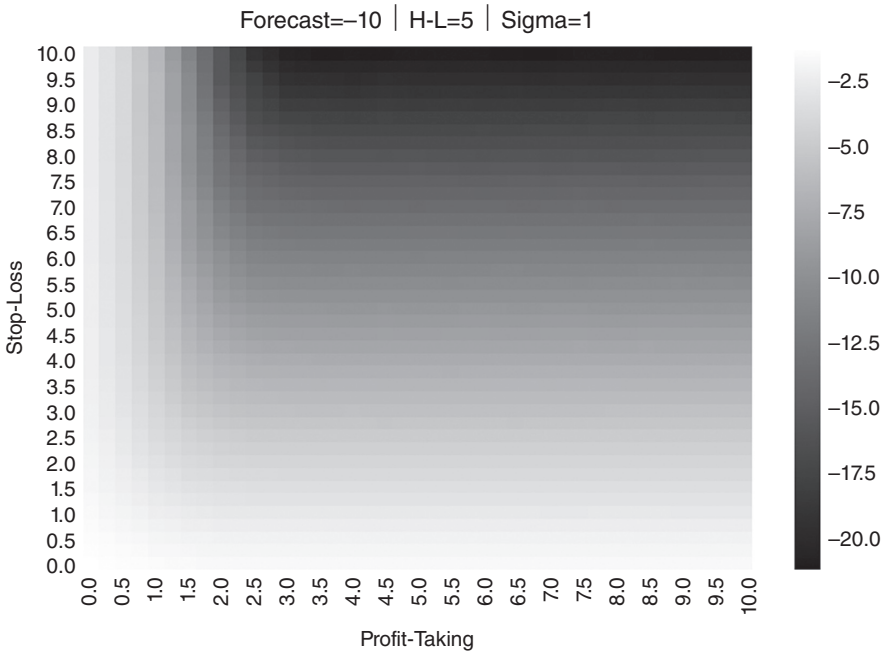


FIGURE 13.21 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-10, 5, 1\}$

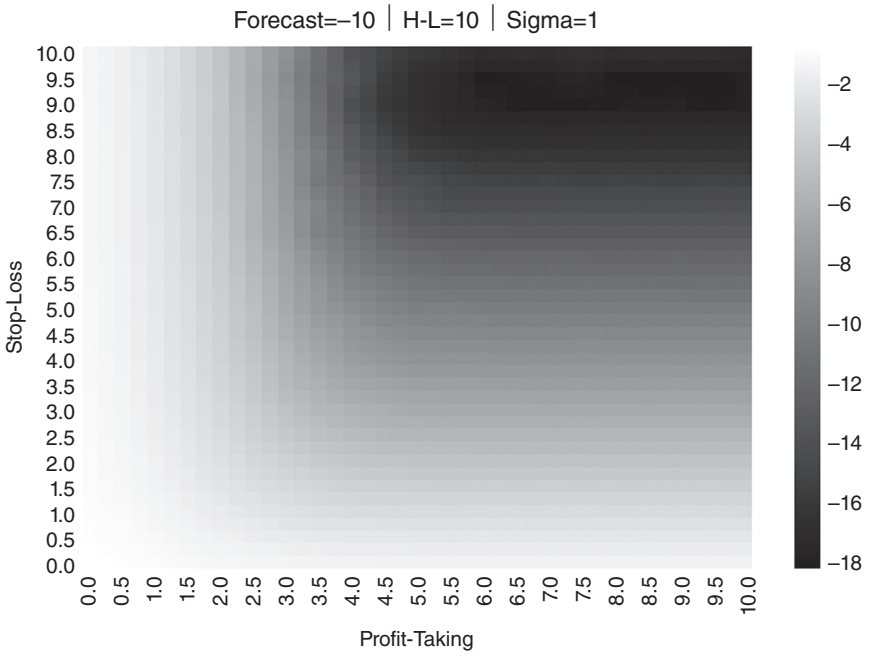


FIGURE 13.22 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-10, 10, 1\}$

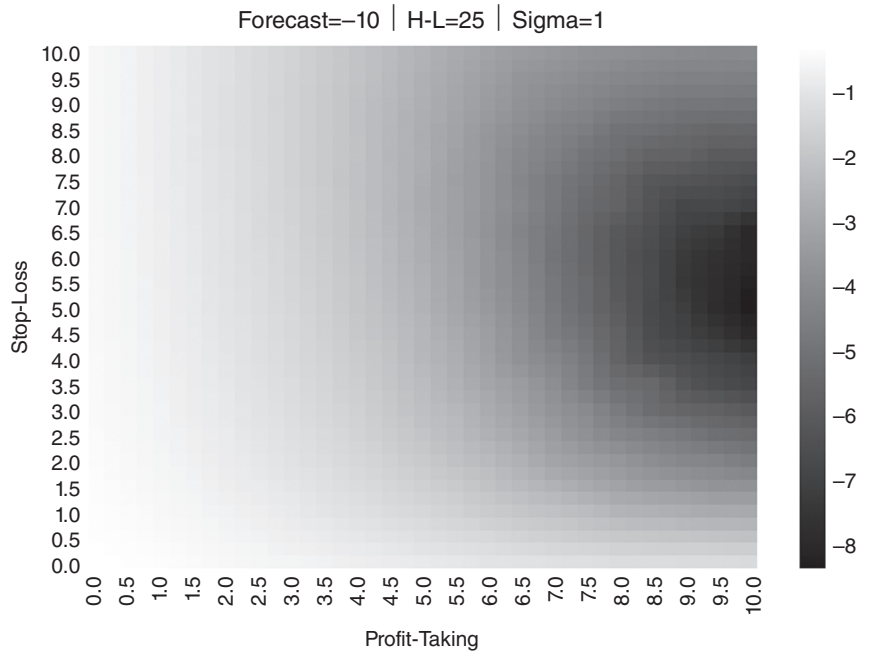


FIGURE 13.23 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \boldsymbol{\sigma}\} = \{-10, 25, 1\}$

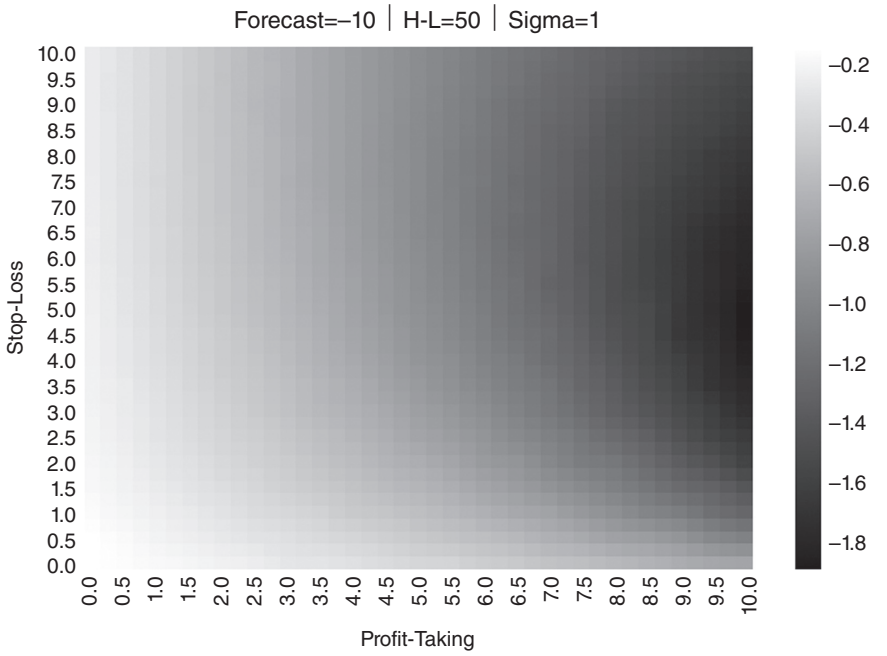


FIGURE 13.24 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-10, 50, 1\}$

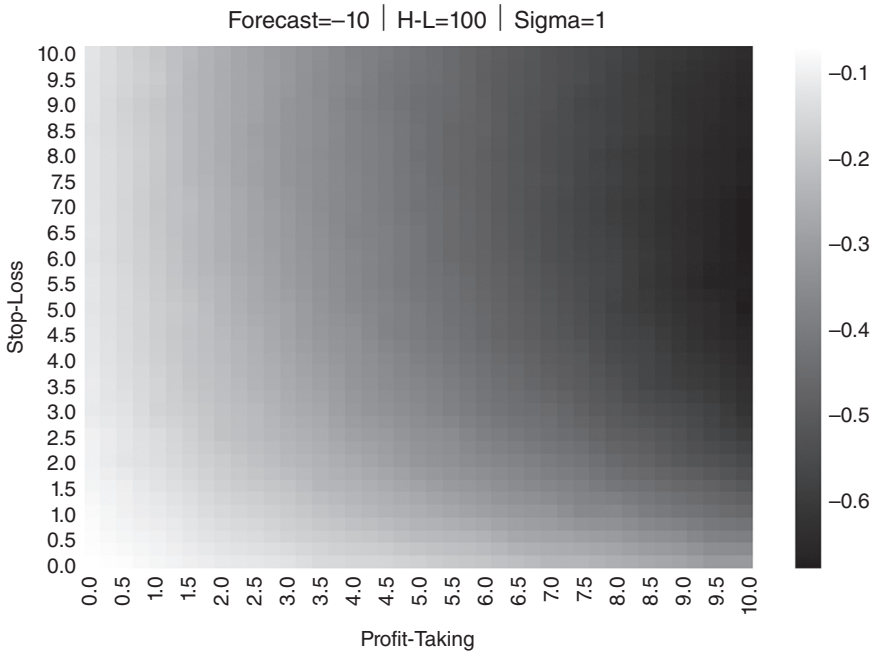


FIGURE 13.25 Heat-map for $\{\mathrm{E}_0[P_{i,T_i}], \tau, \sigma\} = \{-10, 100, 1\}$

SNIPPET 14.1 DERIVING THE TIMING OF BETS FROM A SERIES OF TARGET POSITIONS

```
# A bet takes place between flat positions or position flips
df0=tPos[tPos==0].index
df1=tPos.shift(1);df1=df1[df1!=0].index
bets=df0.intersection(df1) # flattening
df0=tPos.iloc[1:]*tPos.iloc[:-1].values
bets=bets.union(df0[df0<0].index).sort_values() # tPos flips
if tPos.index[-1] not in bets:bets=bets.append(tPos.index[-1:]) # last bet
```

SNIPPET 14.2 IMPLEMENTATION OF A HOLDING PERIOD ESTIMATOR

```
def getHoldingPeriod(tPos):
    # Derive avg holding period (in days) using avg entry time pairing algo
    hp,tEntry=pd.DataFrame(columns=['dT','w']),0.
    pDiff,tDiff=tPos.diff(),(tPos.index-tPos.index[0])/np.timedelta64(1,'D')
    for i in xrange(1,tPos.shape[0]):
        if pDiff.iloc[i]*tPos.iloc[i-1]>=0: # increased or unchanged
            if tPos.iloc[i]!=0:
                tEntry=(tEntry*tPos.iloc[i-1]+tDiff[i]*pDiff.iloc[i])/tPos.iloc[i]
        else: # decreased
            if tPos.iloc[i]*tPos.iloc[i-1]<0: # flip
                hp.loc[tPos.index[i],['dT','w']]=(tDiff[i]-tEntry,abs(tPos.iloc[i-1]))
                tEntry=tDiff[i] # reset entry time
            else:
                hp.loc[tPos.index[i],['dT','w']]=(tDiff[i]-tEntry,abs(pDiff.iloc[i]))
    if hp['w'].sum()>0:hp=(hp['dT']*hp['w']).sum()/hp['w'].sum()
    else:hp=np.nan
    return hp
```

$$r_{i,t} = \frac{\pi_{i,t}}{K_{i,t}}$$

Equation 45

$$\pi_{i,t} = \sum_{j=1}^{J} [(\Delta P_{j,t} + A_{j,t})\theta_{i,j,t-1} + \Delta\theta_{i,j,t}(P_{j,t} - \overline{P}_{j,t-1})]$$

Equation 46

$$K_{i,t} = \sum_{j=1}^{J} \tilde{P}_{j,t-1}\theta_{i,j,t-1} + \max\left\{0, \sum_{j=1}^{J} \overline{\tilde{P}}_{j,t}\Delta\theta_{i,j,t}\right\}$$

Equation 47

$$\varphi_{i,T} = \prod_{t=1}^{T}(1 + r_{i,t})$$

Equation 48

$$R_i = (\varphi_{i,T})^{-y_i} - 1$$

where y_i is the number of years elapsed between $r_{i,1}$ and $r_{i,T}$.

Equation 49

$$r^+ = \{r_t | r_t \geq 0\}_{t=1,\ldots,T}$$
$$r^- = \{r_t | r_t < 0\}_{t=1,\ldots,T}$$

$$w^+ = \left\{ r_t^+ \left(\sum_t r_t^+ \right)^{-1} \right\}_{t=1,\ldots,T}$$
$$w^- = \left\{ r_t^- \left(\sum_t r_t^- \right)^{-1} \right\}_{t=1,\ldots,T}$$

Series 1-4

$$h^+ \equiv \frac{\sum_t \left(w_t^+\right)^2 - ||w^+||^{-1}}{1 - ||w^+||^{-1}} = \left(\frac{\mathrm{E}\left[\left(r_t^+\right)^2\right]}{\mathrm{E}\left[r_t^+\right]^2} - 1 \right) \left(||r^+|| - 1\right)^{-1}$$

Equation 50

$$h^- \equiv \frac{\sum_t \left(w_t^-\right)^2 - ||w^-||^{-1}}{1 - ||w^-||^{-1}} = \left(\frac{\mathrm{E}\left[\left(r_t^-\right)^2\right]}{\mathrm{E}\left[r_t^-\right]^2} - 1 \right) (||r^-|| - 1)^{-1}$$

Equation 51

SNIPPET 14.3 ALGORITHM FOR DERIVING HHI CONCENTRATION

```
rHHIPos=getHHI(ret[ret>=0]) # concentration of positive returns per bet
rHHINeg=getHHI(ret[ret<0]) # concentration of negative returns per bet
tHHI=getHHI(ret.groupby(pd.TimeGrouper(freq='M')).count()) # concentr. bets/month
#———————————————————————————————
def getHHI(betRet):
    if betRet.shape[0]<=2:return np.nan
    wght=betRet/betRet.sum()
    hhi=(wght**2).sum()
    hhi=(hhi-betRet.shape[0]**-1)/(1.-betRet.shape[0]**-1)
    return hhi
```

SNIPPET 14.4 DERIVING THE SEQUENCE OF DD AND TuW

```
def computeDD_TuW(series,dollars=False):
    # compute series of drawdowns and the time under water associated with them
    df0=series.to_frame('pnl')
    df0['hwm']=series.expanding().max()
    df1=df0.groupby('hwm').min().reset_index()
    df1.columns=['hwm','min']
    df1.index=df0['hwm'].drop_duplicates(keep='first').index # time of hwm
    df1=df1[df1['hwm']>df1['min']] # hwm followed by a drawdown
    if dollars:dd=df1['hwm']-df1['min']
    else:dd=1-df1['min']/df1['hwm']
    tuw=((df1.index[1:]-df1.index[:-1])/np.timedelta64(1,'Y')).values# in years
    tuw=pd.Series(tuw,index=df1.index[:-1])
    return dd,tuw
```

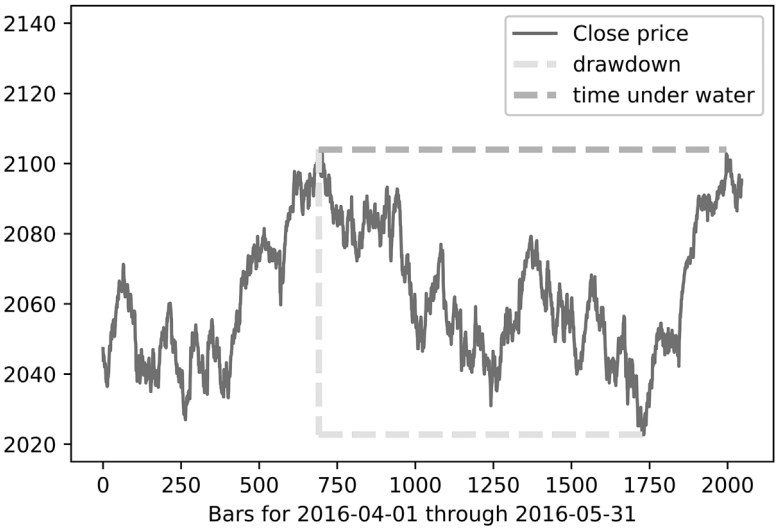


FIGURE 14.1 Examples of drawdown (DD) and time under water + (TuW)

$$\widehat{PSR}\left[SR^*\right] = Z\left[\frac{\left(\widehat{SR} - SR^*\right)\sqrt{T-1}}{\sqrt{1 - \hat{\gamma}_3\widehat{SR} + \frac{\hat{\gamma}_4 - 1}{4}\widehat{SR}^2}}\right]$$

Equation 52

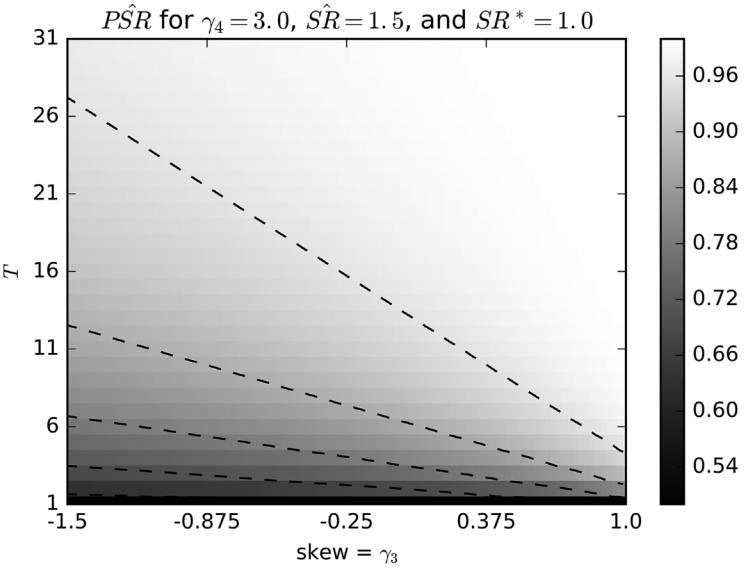


FIGURE 14.2 PSR as a function of skewness and sample length

$$SR^* = \sqrt{V\left[\left\{\widehat{SR}_n\right\}\right]}\left((1-\gamma)Z^{-1}\left[1-\frac{1}{N}\right]+\gamma Z^{-1}\left[1-\frac{1}{N}e^{-1}\right]\right)$$

Equation 53

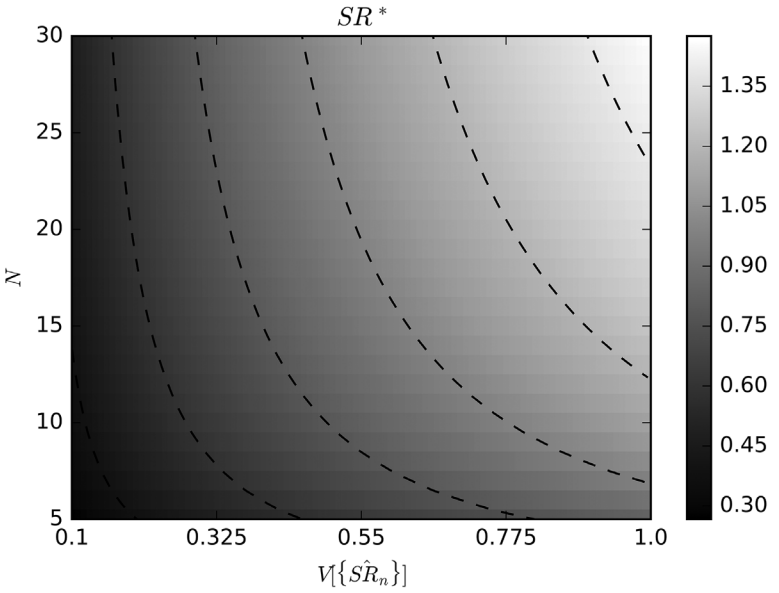


FIGURE 14.3 SR^* as a function of $V[\{\widehat{SR}_n\}]$ and N

TABLE 14.1 The Four Degenerate Cases of Binary Classification

Condition	Collapse	Accuracy	Precision	Recall	F1
Observed all 1s	TN=FP=0	=recall	1	[0,1]	[0,1]
Observed all 0s	TP=FN=0	[0,1]	0	NaN	NaN
Predicted all 1s	TN=FN=0	=precision	[0,1]	1	[0,1]
Predicted all 0s	TP=FP=0	[0,1]	NaN	0	NaN

$$\theta[p,n] = \frac{n\mathrm{E}[X_i]}{\sqrt{n\mathrm{V}[X_i]}} = \underbrace{\frac{2p-1}{2\sqrt{p(1-p)}}}_{\substack{\text{t-value of } p \\ \text{under } \mathrm{H}_0 : p = \frac{1}{2}}} \sqrt{n}$$

Equation 54

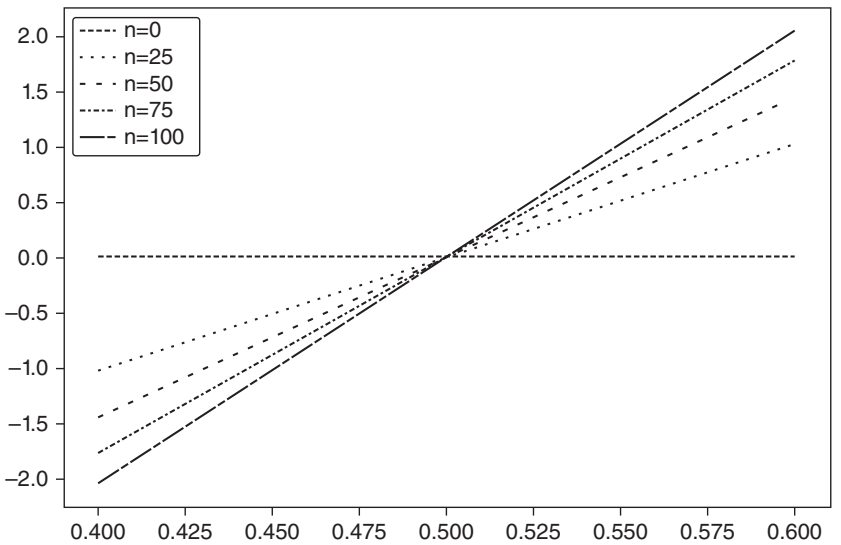


FIGURE 15.1 The relation between precision (x-axis) and sharpe ratio (y-axis) for various bet frequencies (n)

SNIPPET 15.1 TARGETING A SHARPE RATIO AS A FUNCTION OF THE NUMBER OF BETS

```
out,p=[],.55
for i in xrange(1000000):
    rnd=np.random.binomial(n=1,p=p)
    x=(1 if rnd==1 else -1)
    out.append(x)
print np.mean(out),np.std(out),np.mean(out)/np.std(out)
```

$$p = \frac{1}{2}\left(1 + \sqrt{1 - \frac{n}{\theta^2 + n}}\right.$$

Equation 55

$$\theta[p, n, \pi_-, \pi_+] = \frac{n\mathrm{E}[X_i]}{\sqrt{n\mathrm{V}[X_i]}} = \frac{(\pi_+ - \pi_-)p + \pi_-}{(\pi_+ - \pi_-)\sqrt{p(1-p)}}\sqrt{n}$$

Equation 56

$$p = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Equation 57

SNIPPET 15.2 USING THE SymPy LIBRARY FOR SYMBOLIC OPERATIONS

```
>>> from sympy import *
>>> init_printing(use_unicode=False,wrap_line=False,no_global=True)
>>> p,u,d=symbols('p u d')
>>> m2=p*u**2+(1-p)*d**2
>>> m1=p*u+(1-p)*d
>>> v=m2-m1**2
>>> factor(v)
```

SNIPPET 15.3 COMPUTING THE IMPLIED PRECISION

```
def binHR(sl,pt,freq,tSR):
    '''
    Given a trading rule characterized by the parameters {sl,pt,freq},
    what's the min precision p required to achieve a Sharpe ratio tSR?
    1) Inputs
    sl: stop loss threshold
    pt: profit taking threshold
    freq: number of bets per year
    tSR: target annual Sharpe ratio
    2) Output
    p: the min precision rate p required to achieve tSR
    '''
    a=(freq+tSR**2)*(pt-sl)**2
    b=(2*freq*sl-tSR**2*(pt-sl))*(pt-sl)
    c=freq*sl**2
    p=(-b+(b**2-4*a*c)**.5)/(2.*a)
    return p
```

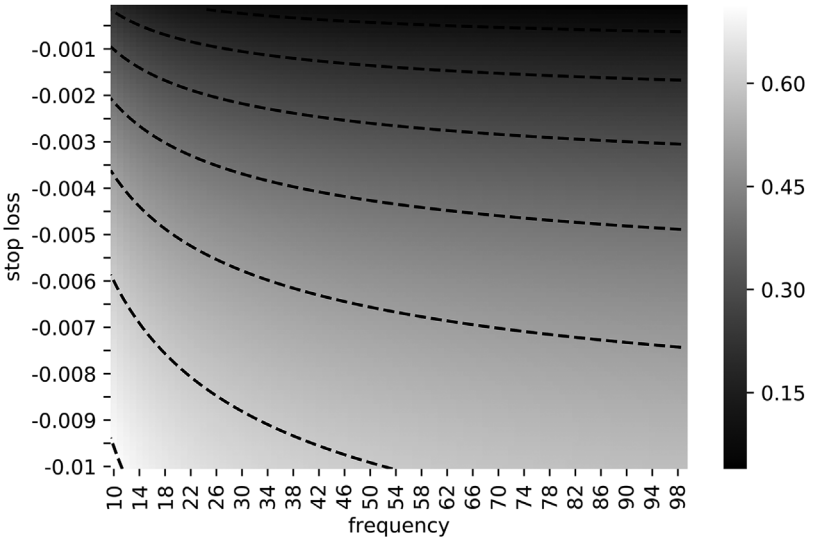


FIGURE 15.2 Heat-map of the implied precision as a function of n and π_-, with $\pi_+ = 0.1$ and $\theta^* = 1.5$

SNIPPET 15.4 COMPUTING THE IMPLIED BETTING FREQUENCY

```
def binFreq(sl,pt,p,tSR):
    '''
    Given a trading rule characterized by the parameters {sl,pt,freq},
    what's the number of bets/year needed to achieve a Sharpe ratio
    tSR with precision rate p?
    Note: Equation with radicals, check for extraneous solution.
    1) Inputs
    sl: stop loss threshold
    pt: profit taking threshold
    p: precision rate p
    tSR: target annual Sharpe ratio
    2) Output
    freq: number of bets per year needed
    '''
    freq=(tSR*(pt-sl))**2*p*(1-p)/((pt-sl)*p+sl)**2 # possible extraneous
    if not np.isclose(binSR(sl,pt,freq,p),tSR):return
    return freq
```

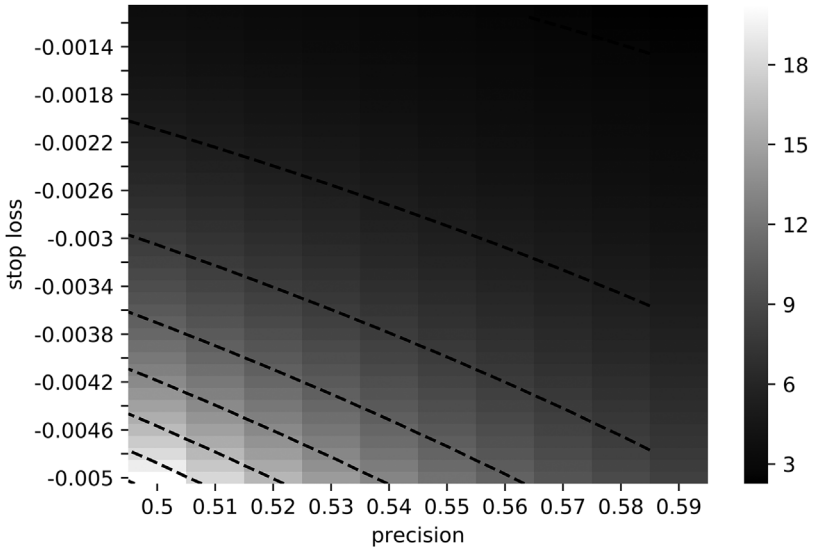


FIGURE 15.3 Implied frequency as a function of p and, with = 0.1 and = 1.5

SNIPPET 15.5 CALCULATING THE STRATEGY RISK IN PRACTICE

```
import numpy as np,scipy.stats as ss
#-------------------------------------------
def mixGaussians(mu1,mu2,sigma1,sigma2,prob1,nObs):
    # Random draws from a mixture of gaussians
    ret1=np.random.normal(mu1,sigma1,size=int(nObs*prob1))
    ret2=np.random.normal(mu2,sigma2,size=int(nObs)-ret1.shape[0])
    ret=np.append(ret1,ret2,axis=0)
    np.random.shuffle(ret)
    return ret
#-------------------------------------------
def probFailure(ret,freq,tSR):
    # Derive probability that strategy may fail
    rPos,rNeg=ret[ret>0].mean(),ret[ret<=0].mean()
    p=ret[ret>0].shape[0]/float(ret.shape[0])
    thresP=binHR(rNeg,rPos,freq,tSR)
    risk=ss.norm.cdf(thresP,p,p*(1-p)) # approximation to bootstrap
    return risk
#-------------------------------------------
def main():
    #1) Parameters
    mu1,mu2,sigma1,sigma2,prob1,nObs=.05,-.1,.05,.1,.75,2600
    tSR,freq=2.,260
    #2) Generate sample from mixture
    ret=mixGaussians(mu1,mu2,sigma1,sigma2,prob1,nObs)
    #3) Compute prob failure
    probF=probFailure(ret,freq,tSR)
    print 'Prob strategy will fail',probF
    return
#-------------------------------------------
if __name__=='__main__':main()
```

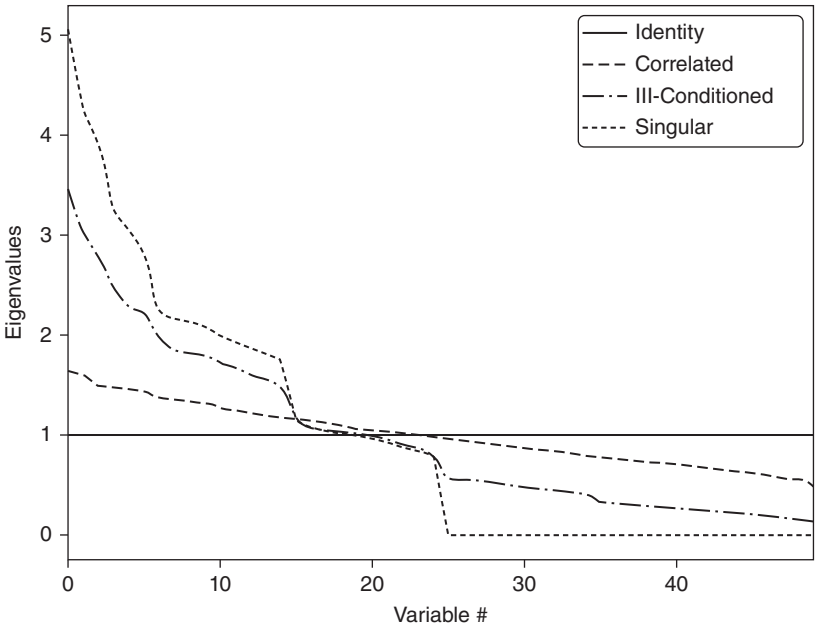


FIGURE 16.1 Visualization of Markowitz's curse
A diagonal correlation matrix has the lowest condition number. As we add correlated investments, the maximum eigenvalue is greater and the minimum eigenvalue is lower. The condition number rises quickly, leading to unstable inverse correlation matrices. At some point, the benefits of diversification are more than offset by estimation errors.

16.4.1 Tree Clustering

Consider a *TxN* matrix of observations *X*, such as returns series of *N* variables over *T* periods. We would like to combine these *N* column-vectors into a hierarchical structure of clusters, so that allocations can flow downstream through a tree graph.

First, we compute an *NxN* correlation matrix with entries $\rho = \{\rho_{i,j}\}_{i,j=1,\dots,N}$, where $\rho_{i,j} = \rho[X_i, X_j]$. We define the distance measure $d : (X_i, X_j) \subset B \to \mathbb{R} \in [0, 1]$, $d_{i,j} = d[X_i, X_j] = \sqrt{\frac{1}{2}(1 - \rho_{i,j})}$, where B is the Cartesian product of items in $\{1, \dots, i, \dots, N\}$. This allows us to compute an *NxN* distance matrix $D = \{d_{i,j}\}_{i,j=1,\dots,N}$. Matrix D is a proper metric space (see Appendix 16.A.1 for a proof), in the sense that $d[x, y] \geq 0$ (non-negativity), $d[x, y] = 0 \Leftrightarrow X = Y$ (coincidence), $d[x, y] = d[Y, X]$ (symmetry), and $d[X, Z] \leq d[x, y] + d[Y, Z]$ (sub-additivity). See Example 16.1.

$$\{\rho_{i,j}\} = \begin{bmatrix} 1 & .7 & .2 \\ .7 & 1 & -.2 \\ .2 & -.2 & 1 \end{bmatrix} \to \{d_{i,j}\} = \begin{bmatrix} 0 & .3873 & .6325 \\ .3873 & 0 & .7746 \\ .6325 & .7746 & 0 \end{bmatrix}$$

Example 16.1 Encoding a correlation matrix ρ as a distance matrix D

Second, we compute the Euclidean distance between any two column-vectors of D, $\tilde{d} : (D_i, D_j) \subset B \to \mathbb{R} \in [0, \sqrt{N}]$, $\tilde{d}_{i,j} = \tilde{d}[D_i, D_j] = \sqrt{\sum_{n=1}^{N} (d_{n,i} - d_{n,j})^2}$. Note

Equation 58

the difference between distance metrics $d_{i,j}$ and $\tilde{d}_{i,j}$. Whereas $d_{i,j}$ is defined on column-vectors of X, $\tilde{d}_{i,j}$ is defined on column-vectors of D (a distance of distances). Therefore, $\tilde{d}$ is a distance defined over the entire metric space D, as each $\tilde{d}_{i,j}$ is a function of the entire correlation matrix (rather than a particular cross-correlation pair). See Example 16.2.

$$\{d_{i,j}\} = \begin{bmatrix} 0 & .3873 & .6325 \\ .3873 & 0 & .7746 \\ .6325 & .7746 & 0 \end{bmatrix} \to \{\tilde{d}_{i,j}\}_{i,j=\{1,2,3\}} = \begin{bmatrix} 0 & .5659 & .9747 \\ .5659 & 0 & 1.1225 \\ .9747 & 1.1225 & 0 \end{bmatrix}$$

Example 16.2 Euclidean distance of correlation distances

Third, we cluster together the pair of columns (i^*, j^*) such that $(i^*, j^*) = \text{argmin}(i,j)_{i \neq j}\{\tilde{d}_{i,j}\}$, and denote this cluster as $u[1]$. See Example 16.3.

$$\{\tilde{d}_{i,j}\}_{i,j=\{1,2,3\}} = \begin{bmatrix} 0 & .5659 & .9747 \\ .5659 & 0 & 1.1225 \\ .9747 & 1.1225 & 0 \end{bmatrix} \to u[1] = (1, 2)$$

Example 16.3 Clustering items

Fourth, we need to define the distance between a newly formed cluster $u[1]$ and the single (unclustered) items, so that $\{\tilde{d}_{i,j}\}$ may be updated. In hierarchical clustering analysis, this is known as the "linkage criterion." For example, we can define the distance between an item i of $\tilde{d}$ and the new cluster $u[1]$ as $\dot{d}_{i,u[1]} = \min[\{\tilde{d}_{i,j}\}_{j \in u[1]}]$ (the nearest point algorithm). See Example 16.4.

$$u[1] = (1,2) \rightarrow \{\dot{d}_{i,u[1]}\} = \begin{bmatrix} \min[0, .5659] \\ \min[.5659, 0] \\ \min[.9747, 1.1225] \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ .9747 \end{bmatrix}$$

Example 16.4 Updating matrix $\{\tilde{d}_{i,j}\}$ with the new cluster u

Fifth, matrix $\{\tilde{d}_{i,j}\}$ is updated by appending $\dot{d}_{i,u[1]}$ and dropping the clustered columns and rows $j \in u[1]$. See Example 16.5.

$$\{\tilde{d}_{i,j}\}_{i,j=\{1,2,3,4\}} = \begin{bmatrix} 0 & .5659 & .9747 & 0 \\ .5659 & 0 & 1.1225 & 0 \\ .9747 & 1.1225 & 0 & .9747 \\ 0 & 0 & .9747 & 0 \end{bmatrix}$$

$$\{\tilde{d}_{i,j}\}_{i,j=\{3,4\}} = \begin{bmatrix} 0 & .9747 \\ .9747 & 0 \end{bmatrix}$$

Example 16.5 Updating matrix $\{\tilde{d}_{i,j}\}$ with the new cluster u

Sixth, applied recursively, steps 3, 4, and 5 allow us to append $N - 1$ such clusters to matrix D, at which point the final cluster contains all of the original items, and the clustering algorithm stops. See Example 16.6.

$$\{\tilde{d}_{i,j}\}_{i,j=\{3,4\}} = \begin{bmatrix} 0 & .9747 \\ .9747 & 0 \end{bmatrix} \rightarrow u[2] = (3,4) \rightarrow \text{Stop}$$

Example 16.6 Recursion in search of remaining clusters

Figure 16.3 displays the clusters formed at each iteration for this example, as well as the distances $\tilde{d}_{i^*,j^*}$ that triggered every cluster (third step). This procedure can be applied to a wide array of distance metrics $d_{i,j}$, $\tilde{d}_{i,j}$ and $\dot{d}_{i,u}$, beyond those illustrated in this chapter. See Rokach and Maimon [2005] for alternative metrics, the discussion on Fiedler's vector and Stewart's spectral clustering method in Brualdi [2010], as well as algorithms in the scipy library.[2] Snippet 16.1 provides an example of tree clustering using scipy functionality.

This stage allows us to define a linkage matrix as an $(N-1)x4$ matrix with structure $Y = \{(y_{m,1}, y_{m,2}, y_{m,3}, y_{m,4})\}_{m=1,\ldots,N-1}$ (i.e., with one 4-tuple per cluster). Items $(y_{m,1}, y_{m,2})$ report the constituents. Item $y_{m,3}$ reports the distance between $y_{m,1}$ and $y_{m,2}$, that is $y_{m,3} = \tilde{d}_{y_{m,1},y_{m,2}}$. Item $y_{m,4} \leq N$ reports the number of original items included in cluster m.

[2] For additional metrics see:

http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html

http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.cluster.hierarchy.linkage.html

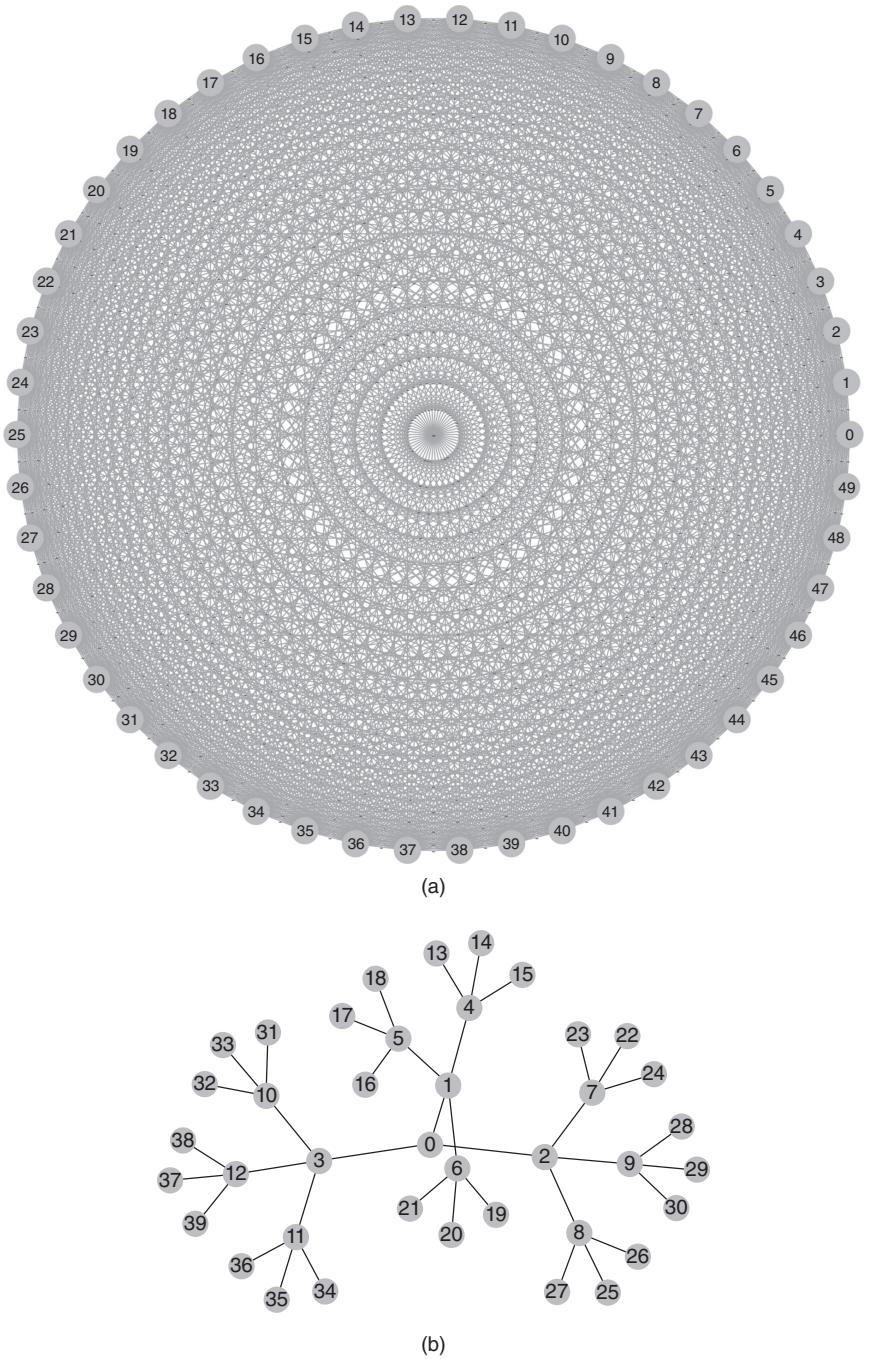


FIGURE 16.2 The complete-graph (top) and the tree-graph (bottom) structures
Correlation matrices can be represented as complete graphs, which lack the notion of hierarchy: Each investment is substitutable with another. In contrast, tree structures incorporate hierarchical relationships.

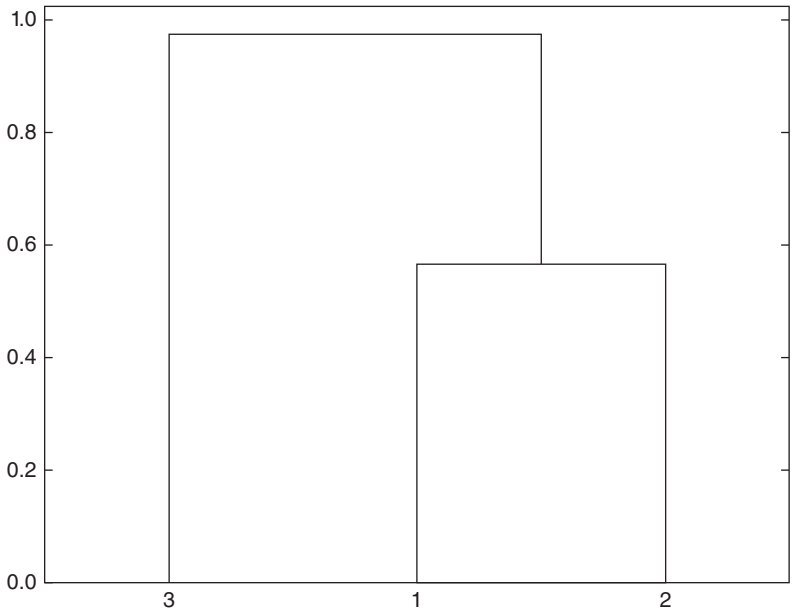


FIGURE 16.3 Sequence of cluster formation
A tree structure derived from our numerical example, here plotted as a dendogram. The y-axis measures the distance between the two merging leaves.

SNIPPET 16.1 TREE CLUSTERING USING SCIPY FUNCTIONALITY

```
import scipy.cluster.hierarchy as sch
import numpy as np
import pandas as pd
cov,corr=x.cov(),x.corr()
dist=((1-corr)/2.)**.5 # distance matrix
link=sch.linkage(dist,'single') # linkage matrix
```

SNIPPET 16.2 QUASI-DIAGONALIZATION

```
def getQuasiDiag(link):
    # Sort clustered items by distance
    link=link.astype(int)
    sortIx=pd.Series([link[-1,0],link[-1,1]])
    numItems=link[-1,3] # number of original items
    while sortIx.max()>=numItems:
        sortIx.index=range(0,sortIx.shape[0]*2,2) # make space
        df0=sortIx[sortIx>=numItems] # find clusters
        i=df0.index;j=df0.values-numItems
        sortIx[i]=link[j,0] # item 1
        df0=pd.Series(link[j,1],index=i+1)
        sortIx=sortIx.append(df0) # item 2
        sortIx=sortIx.sort_index() # re-sort
        sortIx.index=range(sortIx.shape[0]) # re-index
    return sortIx.tolist()
```

SNIPPET 16.3 RECURSIVE BISECTION

```
def getRecBipart(cov,sortIx):
    # Compute HRP alloc
    w=pd.Series(1,index=sortIx)
    cItems=[sortIx] # initialize all items in one cluster
    while len(cItems)>0:
        cItems=[i[j:k] for i in cItems for j,k in ((0,len(i)/2),\
            (len(i)/2,len(i))) if len(i)>1] # bi-section
        for i in xrange(0,len(cItems),2): # parse in pairs
            cItems0=cItems[i] # cluster 1
            cItems1=cItems[i+1] # cluster 2
            cVar0=getClusterVar(cov,cItems0)
            cVar1=getClusterVar(cov,cItems1)
            alpha=1-cVar0/(cVar0+cVar1)
            w[cItems0]*=alpha # weight 1
            w[cItems1]*=1-alpha # weight 2
    return w
```

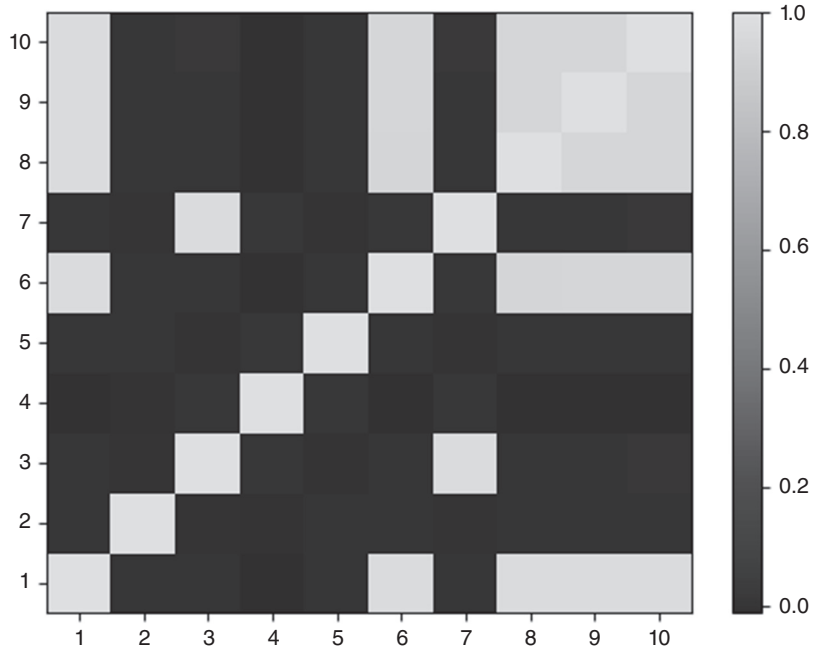


FIGURE 16.4 Heat-map of original covariance matrix
This correlation matrix has been computed using function `generateData` from snippet 16.4 (see Section 16.A.3). The last five columns are partially correlated to some of the first five series.

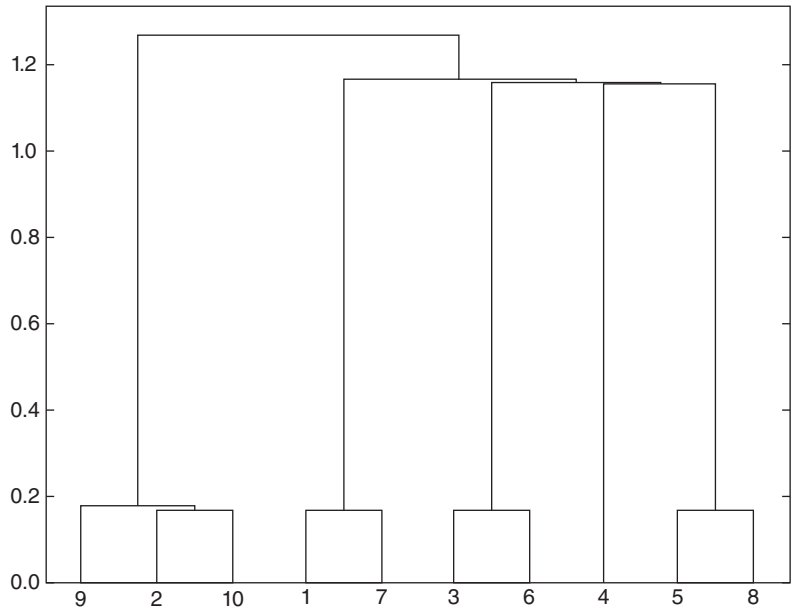


FIGURE 16.5 Dendogram of cluster formation
The clustering procedure has correctly identified that series 9 and 10 were perturbations of series 2, hence $(9, 2, 10)$ are clustered together. Similarly, 7 is a perturbation of 1, 6 is a perturbation of 3, and 8 is a perturbation of 5. The only original item that was not perturbated is 4, and that is the one item for which the clustering algorithm found no similarity.

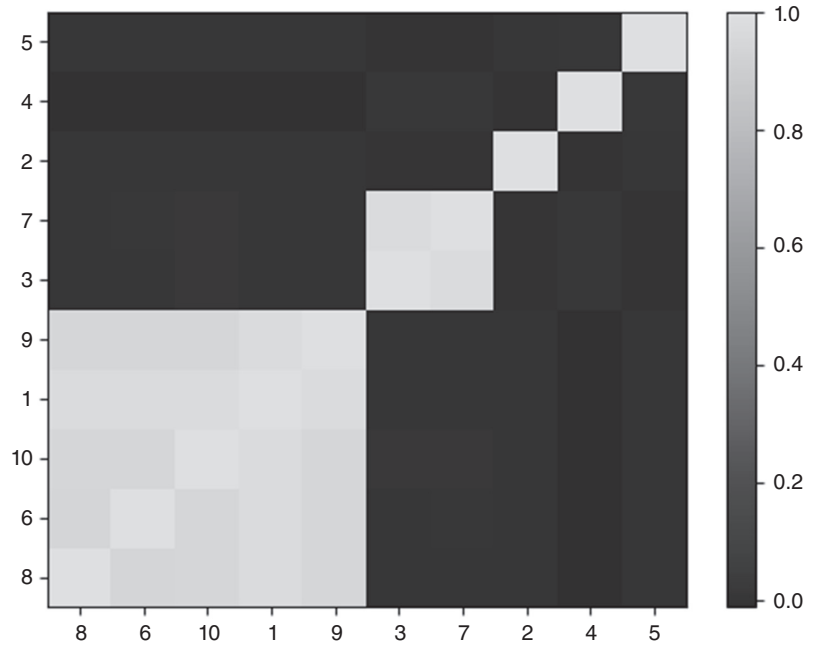


FIGURE 16.6 Clustered covariance matrix
Stage 2 quasi-diagonalizes the correlation matrix, in the sense that the largest values lie along the diagonal. However, unlike PCA or similar procedures, HRP does not require a change of basis. HRP solves the allocation problem robustly, while working with the original investments.

TABLE 16.1 A Comparison of Three Allocations

Weight #	CLA	HRP	IVP
1	14.44%	7.00%	10.36%
2	19.93%	7.59%	10.28%
3	19.73%	10.84%	10.36%
4	19.87%	19.03%	10.25%
5	18.68%	9.72%	10.31%
6	0.00%	10.19%	9.74%
7	5.86%	6.62%	9.80%
8	1.49%	9.10%	9.65%
9	0.00%	7.12%	9.64%
10	0.00%	12.79%	9.61%

A characteristic outcome of the three methods studied: CLA concentrates weights on a few investments, hence becoming exposed to idiosyncratic shocks. IVP evenly spreads weights through all investments, ignoring the correlation structure. This makes it vulnerable to systemic shocks. HRP finds a compromise between diversifying across all investments and diversifying across cluster, which makes it more resilient against both types of shocks.

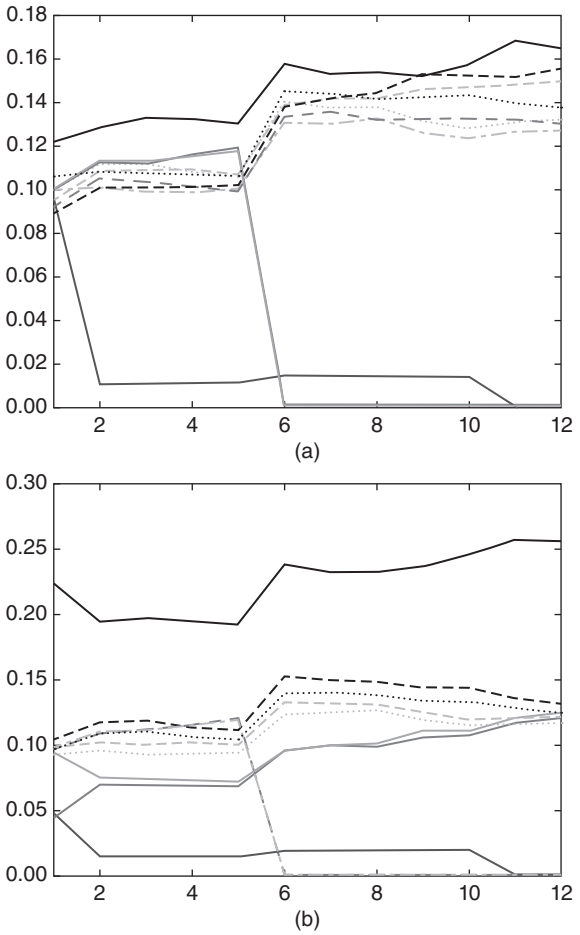


FIGURE 16.7 (a) Time series of allocations for IVP.
Between the first and second rebalance, one investment receives an idiosyncratic shock, which increases its variance. IVP's response is to reduce the allocation to that investment, and spread that former exposure across all other investments. Between the fifth and sixth rebalance, two investments are affected by a common shock. IVP's response is the same. As a result, allocations among the seven unaffected investments grow over time, regardless of their correlation.
(b) Time series of allocations for HRP
HRP's response to the idiosyncratic shock is to reduce the allocation to the affected investment, and use that reduced amount to increase the allocation to a correlated investment that was unaffected. As a response to the common shock, HRP reduces allocation to the affected investments and increases allocation to uncorrelated ones (with lower variance).
(c) Time series of allocations for CLA
CLA allocations respond erratically to idiosyncratic and common shocks. If we had taken into account rebalancing costs, CLA's performance would have been very negative.

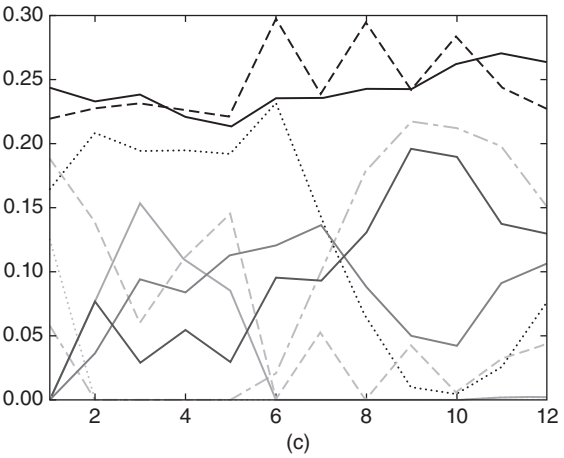


FIGURE 16.7 (*Continued*)

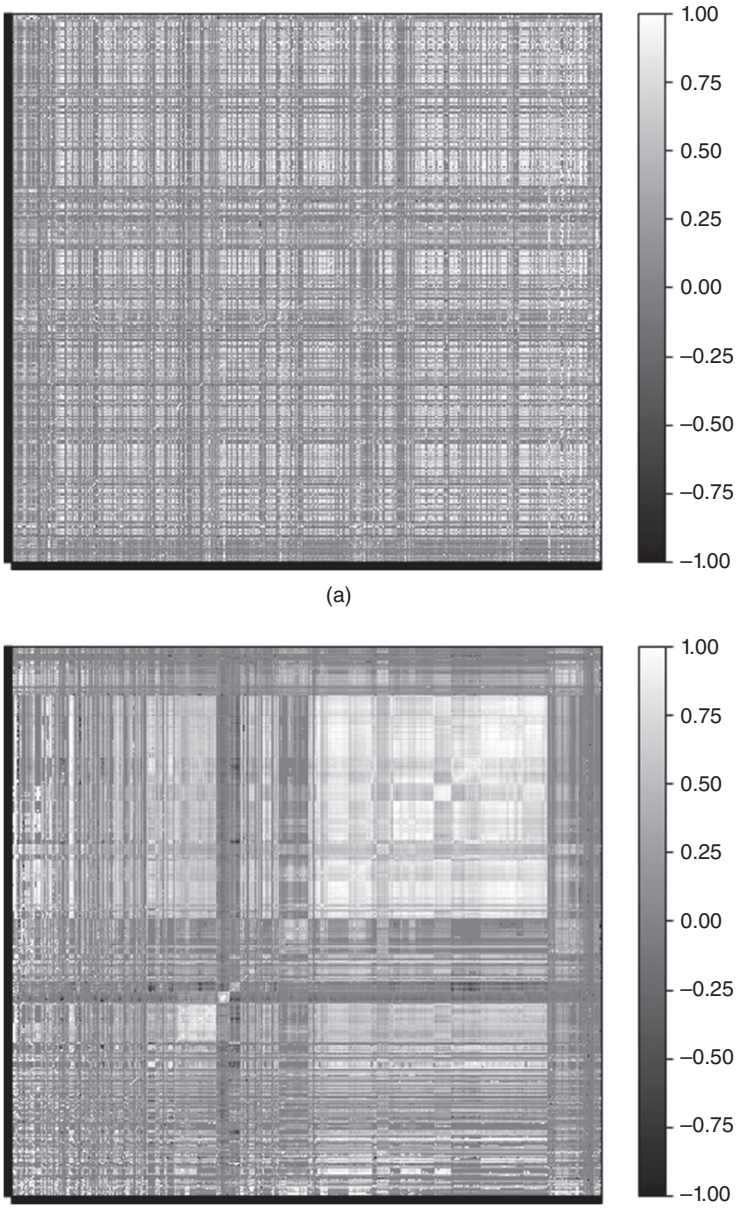


FIGURE 16.8 Correlation matrix before and after clustering
The methodology described in this chapter can be applied to problems beyond optimization. For example, a PCA analysis of a large fixed income universe suffers the same drawbacks we described for CLA. Small-data techniques developed decades and centuries ago (factor models, regression analysis, econometrics) fail to recognize the hierarchical nature of financial big data.

CHAPTER 16 APPENDICES

16.A.1 CORRELATION-BASED METRIC

Consider two real-valued vectors X, Y of size T, and a correlation variable $\rho[x, y]$, with the only requirement that $\sigma[x, y] = \rho[x, y]\sigma[X]\sigma[Y]$, where $\sigma[x, y]$ is the covariance between the two vectors, and $\sigma[.]$ is the standard deviation. Note that Pearson's is not the only correlation to satisfy these requirements.

Let us prove that $d[x, y] = \sqrt{\frac{1}{2}(1 - \rho[x, y])}$ is a true metric. First, the Euclidean distance between the two vectors is $d[x, y] = \sqrt{\sum_{t=1}^{T}(X_t - Y_t)^2}$. Second, we z-standardize those vectors as $x = \frac{X-\bar{X}}{\sigma[X]}$, $y = \frac{Y-\bar{Y}}{\sigma[Y]}$. Consequently, $0 \leq \rho[x, y] = \rho[x, y]$. Third, we derive the Euclidean distance $d[x, y]$ as,

$$d[x, y] = \sqrt{\sum_{t=1}^{T}(x_t - y_t)^2} = \sqrt{\sum_{t=1}^{T}x_t^2 + \sum_{t=1}^{T}y_t^2 - 2\sum_{t=1}^{T}x_t y_t}$$

$$= \sqrt{T + T - 2T\sigma[x, y]} = \sqrt{2T\left(1 - \underbrace{\rho[x, y]}_{=\rho[x,y]}\right)} = \sqrt{4T}d[x, y]$$

In other words, the distance $d[x, y]$ is a linear multiple of the Euclidean distance between the vectors $\{X, Y\}$ after z-standardization, hence it inherits the true-metric properties of the Euclidean distance.

Similarly, we can prove that $d[x, y] = \sqrt{1 - |\rho[x, y]|}$ descends to a true metric on the $\mathbb{Z}/2\mathbb{Z}$ quotient. In order to do that, we redefine $y = \frac{Y-\bar{Y}}{\sigma[Y]}\text{sgn}[\rho[x, y]]$, where sgn $[.]$ is the sign operator, so that $0 \leq \rho[x, y] = |\rho[x, y]|$. Then,

$$d[x, y] = \sqrt{2T\left(1 - \underbrace{\rho[x, y]}_{=|\rho[x,y]|}\right)} = \sqrt{2T}d[x, y]$$

16.A.2 INVERSE VARIANCE ALLOCATION

Stage 3 (see Section 16.4.3) splits a weight in inverse proportion to the subset's variance. We now prove that such allocation is optimal when the covariance matrix is diagonal. Consider the standard quadratic optimization problem of size N,

$$\min_{\omega} \omega' V \omega$$
$$\text{s.t. : } \omega' a = 1_I$$

with solution $\omega = \frac{V^{-1}\, a}{a' V^{-1}\, a}$. For the characteristic vector $a = 1_N$, the solution is the minimum variance portfolio. If V is diagonal, $\omega_n = \frac{V_{n,n}^{-1}}{\sum_{i=1}^{N} V_{i,i}^{-1}}$. In the particular case of $N = 2$, $\omega_1 = \frac{\frac{1}{V_{1,1}}}{\frac{1}{V_{1,1}} + \frac{1}{V_{2,2}}} = 1 - \frac{V_{1,1}}{V_{1,1} + V_{2,2}}$, which is how stage 3 splits a weight between two bisections of a subset.

16.A.3 REPRODUCING THE NUMERICAL EXAMPLE

Snippet 16.4 can be used to reproduce our results and simulate additional numerical examples. Function `generateData` produces a matrix of time series where a number `size0` of vectors are uncorrelated, and a number `size1` of vectors are correlated. The reader can change the `np.random.seed` in `generateData` to run alternative examples and gain an intuition of how HRP works. Scipy's function `linkage` can be used to perform stage 1 (Section 16.4.1), function `getQuasiDiag` performs stage 2 (Section 16.4.2), and function `getRecBipart` carries out stage 3 (Section 16.4.3).

SNIPPET 16.4 FULL IMPLEMENTATION OF THE HRP ALGORITHM

```
import matplotlib.pyplot as mpl
import scipy.cluster.hierarchy as sch,random,numpy as np,pandas as pd
#------------------------------------------------
def getIVP(cov,**kargs):
    # Compute the inverse-variance portfolio
    ivp=1./np.diag(cov)
    ivp/=ivp.sum()
    return ivp
#------------------------------------------------
def getClusterVar(cov,cItems):
    # Compute variance per cluster
    cov_=cov.loc[cItems,cItems] # matrix slice
    w_=getIVP(cov_).reshape(-1,1)
    cVar=np.dot(np.dot(w_.T,cov_),w_)[0,0]
    return cVar
#------------------------------------------------
def getQuasiDiag(link):
    # Sort clustered items by distance
    link=link.astype(int)
    sortIx=pd.Series([link[-1,0],link[-1,1]])
    numItems=link[-1,3] # number of original items
    while sortIx.max()>=numItems:
        sortIx.index=range(0,sortIx.shape[0]*2,2) # make space
        df0=sortIx[sortIx>=numItems] # find clusters
        i=df0.index;j=df0.values-numItems
        sortIx[i]=link[j,0] # item 1
```

```
        df0=pd.Series(link[j,1],index=i+1)
        sortIx=sortIx.append(df0) # item 2
        sortIx=sortIx.sort_index() # re-sort
        sortIx.index=range(sortIx.shape[0]) # re-index
    return sortIx.tolist()
#-------------------------------------------------
def getRecBipart(cov,sortIx):
    # Compute HRP alloc
    w=pd.Series(1,index=sortIx)
    cItems=[sortIx] # initialize all items in one cluster
    while len(cItems)>0:
        cItems=[i[j:k] for i in cItems for j,k in ((0,len(i)/2), \
            (len(i)/2,len(i))) if len(i)>1] # bi-section
        for i in xrange(0,len(cItems),2): # parse in pairs
            cItems0=cItems[i] # cluster 1
            cItems1=cItems[i+1] # cluster 2
            cVar0=getClusterVar(cov,cItems0)
            cVar1=getClusterVar(cov,cItems1)
            alpha=1-cVar0/(cVar0+cVar1)
            w[cItems0]*=alpha # weight 1
            w[cItems1]*=1-alpha # weight 2
    return w
#-------------------------------------------------
def correlDist(corr):
    # A distance matrix based on correlation, where 0<=d[i,j]<=1
    # This is a proper distance metric
    dist=((1-corr)/2.)**.5 # distance matrix
    return dist
#-------------------------------------------------
def plotCorrMatrix(path,corr,labels=None):
    # Heatmap of the correlation matrix
    if labels is None:labels=[]
    mpl.pcolor(corr)
    mpl.colorbar()
    mpl.yticks(np.arange(.5,corr.shape[0]+.5),labels)
    mpl.xticks(np.arange(.5,corr.shape[0]+.5),labels)
    mpl.savefig(path)
    mpl.clf();mpl.close() # reset pylab
    return
#-------------------------------------------------
def generateData(nObs,size0,size1,sigma1):
    # Time series of correlated variables
    #1) generating some uncorrelated data
    np.random.seed(seed=12345);random.seed(12345)
    x=np.random.normal(0,1,size=(nObs,size0)) # each row is a variable
    #2) creating correlation between the variables
    cols=[random.randint(0,size0-1) for i in xrange(size1)]
    y=x[:,cols]+np.random.normal(0,sigma1,size=(nObs,len(cols)))
    x=np.append(x,y,axis=1)
```

```
    x=pd.DataFrame(x,columns=range(1,x.shape[1]+1))
    return x,cols
#-----------------------------------------
def main():
    #1) Generate correlated data
    nObs,size0,size1,sigma1=10000,5,5,.25
    x,cols=generateData(nObs,size0,size1,sigma1)
    print [(j+1,size0+i) for i,j in enumerate(cols,1)]
    cov,corr=x.cov(),x.corr()
    #2) compute and plot correl matrix
    plotCorrMatrix('HRP3_corr0.png',corr,labels=corr.columns)
    #3) cluster
    dist=correlDist(corr)
    link=sch.linkage(dist,'single')
    sortIx=getQuasiDiag(link)
    sortIx=corr.index[sortIx].tolist() # recover labels
    df0=corr.loc[sortIx,sortIx] # reorder
    plotCorrMatrix('HRP3_corr1.png',df0,labels=df0.columns)
    #4) Capital allocation
    hrp=getRecBipart(cov,sortIx)
    print hrp
    return
#-----------------------------------------
if __name__=='__main__':main()
```

16.A.4 REPRODUCING THE MONTE CARLO EXPERIMENT

Snippet 16.5 implements Monte Carlo experiments on three allocation methods: HRP, CLA, and IVP. All libraries are standard except for HRP, which is provided in Appendix 16.A.3, and CLA, which can be found in Bailey and López de Prado [2013]. The subroutine `generateData` simulates the correlated data, with two types of random shocks: common to various investments and specific to a single investment. There are two shocks of each type, one positive and one negative. The variables for the experiments are set as arguments of `hrpMC`. They were chosen arbitrarily, and the user can experiment with alternative combinations.

SNIPPET 16.5 MONTE CARLO EXPERIMENT ON HRP OUT-OF-SAMPLE PERFORMANCE

```
import scipy.cluster.hierarchy as sch,random,numpy as np,pandas as pd,CLA
from HRP import correlDist,getIVP,getQuasiDiag,getRecBipart
#-----------------------------------------
def generateData(nObs,sLength,size0,size1,mu0,sigma0,sigma1F):
    # Time series of correlated variables
    #1) generate random uncorrelated data
```

```
    x=np.random.normal(mu0,sigma0,size=(nObs,size0))
    #2) create correlation between the variables
    cols=[random.randint(0,size0-1) for i in xrange(size1)]
    y=x[:,cols]+np.random.normal(0,sigma0*sigma1F,size=(nObs,len(cols)))
    x=np.append(x,y,axis=1)
    #3) add common random shock
    point=np.random.randint(sLength,nObs-1,size=2)
    x[np.ix_(point,[cols[0],size0])]=np.array([[-.5,-.5],[2,2]])
    #4) add specific random shock
    point=np.random.randint(sLength,nObs-1,size=2)
    x[point,cols[-1]]=np.array([-.5,2])
    return x,cols
    #-----------------------------------------
def getHRP(cov,corr):
    # Construct a hierarchical portfolio
    corr,cov=pd.DataFrame(corr),pd.DataFrame(cov)
    dist=correlDist(corr)
    link=sch.linkage(dist,'single')
    sortIx=getQuasiDiag(link)
    sortIx=corr.index[sortIx].tolist() # recover labels
    hrp=getRecBipart(cov,sortIx)
    return hrp.sort_index()
    #-----------------------------------------
def getCLA(cov,**kargs):
    # Compute CLA's minimum variance portfolio
    mean=np.arange(cov.shape[0]).reshape(-1,1) # Not used by C portf
    lB=np.zeros(mean.shape)
    uB=np.ones(mean.shape)
    cla=CLA.CLA(mean,cov,lB,uB)
    cla.solve()
    return cla.w[-1].flatten()
#-------------------------------------------------
def hrpMC(numIters=1e4,nObs=520,size0=5,size1=5,mu0=0,sigma0=1e-2, \
    sigma1F=.25,sLength=260,rebal=22):
    # Monte Carlo experiment on HRP
    methods=[getIVP,getHRP,getCLA]
    stats,numIter={i.__name__:pd.Series() for i in methods},0
    pointers=range(sLength,nObs,rebal)
    while numIter<numIters:
        print numIter
        #1) Prepare data for one experiment
        x,cols=generateData(nObs,sLength,size0,size1,mu0,sigma0,sigma1F)
        r={i.__name__:pd.Series() for i in methods}
        #2) Compute portfolios in-sample
        for pointer in pointers:
            x_=x[pointer-sLength:pointer]
            cov_,corr_=np.cov(x_,rowvar=0),np.corrcoef(x_,rowvar=0)
            #3) Compute performance out-of-sample
            x_=x[pointer:pointer+rebal]
```

```
            for func in methods:
                w_=func(cov=cov_,corr=corr_) # callback
                r_=pd.Series(np.dot(x_,w_))
                r[func.__name__]=r[func.__name__].append(r_)
        #4) Evaluate and store results
        for func in methods:
            r_=r[func.__name__].reset_index(drop=True)
            p_=(1+r_).cumprod()
            stats[func.__name__].loc[numIter]=p_.iloc[-1]-1
        numIter+=1
    #5) Report results
    stats=pd.DataFrame.from_dict(stats,orient='columns')
    stats.to_csv('stats.csv')
    df0,df1=stats.std(),stats.var()
    print pd.concat([df0,df1,df1/df1['getHRP']-1],axis=1)
    return
#-------------------------------------------
if __name__=='__main__':hrpMC()
```

17.3 CUSUM TESTS

In Chapter 2 we introduced the CUSUM filter, which we applied in the context of event-based sampling of bars. The idea was to sample a bar whenever some variable, like cumulative prediction errors, exceeded a predefined threshold. This concept can be further extended to test for structural breaks.

17.3.1 Brown-Durbin-Evans CUSUM Test on Recursive Residuals

This test was proposed by Brown, Durbin and Evans [1975]. Let us assume that at every observation $t = 1, \ldots, T$, we count with an array of features x_t predictive of a value y_t. Matrix X_t is composed of the time series of features $t \leq T$, $\{x_i\}_{i=1,\ldots,t}$. These authors propose that we compute recursive least squares (RLS) estimates of β, based on the specification

$$y_t = \beta_t' x_t + \varepsilon_t$$

which is fit on subsamples $([1, k+1], [1, k+2], \ldots, [1, T])$, giving $T - k$ least squares estimates $(\hat{\beta}_{k+1}, \ldots, \hat{\beta}_T)$. We can compute the standardized 1-step ahead recursive residuals as

$$\hat{\omega}_t = \frac{y_t - \hat{\beta}_{t-1}' x_t}{\sqrt{f_t}}$$

$$f_t = \hat{\sigma}_\varepsilon^2 \left[1 + x_t' \left(X_t' X_t\right)^{-1} x_t\right]$$

The CUSUM statistic is defined as

$$S_t = \sum_{j=k+1}^{t} \frac{\hat{\omega}_j}{\hat{\sigma}_\omega}$$

$$\hat{\sigma}_\omega^2 = \frac{1}{T-k} \sum_{t=k}^{T} (\hat{\omega}_t - \mathrm{E}[\hat{\omega}_t])^2$$

Under the null hypothesis that β is some constant value, $H_0 : \beta_t = \beta$, then $S_t \sim N[0, t-k-1]$. One caveat of this procedure is that the starting point is chosen arbitrarily, and results may be inconsistent due to that.

17.3.2 Chu-Stinchcombe-White CUSUM Test on Levels

This test follows Homm and Breitung [2012]. It simplifies the previous method by dropping $\{x_t\}_{t=1,\dots,T}$, and assuming that $H_0 : \beta_t = 0$, that is, we forecast no change ($\mathrm{E}_{t-1}[\Delta y_t] = 0$). This will allow us to work directly with y_t levels, hence reducing the computational burden. We compute the standardized departure of log-price y_t relative to the log-price at y_n, $t > n$, as

$$S_{n,t} = (y_t - y_n)\left(\hat{\sigma}_t \sqrt{t-n}\right)^{-1}$$

$$\hat{\sigma}_t^2 = (t-1)^{-1} \sum_{i=2}^{t} (\Delta y_i)^2$$

Under the null hypothesis $H_0 : \beta_t = 0$, then $S_{n,t} \sim N[0,1]$. The time-dependent critical value for the *one-sided test* is

$$c_\alpha[n,t] = \sqrt{b_\alpha + \log[t-n]}$$

These authors derived via Monte Carlo that $b_{0.05} = 4.6$. One disadvantage of this method is that the reference level y_n is set somewhat arbitrarily. To overcome this pitfall, we could estimate $S_{n,t}$ on a series of backward-shifting windows $n \in [1,t]$, and pick $S_t = \sup_{n \in [1,t]} \{S_{n,t}\}$.

$$\Delta y_t = \delta y_{t-1} D_t[\tau^*] + \varepsilon_t$$

Equation 59

$$DFC_{\tau^*} = \frac{\hat{\delta}}{\hat{\sigma}_{\delta}}$$

Equation 60

$$SDFC = \sup_{\tau^* \in [\tau_0, 1-\tau_0]} \{DFC_{\tau^*}\}$$

Equation 61

$$\Delta y_t = \alpha + \beta y_{t-1} + \sum_{l=1}^{L} \gamma_l \Delta y_{t-l} + \varepsilon_t$$

Equation 62

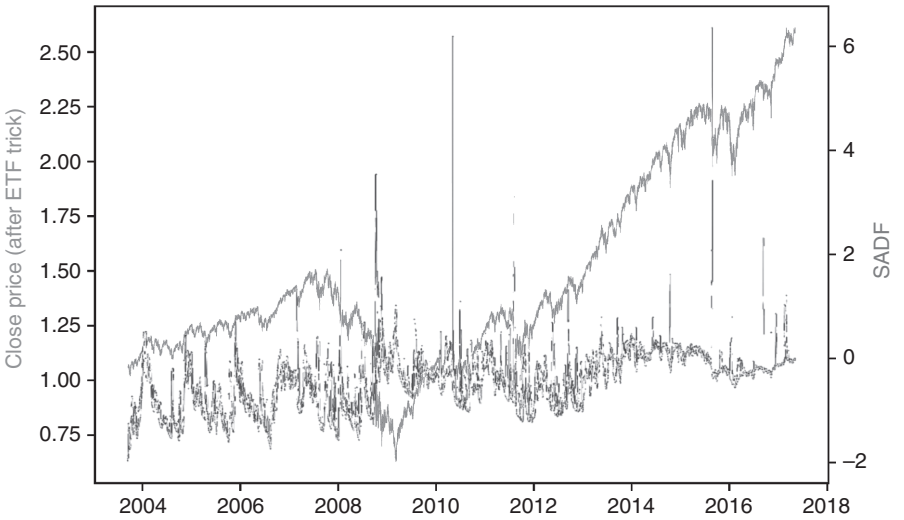


FIGURE 17.1 Prices (left y-axis) and SADF (right y-axis) over time

$$SADF_t = \sup_{t_0 \in [1, t-\tau]} \{ADF_{t_0,t}\} = \sup_{t_0 \in [1, t-\tau]} \left\{ \frac{\hat{\beta}_{t_0,t}}{\hat{\sigma}_{\beta_{t_0,t}}} \right\}$$

Equation 63

$$\sum_{t=\tau}^{T} t - \tau + 1 = \frac{1}{2}(T - \tau + 2)(T - \tau + 1) = \binom{T - \tau + 2}{2}$$

Equation 64

TABLE 17.1 FLOPs per ADF Estimate

Matrix Operation	FLOPs
$o_1 = X'y$	$(2T-1)N$
$o_2 = X'X$	$(2T-1)N^2$
$o_3 = o_2^{-1}$	$N^3 + N^2 + N$
$o_4 = o_3 o_1$	$2N^2 - N$
$o_5 = y - Xo_4$	$T + (2N-1)T$
$o_6 = o_5' o_5$	$2T - 1$
$o_7 = o_3 o_6 \frac{1}{T-N}$	$2 + N^2$
$o_8 = \frac{o_4[0,0]}{\sqrt{o_7[0,0]}}$	1

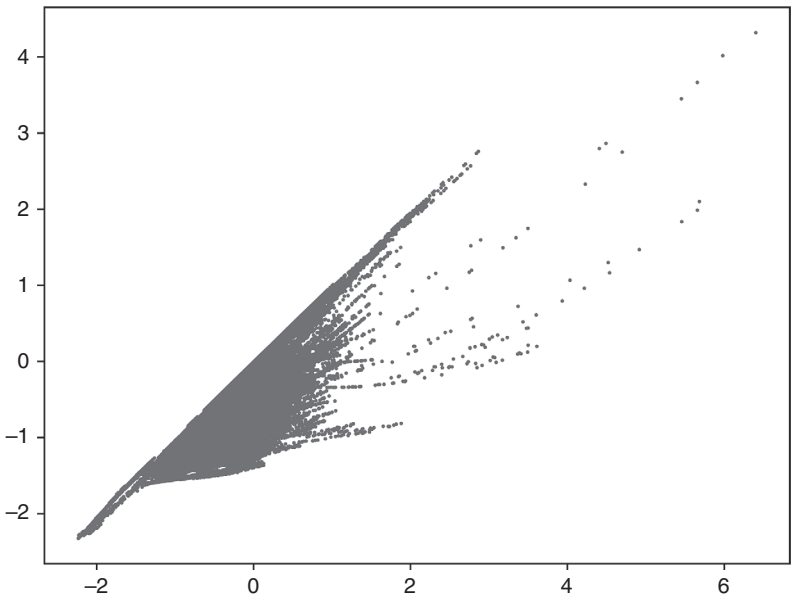
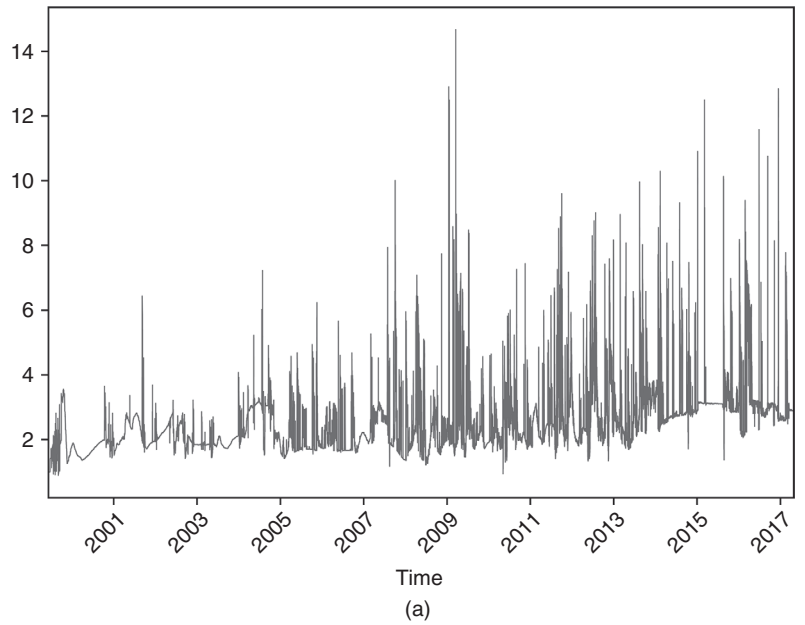


FIGURE 17.2 SADF (x-axis) vs CADF (y-axis)



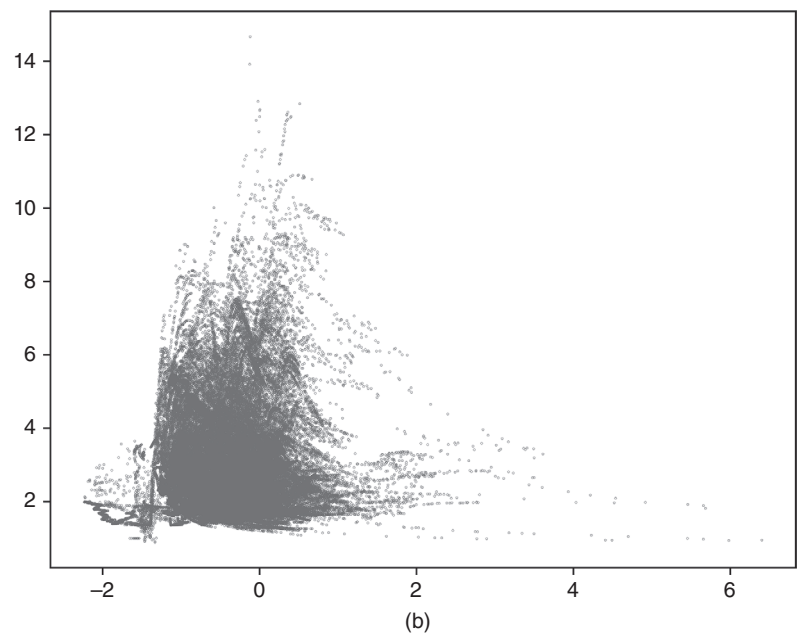


FIGURE 17.3 (a) $(SADF_t - C_{t,q})/\dot{C}_{t,q}$ over time (b) $(SADF_t - C_{t,q})/\dot{C}_{t,q}$ (y-axis) as a function of $SADF_t$ (x-axis)

SNIPPET 17.1 SADF'S INNER LOOP

```
def get_bsadf(logP,minSL,constant,lags):
    y,x=getYX(logP,constant=constant,lags=lags)
    startPoints,bsadf,allADF=range(0,y.shape[0]+lags-minSL+1),None,[]
    for start in startPoints:
        y_,x_=y[start:],x[start:]
        bMean_,bStd_=getBetas(y_,x_)
        bMean_,bStd_=bMean_[0,0],bStd_[0,0]**.5
        allADF.append(bMean_/bStd_)
        if allADF[-1]>bsadf:bsadf=allADF[-1]
    out={'Time':logP.index[-1],'gsadf':bsadf}
    return out
```

SNIPPET 17.2 PREPARING THE DATASETS

```
def getYX(series,constant,lags):
    series_=series.diff().dropna()
    x=lagDF(series_,lags).dropna()
    x.iloc[:,0]=series.values[-x.shape[0]-1:-1,0] # lagged level
    y=series_.iloc[-x.shape[0]:].values
        if constant!='nc':
            x=np.append(x,np.ones((x.shape[0],1)),axis=1)
            if constant[:2]=='ct':
                trend=np.arange(x.shape[0]).reshape(-1,1)
                x=np.append(x,trend,axis=1)
            if constant=='ctt':
                x=np.append(x,trend**2,axis=1)
        return y,x
```

SNIPPET 17.3 APPLY LAGS TO DATAFRAME

```
def lagDF(df0,lags):
    df1=pd.DataFrame()
    if isinstance(lags,int):lags=range(lags+1)
    else:lags=[int(lag) for lag in lags]
    for lag in lags:
        df_=df0.shift(lag).copy(deep=True)
        df_.columns=[str(i)+'_'+str(lag) for i in df_.columns]
        df1=df1.join(df_,how='outer')
    return df1
```

SNIPPET 17.4 FITTING THE ADF SPECIFICATION

```
def getBetas(y,x):
    xy=np.dot(x.T,y)
    xx=np.dot(x.T,x)
    xxinv=np.linalg.inv(xx)
    bMean=np.dot(xxinv,xy)
    err=y-np.dot(x,bMean)
    bVar=np.dot(err.T,err)/(x.shape[0]-x.shape[1])*xxinv
    return bMean,bVar
```

$$SMT_t = \sup_{t_0 \in [1, t-\tau]} \left\{ \frac{|\hat{\beta}_{t_0,t}|}{\hat{\sigma}_{\beta_{t_0,t}}} \right\}$$

Equation 65

$$SMT_t = \sup_{t_0 \in [1, t-\tau]} \left\{ \frac{|\hat{\beta}_{t_0,t}|}{\hat{\sigma}_{\beta_{t_0,t}} (t - t_0)^{\varphi}} \right\}$$

Equation 66

18.2 SHANNON'S ENTROPY

In this section we will review a few concepts from information theory that will be useful in the remainder of the chapter. The reader can find a complete exposition in MacKay [2003]. The father of information theory, Claude Shannon, defined entropy as the average amount of information (over long messages) produced by a stationary source of data. It is the smallest number of bits per character required to describe the message in a uniquely decodable way. Mathematically, Shannon [1948] defined the entropy of a discrete random variable X with possible values $x \in A$ as

$$H[X] \equiv -\sum_{x \in A} p[x] \log_2 p[x]$$

with $0 \leq H[X] \leq \log_2[\|A\|]$ where: $p[x]$ is the probability of x; $H[X] = 0 \Leftrightarrow \exists x | p[x] = 1$; $H[X] = \log_2[\|A\|] \Leftrightarrow p[x] = \frac{1}{\|A\|}$ for all x; and $\|A\|$ is the size of the set A. This can be interpreted as the probability weighted average of informational content in X, where the bits of information are measured as $\log_2 \frac{1}{p[x]}$. The rationale for measuring information as $\log_2 \frac{1}{p[x]}$ comes from the observation that low-probability outcomes reveal more information than high-probability outcomes. In other words, we learn when something unexpected happens. Similarly, redundancy is defined as

$$R[X] \equiv 1 - \frac{H[X]}{\log_2[\|A\|]}$$

with $0 \leq R[X] \leq 1$. Kolmogorov [1965] formalized the connection between redundancy and complexity of a Markov information source. The mutual information between two variables is defined as the Kullback-Leibler divergence from the joint probability density to the product of the marginal probability densities.

$$MI[X, Y] = \mathrm{E}_{f[x,y]} \left[\log \frac{f[x, y]}{f[x] f[y]} \right] = H[X] + H[Y] - H[X, Y]$$

The mutual information (MI) is always non-negative, symmetric, and equals zero if and only if X and Y are independent. For normally distributed variables, the mutual information is closely related to the familiar Pearson correlation, ρ.

$$MI[X, Y] = -\frac{1}{2} \log[1 - \rho^2]$$

Therefore, mutual information is a natural measure of the association between variables, regardless of whether they are linear or nonlinear in nature (Hausser and Strimmer [2009]). The normalized variation of information is a metric derived from mutual information. For several entropy estimators, see:

- In R: http://cran.r-project.org/web/packages/entropy/entropy.pdf
- In Python: https://code.google.com/archive/p/pyentropy/

$$\hat{H}_{n,w} = -\frac{1}{w} \sum_{y_1^w \in A^w} \hat{p}_w \left[y_1^w \right] \log_2 \hat{p}_w \left[y_1^w \right]$$

Equation 67

SNIPPET 18.1 PLUG-IN ENTROPY ESTIMATOR

```
import time,numpy as np
#———————————————————————————————
def plugIn(msg,w):
    # Compute plug-in (ML) entropy rate
    pmf=pmf1(msg,w)
    out=-sum([pmf[i]*np.log2(pmf[i]) for i in pmf])/w
    return out,pmf
#———————————————————————————————
def pmf1(msg,w):
    # Compute the prob mass function for a one-dim discrete rv
    # len(msg)-w occurrences
    lib={}
    if not isinstance(msg,str):msg=''.join(map(str,msg))
    for i in xrange(w,len(msg)):
        msg_=msg[i-w:i]
        if msg_ not in lib:lib[msg_]=[i-w]
        else:lib[msg_]=lib[msg_]+[i-w]
    pmf=float(len(msg)-w)
    pmf={i:len(lib[i])/pmf for i in lib}
    return pmf
```

SNIPPET 18.2 A LIBRARY BUILT USING THE LZ ALGORITHM

```
def lempelZiv_lib(msg):
    i,lib=1,[msg[0]]
    while i<len(msg):
        for j in xrange(i,len(msg)):
            msg_=msg[i:j+1]
            if msg_ not in lib:
                lib.append(msg_)
                break
        i=j+1
    return lib
```

$$L_i^n = 1 + \max\left\{l \middle| x_i^{i+l} = x_j^{j+l} \text{ for some } i - n \leq j \leq i - 1, l \in [0, n]\right\}$$

Equation 68

SNIPPET 18.3 FUNCTION THAT COMPUTES THE LENGTH OF THE LONGEST MATCH

```
def matchLength(msg,i,n):
    # Maximum matched length+1, with overlap.
    # i>=n & len(msg)>=i+n
    subS=''
    for l in xrange(n):
        msg1=msg[i:i+l+1]
        for j in xrange(i-n,i):
            msg0=msg[j:j+l+1]
            if msg1==msg0:
                subS=msg1
                break # search for higher l.
    return len(subS)+1,subS # matched length + 1
```

$$\lim_{n\to\infty} \frac{L_i^n}{\log_2[n]} = \frac{1}{H}$$

Equation 69

$$\hat{H}_{n,k} = \left[\frac{1}{k}\sum_{i=1}^{k} \frac{L_i^n}{\log_2[n]}\right]^{-1}$$

Equation 70

$$\hat{H}_n = \left[\frac{1}{n}\sum_{i=2}^{n} \frac{L_i^i}{\log_2[i]}\right]^{-1}$$

Equation 71

$$\tilde{H}_{n,k} = \frac{1}{k} \sum_{i=1}^{k} \frac{\log_2[n]}{L_i^n}$$

Equation 72

$$\tilde{H}_n = \frac{1}{n} \sum_{i=2}^{n} \frac{\log_2[i]}{L_i^i}$$

Equation 73

SNIPPET 18.4 IMPLEMENTATION OF ALGORITHMS DISCUSSED IN GAO ET AL. [2008]

```
def konto(msg,window=None):
    '''
    * Kontoyiannis' LZ entropy estimate, 2013 version (centered window).
    * Inverse of the avg length of the shortest non-redundant substring.
    * If non-redundant substrings are short, the text is highly entropic.
    * window==None for expanding window, in which case len(msg)%2==0
    * If the end of msg is more relevant, try konto(msg[::-1])
    '''
    out={'num':0,'sum':0,'subS':[]}
    if not isinstance(msg,str):msg=''.join(map(str,msg))
    if window is None:
        points=xrange(1,len(msg)/2+1)
    else:
        window=min(window,len(msg)/2)
        points=xrange(window,len(msg)-window+1)
    for i in points:
        if window is None:
            l,msg_=matchLength(msg,i,i)
            out['sum']+=np.log2(i+1)/l # to avoid Doeblin condition
        else:
            l,msg_=matchLength(msg,i,window)
            out['sum']+=np.log2(window+1)/l # to avoid Doeblin condition
        out['subS'].append(msg_)
        out['num']+=1
    out['h']=out['sum']/out['num']
    out['r']=1-out['h']/np.log2(len(msg)) # redundancy, 0<=r<=1
    return out
#-------------------------------------------
if __name__=='__main__':
    msg='101010'
    print konto(msg*2)
    print konto(msg+msg[::-1])
```

$$H = \frac{1}{2}\log[2\pi e\sigma^2]$$

Equation 74

$$M_q[x,p] = \left(\sum_{i=1}^{n} p_i x_i^q\right)^{1/q}$$

Equation 75

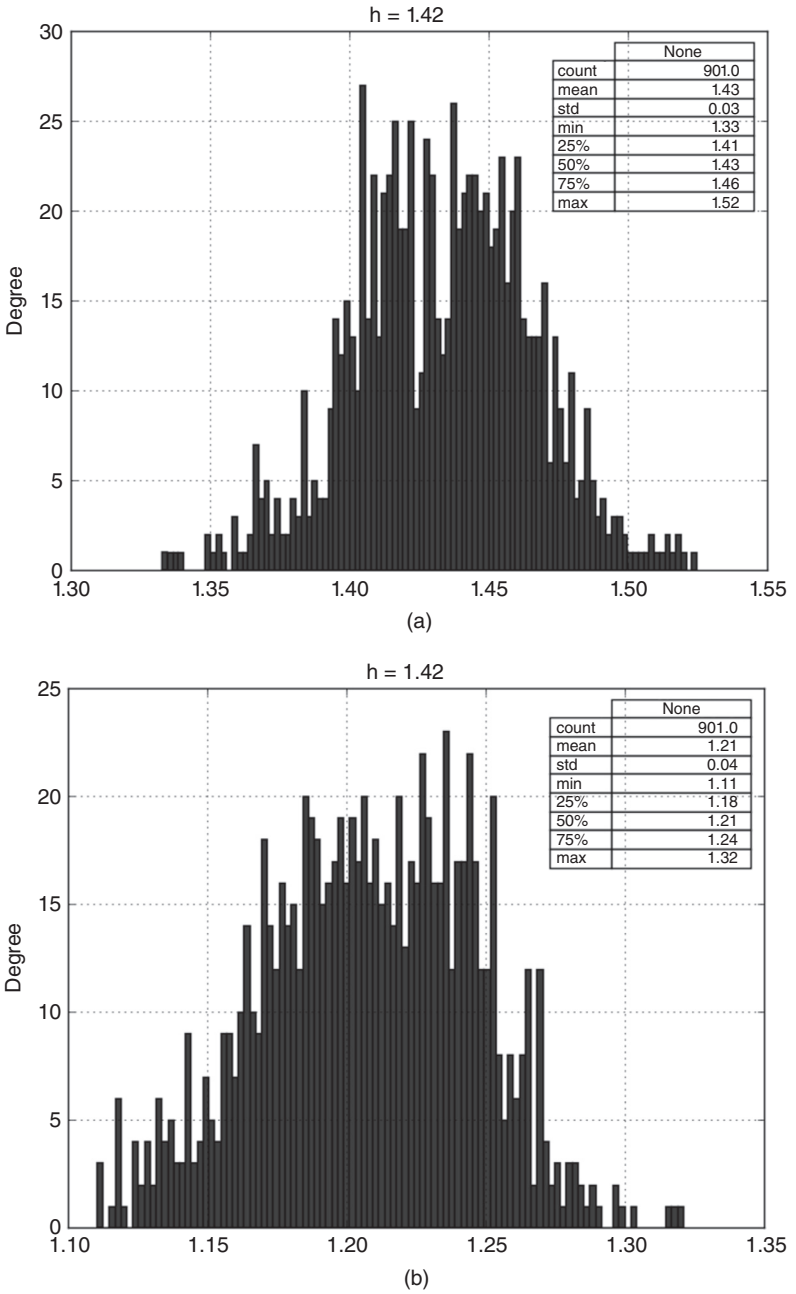
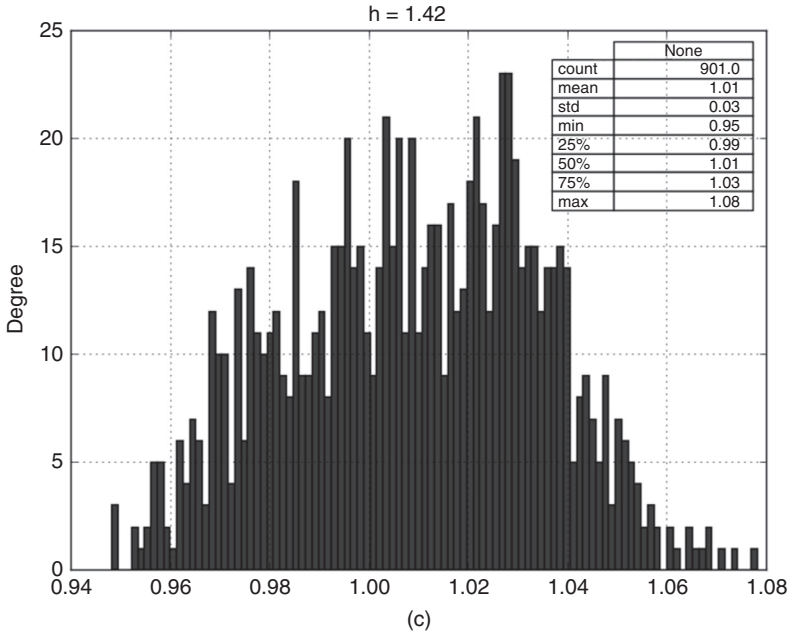


FIGURE 18.1 Distribution of entropy estimates under 10 (top), 7 (bottom), letter encodings, on messages of length 100



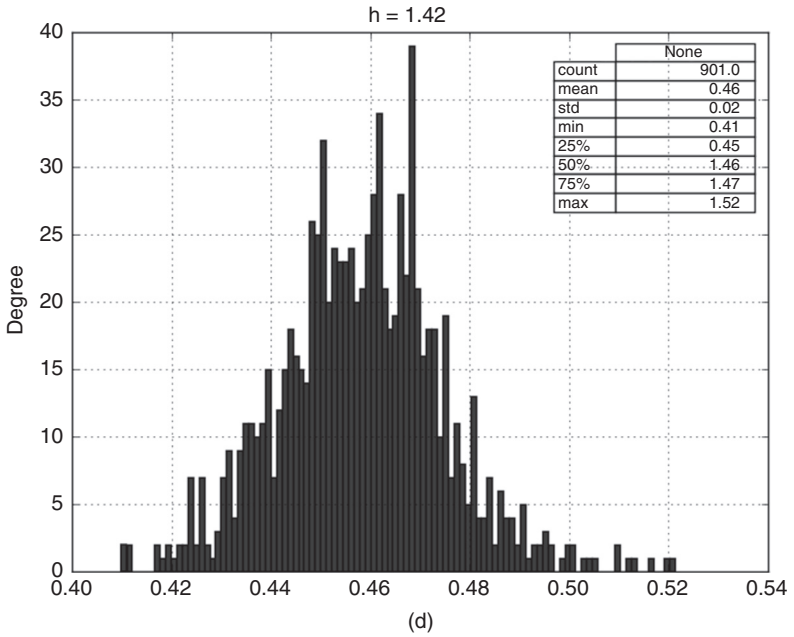


FIGURE 18.1 (*Continued*) Distribution of entropy estimates under 5 (top), and 2 (bottom) letter encodings, on messages of length 100

$$M_q[p,p] = \left(\sum_{i=1}^{n} p_i p_i^q \right)^{1/q}$$

Equation 76

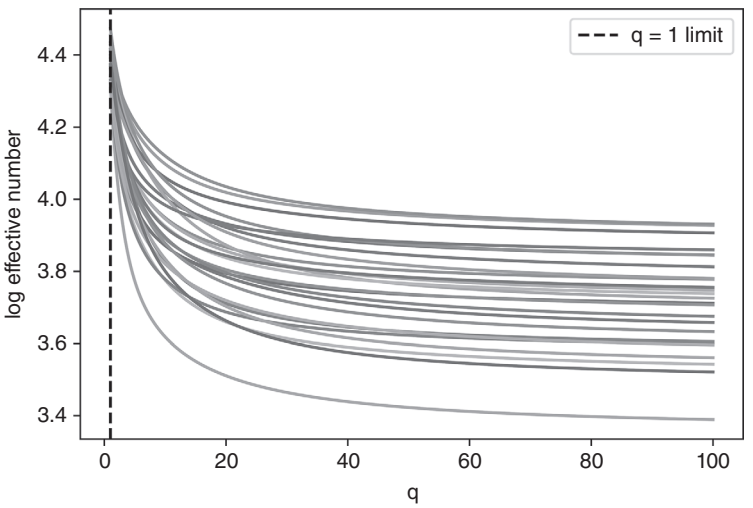


FIGURE 18.2 Log effective numbers for a family of randomly generated p arrays

$$\theta_i = \frac{[f_\omega]_i^2 \Lambda_{i,i}}{\sum_{n=1}^{N} [f_\omega]_n^2 \Lambda_{n,n}}$$

Equation 77

$$H = 1 - \frac{1}{N} e^{-\sum_{n=1}^{N} \theta_i \log[\theta_i]}$$

Equation 78

$$PIN = \frac{\alpha\mu}{\alpha\mu + 2\varepsilon}$$

Equation 79

$$VPIN = \frac{\alpha\mu}{\alpha\mu + 2\varepsilon} = \frac{\alpha\mu}{V} \approx \frac{1}{V} \mathrm{E}\left[\left|2V_\tau^B - V\right|\right] = \mathrm{E}\left[\left|2v_\tau^B - 1\right|\right]$$

Equation 80

$$b_t = \begin{cases} 1 & \text{if } \Delta p_t > 0 \\ -1 & \text{if } \Delta p_t < 0 \\ b_{t-1} & \text{if } \Delta p_t = 0 \end{cases}$$

Equation 81

$$\sigma^2 \left[\Delta p_t\right] = \mathrm{E}\left[\left(\Delta p_t\right)^2\right] - \left(\mathrm{E}\left[\left(\Delta p_t\right)\right]\right)^2 = 2c^2 + \sigma_u^2$$

Equation 82

$$\sigma \left[\Delta p_t, \Delta p_{t-1}\right] = -c^2$$

Equation 83

$$\mathrm{E}\left[\frac{1}{T}\sum_{t=1}^{T}\left(\log\left[\frac{H_t}{L_t}\right]\right)^2\right] = k_1\sigma_{HL}^2$$

Equation 84

$$\mathrm{E}\left[\frac{1}{T}\sum_{t=1}^{T}\left(\log\left[\frac{H_t}{L_t}\right]\right)\right] = k_2\sigma_{HL}$$

Equation 85

$$S_t = \frac{2\left(e^{\alpha_t}-1\right)}{1+e^{\alpha_t}}$$

Equation 86

$$\alpha_t = \frac{\sqrt{2\beta_t}-\sqrt{\beta_t}}{3-2\sqrt{2}} - \sqrt{\frac{\gamma_t}{3-2\sqrt{2}}}$$

Equation 87

$$\beta_t = \mathrm{E}\left[\sum_{j=0}^{1}\left[\log\left(\frac{H_{t-j}}{L_{t-j}}\right)\right]^2\right]$$

Equation 88

$$\gamma_t = \left[\log\left(\frac{H_{t-1,t}}{L_{t-1,t}}\right)\right]^2$$

Equation 89

SNIPPET 19.1 IMPLEMENTATION OF THE CORWIN-SCHULTZ ALGORITHM

```
def getBeta(series,sl):
    hl=series[['High','Low']].values
    hl=np.log(hl[:,0]/hl[:,1])**2
    hl=pd.Series(hl,index=series.index)
    beta=pd.stats.moments.rolling_sum(hl,window=2)
    beta=pd.stats.moments.rolling_mean(beta,window=sl)
    return beta.dropna()
#———————————————————————————————————-
def getGamma(series):
    h2=pd.stats.moments.rolling_max(series['High'],window=2)
    l2=pd.stats.moments.rolling_min(series['Low'],window=2)
    gamma=np.log(h2.values/l2.values)**2
    gamma=pd.Series(gamma,index=h2.index)
    return gamma.dropna()
#———————————————————————————————————-
def getAlpha(beta,gamma):
    den=3-2*2**.5
    alpha=(2**.5-1)*(beta**.5)/den
    alpha-=(gamma/den)**.5
    alpha[alpha<0]=0 # set negative alphas to 0 (see p.727 of paper)
    return alpha.dropna()
#———————————————————————————————————-
def corwinSchultz(series,sl=1):
    # Note: S<0 iif alpha<0
    beta=getBeta(series,sl)
    gamma=getGamma(series)
    alpha=getAlpha(beta,gamma)
    spread=2*(np.exp(alpha)-1)/(1+np.exp(alpha))
    startTime=pd.Series(series.index[0:spread.shape[0]],index=spread.index)
    spread=pd.concat([spread,startTime],axis=1)
    spread.columns=['Spread','Start_Time'] # 1st loc used to compute beta
    return spread
```

SNIPPET 19.2 ESTIMATING VOLATILITY FOR HIGH-LOW PRICES

```
def getSigma(beta,gamma):
    k2=(8/np.pi)**.5
    den=3-2*2**.5
    sigma=(2**-.5-1)*beta**.5/(k2*den)
    sigma+=(gamma/(k2**2*den))**.5
    sigma[sigma<0]=0
    return sigma
```

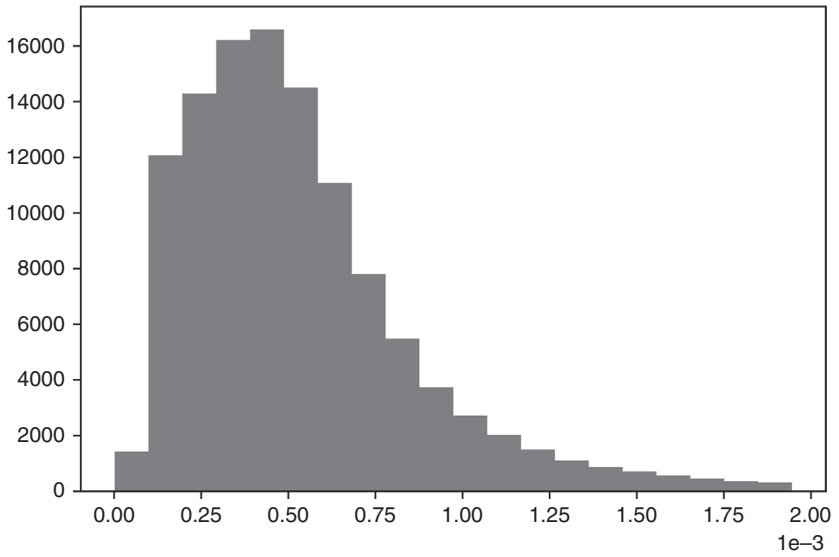


FIGURE 19.1 Kyle's Lambdas Computed on E-mini S&P 500 Futures

$$\Delta p_t = \lambda \left(b_t V_t \right) + \varepsilon_t$$

Equation 90

$$\left| \Delta \log \left[\tilde{p}_\tau \right] \right| = \lambda \sum_{t \in B_\tau} \left(p_t V_t \right) + \varepsilon_\tau$$

Equation 91

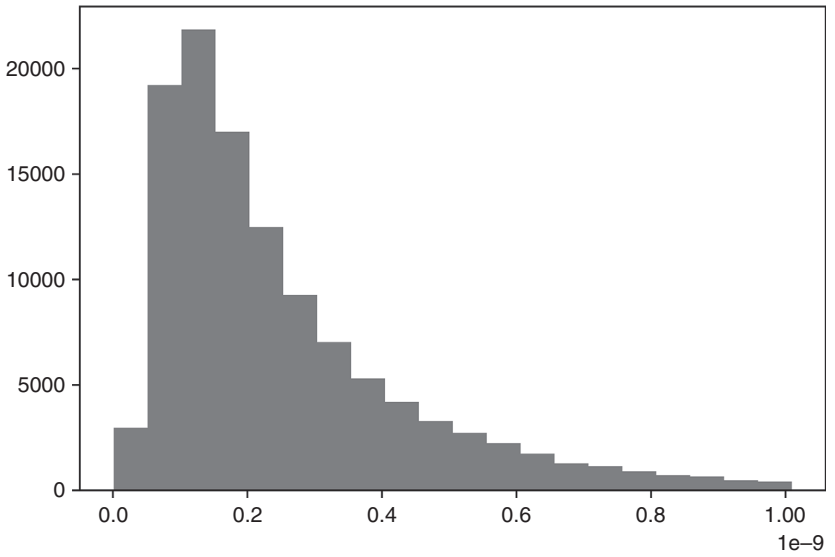


FIGURE 19.2 Amihud's lambdas estimated on E-mini S&P 500 futures

$$\log\left[\tilde{p}_{i,\tau}\right] - \log\left[\tilde{p}_{i,\tau-1}\right] = \lambda_i \sum_{t \in B_{i,\tau}} \left(b_{i,t}\sqrt{p_{i,t}V_{i,t}}\right) + \varepsilon_{i,\tau}$$

Equation 92

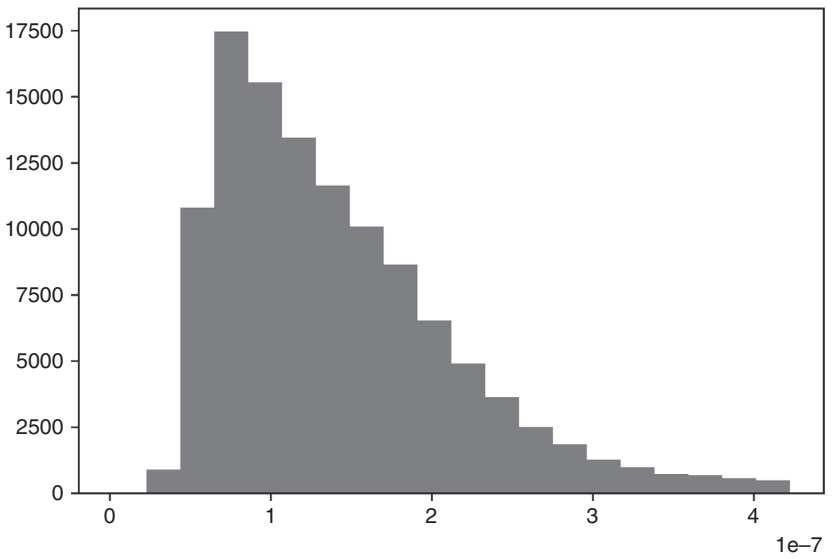


FIGURE 19.3 Hasbrouck's lambdas estimated on E-mini S&P 500 futures

19.5.1 Probability of Information-based Trading

Easley et al. [1996] use trade data to determine the probability of information-based trading (PIN) of individual securities. This microstructure model views trading as a game between market makers and position takers that is repeated over multiple trading periods.

Denote a security's price as S, with present value S_0. However, once a certain amount of new information has been incorporated into the price, S will be either S_B (bad news) or S_G (good news). There is a probability α that new information will arrive within the timeframe of the analysis, a probability δ that the news will be bad, and a probability $(1 - \delta)$ that the news will be good. These authors prove that the expected value of the security's price can then be computed at time t as

$$\mathrm{E}\left[S_t\right] = \left(1 - \alpha_t\right) S_0 + \alpha_t \left[\delta_t S_B + \left(1 - \delta_t\right) S_G\right]$$

Following a Poisson distribution, informed traders arrive at a rate μ, and uninformed traders arrive at a rate ε. Then, in order to avoid losses from informed traders, market makers reach breakeven at a bid level B_t,

$$\mathrm{E}\left[B_t\right] = \mathrm{E}\left[S_t\right] - \frac{\mu\alpha_t\delta_t}{\varepsilon + \mu\alpha_t\delta_t}\left(\mathrm{E}\left[S_t\right] - S_B\right)$$

and the breakeven ask level A_t at time t must be,

$$\mathrm{E}\left[A_t\right] = \mathrm{E}\left[S_t\right] + \frac{\mu\alpha_t\left(1-\delta_t\right)}{\varepsilon + \mu\alpha_t\left(1-\delta_t\right)}\left(S_G - \mathrm{E}\left[S_t\right]\right)$$

It follows that the breakeven bid-ask spread is determined as

$$\mathrm{E}\left[A_t - B_t\right] = \frac{\mu\alpha_t\left(1-\delta_t\right)}{\varepsilon + \mu\alpha_t\left(1-\delta_t\right)}\left(S_G - \mathrm{E}\left[S_t\right]\right) + \frac{\mu\alpha_t\delta_t}{\varepsilon + \mu\alpha_t\delta_t}\left(\mathrm{E}\left[S_t\right] - S_B\right)$$

For the standard case when $\delta_t = \frac{1}{2}$, we obtain

$$\delta_t = \frac{1}{2} \Rightarrow \mathrm{E}\left[A_t - B_t\right] = \frac{\alpha_t\mu}{\alpha_t\mu + 2\varepsilon}\left(S_G - S_B\right)$$

This equation tells us that the critical factor that determines the price range at which market makers provide liquidity is

$$PIN_t = \frac{\alpha_t\mu}{\alpha_t\mu + 2\varepsilon}$$

The subscript t indicates that the probabilities α and δ are estimated at that point in time. The authors apply a Bayesian updating process to incorporate information after each trade arrives to the market.

In order to determine the value PIN_t, we must estimate four non-observable parameters, namely $\{\alpha, \delta, \mu, \varepsilon\}$. A maximum-likelihood approach is to fit a mixture of three Poisson distributions,

$$\begin{aligned}\mathrm{P}[V^B, V^S] = {} & (1-\alpha)\mathrm{P}[V^B, \varepsilon]\mathrm{P}[V^S, \varepsilon] \\ & + \alpha(\delta\mathrm{P}[V^B, \varepsilon]\mathrm{P}[V^S, \mu+\varepsilon] + (1-\delta)\mathrm{P}[V^B, \mu+\varepsilon]\mathrm{P}[V^S, \varepsilon])\end{aligned}$$

where V^B is the volume traded against the ask (buy-initiated trades), and V^S is the volume traded against the bid (sell-initiated trades).

19.5.2 Volume-Synchronized Probability of Informed Trading

Easley et al. [2008] proved that

$$\begin{aligned}\mathrm{E}\left[V^B - V^S\right] &= (1-\alpha)(\varepsilon-\varepsilon) + \alpha(1-\delta)(\varepsilon-(\mu+\varepsilon)) + \alpha\delta(\mu+\varepsilon-\varepsilon) \\ &= \alpha\mu(1-2\delta)\end{aligned}$$

and in particular, for a sufficiently large μ,

$$\mathrm{E}[|V^B - V^S|] \approx \alpha\mu$$

Easley et al. [2011] proposed a high-frequency estimate of PIN, which they named volume-synchronized probability of informed trading (VPIN). This procedure adopts a *volume clock*, which synchronizes the data sampling with market activity, as captured by volume (see Chapter 2). We can then estimate

$$\frac{1}{n}\sum_{\tau=1}^{n}\left|V_\tau^B - V_\tau^S\right| \approx \alpha\mu$$

where V_τ^B is the sum of volumes from buy-initiated trades within volume bar τ, V_τ^S is the sum of volumes from sell-initiated trades within volume bar τ, and n is the number of bars used to produce this estimate. Because all volume bars are of the same size, V, we know that by construction

$$\frac{1}{n}\sum_{\tau=1}^{n}\left(V_\tau^B + V_\tau^S\right) = V = \alpha\mu + 2\varepsilon$$

Hence, PIN can be estimated in high-frequency as

$$VPIN_\tau = \frac{\sum_{\tau=1}^{n}\left|V_\tau^B - V_\tau^S\right|}{\sum_{\tau=1}^{n}\left(V_\tau^B + V_\tau^S\right)} = \frac{\sum_{\tau=1}^{n}\left|V_\tau^B - V_\tau^S\right|}{nV}$$

For additional details and case studies of VPIN, see Easley et al. [2013]. Using linear regressions, Andersen and Bondarenko [2013] concluded that VPIN is not a good predictor of volatility. However, a number of studies have found that VPIN indeed has predictive power: Abad and Yague [2012], Bethel et al. [2012], Cheung et al. [2015], Kim et al. [2014], Song et al. [2014], Van Ness et al. [2017], and Wei et al. [2013], to cite a few. In any case, linear regression is a technique that was already known to 18th-century mathematicians (Stigler [1981]), and economists should not be surprised when it fails to recognize complex non-linear patterns in 21st-century financial markets.

SNIPPET 20.1 UN-VECTORIZED CARTESIAN PRODUCT

```
# Cartesian product of dictionary of lists
dict0={'a':['1','2'],'b':['+','*'],'c':['!','@']}
for a in dict0['a']:
    for b in dict0['b']:
        for c in dict0['c']:
            print {'a':a,'b':b,'c':c}
```

SNIPPET 20.2 VECTORIZED CARTESIAN PRODUCT

```
# Cartesian product of dictionary of lists
from itertoolsimport izip,product
dict0={'a':['1','2'],'b':['+','*'],'c':['!','@']}
jobs=(dict(izip(dict0,i)) for i in product(*dict0.values()))
for i in jobs:print i
```

SNIPPET 20.3 SINGLE-THREAD IMPLEMENTATION OF A ONE-TOUCH DOUBLE BARRIER

```
import numpy as np
#---------------------------------------
def main0():
    # Path dependency: Sequential implementation
    r=np.random.normal(0,.01,size=(1000,10000))
    t=barrierTouch(r)
    return
#---------------------------------------
def barrierTouch(r,width=.5):
    # find the index of the earliest barrier touch
    t,p={},np.log((1+r).cumprod(axis=0))
    for j in xrange(r.shape[1]): # go through columns
        for i in xrange(r.shape[0]): # go through rows
            if p[i,j]>=width or p[i,j]<=-width:
                t[j]=i
                continue
    return t
#---------------------------------------
if __name__=='__main__':
    import timeit
    print min(timeit.Timer('main0()',setup='from __main__ import main0').repeat(5,10))
```

SNIPPET 20.4 MULTIPROCESSING IMPLEMENTATION OF A ONE-TOUCH DOUBLE BARRIER

```
import numpy as np
import multiprocessing as mp
#---------------------------------------
def main1():
    # Path dependency: Multi-threaded implementation
    r,numThreads=np.random.normal(0,.01,size=(1000,10000)),24
    parts=np.linspace(0,r.shape[0],min(numThreads,r.shape[0])+1)
    parts,jobs=np.ceil(parts).astype(int),[]
    for i in xrange(1,len(parts)):
        jobs.append(r[:,parts[i-1]:parts[i]]) # parallel jobs
    pool,out=mp.Pool(processes=numThreads),[]
    outputs=pool.imap_unordered(barrierTouch,jobs)
    for out_ in outputs:out.append(out_) # asynchronous response
    pool.close();pool.join()
    return
#---------------------------------------
if __name__=='__main__':
    import timeit
    print min(timeit.Timer('main1()',setup='from __main__ import main1').repeat(5,10))
```

SNIPPET 20.5 THE `linParts`FUNCTION

```
import numpy as np
#————————————————————————————
def linParts(numAtoms,numThreads):
    # partition of atoms with a single loop
    parts=np.linspace(0,numAtoms,min(numThreads,numAtoms)+1)
    parts=np.ceil(parts).astype(int)
    return parts
```

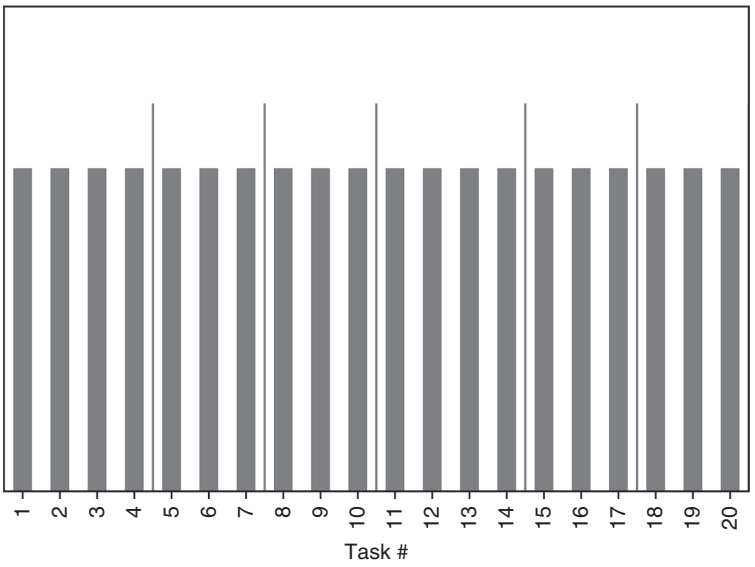


FIGURE 20.1 A linear partition of 20 atomic tasks into 6 molecules

$$r_1 = \frac{-1 + \sqrt{1 + 4N(N+1)M^{-1}}}{2}$$

Equation 93

$$r_2 = \frac{-1 + \sqrt{1 + 4\left(r_1^2 + r_1 + N(N+1)M^{-1}\right)}}{2}$$

Equation 94

$$r_m = \frac{-1 + \sqrt{1 + 4\left(r_{m-1}^2 + r_{m-1} + N(N+1)M^{-1}\right)}}{2}$$

Equation 95

SNIPPET 20.6 THE `nestedParts`FUNCTION

```
def nestedParts(numAtoms,numThreads,upperTriang=False):
    # partition of atoms with an inner loop
    parts,numThreads_=[0],min(numThreads,numAtoms)
    for num in xrange(numThreads_):
        part=1+4*(parts[-1]**2+parts[-1]+numAtoms*(numAtoms+1.)/numThreads_)
        part=(-1+part**.5)/2.
        parts.append(part)
    parts=np.round(parts).astype(int)
    if upperTriang: # the first rows are the heaviest
        parts=np.cumsum(np.diff(parts)[::-1])
        parts=np.append(np.array([0]),parts)
    return parts
```

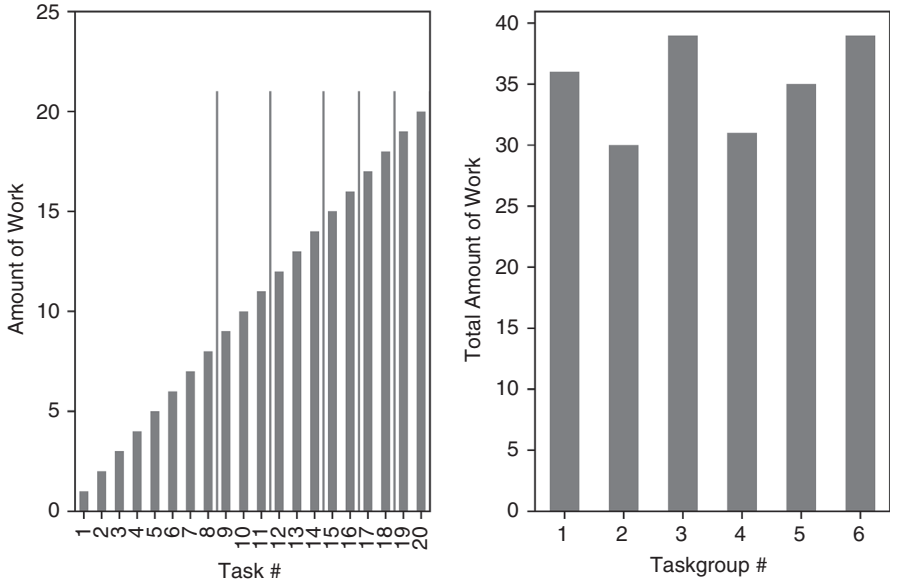


FIGURE 20.2 A two-nested loops partition of atoms into molecules

SNIPPET 20.7 THE `mpPandasObj`, USED AT VARIOUS POINTS IN THE BOOK

```
def mpPandasObj(func,pdObj,numThreads=24,mpBatches=1,linMols=True,**kargs):
    '''
    Parallelize jobs, return a DataFrame or Series
    + func: function to be parallelized. Returns a DataFrame
    + pdObj[0]: Name of argument used to pass the molecule
    + pdObj[1]: List of atoms that will be grouped into molecules
    + kargs: any other argument needed by func
    Example: df1=mpPandasObj(func,('molecule',df0.index),24,**kargs)
    '''
    import pandas as pd
    if linMols:parts=linParts(len(argList[1]),numThreads*mpBatches)
    else:parts=nestedParts(len(argList[1]),numThreads*mpBatches)
    jobs=[]    for i in xrange(1,len(parts)):
        job={pdObj[0]:pdObj[1][parts[i-1]:parts[i]],'func':func}
        job.update(kargs)
        jobs.append(job)
    if numThreads==1:out=processJobs_(jobs)
    else:out=processJobs(jobs,numThreads=numThreads)
    if isinstance(out[0],pd.DataFrame):df0=pd.DataFrame()
    elif isinstance(out[0],pd.Series):df0=pd.Series()
    else:return out
    for i in out:df0=df0.append(i)
    df0=df0.sort_index()
    return df0
```

SNIPPET 20.8 SINGLE-THREAD EXECUTION, FOR DEBUGGING

```
def processJobs_(jobs):
    # Run jobs sequentially, for debugging
    out=[]
    for job in jobs:
        out_=expandCall(job)
        out.append(out_)
    return out
```

SNIPPET 20.9 EXAMPLE OF ASYNCHRONOUS CALL TO PYTHON'S MULTIPROCESSING LIBRARY

```
import multiprocessing as mp
#————————————————————————————————
def reportProgress(jobNum,numJobs,time0,task):
    # Report progress as asynch jobs are completed
    msg=[float(jobNum)/numJobs,(time.time()-time0)/60.]
    msg.append(msg[1]*(1/msg[0]-1))
    timeStamp=str(dt.datetime.fromtimestamp(time.time()))
    msg=timeStamp+' '+str(round(msg[0]*100,2))+'% '+task+' done after '+ \
        str(round(msg[1],2))+' minutes. Remaining '+str(round(msg[2],2))+' minutes.'
    if jobNum<numJobs:sys.stderr.write(msg+'\r')
    else:sys.stderr.write(msg+'\n')
    return
#————————————————————————————————
def processJobs(jobs,task=None,numThreads=24):
    # Run in parallel.
    # jobs must contain a 'func' callback, for expandCall
    if task is None:task=jobs[0]['func'].__name__
    pool=mp.Pool(processes=numThreads)
    outputs,out,time0=pool.imap_unordered(expandCall,jobs),[],time.time()
    # Process asynchronous output, report progress
    for i,out_ in enumerate(outputs,1):
        out.append(out_)
        reportProgress(i,len(jobs),time0,task)
    pool.close();pool.join() # this is needed to prevent memory leaks
    return out
```

SNIPPET 20.10 PASSING THE JOB (MOLECULE) TO THE CALLBACK FUNCTION

```
def expandCall(kargs):
    # Expand the arguments of a callback function, kargs['func']
    func=kargs['func']
    del kargs['func']
    out=func(**kargs)
    return out
```

SNIPPET 20.11 PLACE THIS CODE AT THE BEGINNING OF YOUR ENGINE

```
def _pickle_method(method):
    func_name=method.im_func.__name__
    obj=method.im_self
    cls=method.im_class
    return _unpickle_method,(func_name,obj,cls)
#————————————————————————————
def _unpickle_method(func_name,obj,cls):
    for cls in cls.mro():
        try:func=cls.__dict__[func_name]
        except KeyError:pass
        else:break
    return func.__get__(obj,cls)
#————————————————————————————
import copy_reg,types,multiprocessing as mp
copy_reg.pickle(types.MethodType,_pickle_method,_unpickle_method)
```

SNIPPET 20.12 ENHANCING `processJobs` TO PERFORM ON-THE-FLY OUTPUT REDUCTION

```
def processJobsRedux(jobs,task=None,numThreads=24,redux=None,reduxArgs={},
                     reduxInPlace=False):
    '''
    Run in parallel
    jobs must contain a 'func' callback, for expandCall
    redux prevents wasting memory by reducing output on the fly
    '''
    if task is None:task=jobs[0]['func'].__name__
    pool=mp.Pool(processes=numThreads)
    imap,out,time0=pool.imap_unordered(expandCall,jobs),None,time.time()
    # Process asynchronous output, report progress
    for i,out_ in enumerate(imap,1):
        if out is None:
            if redux is None:out,redux,reduxInPlace=[out_],list.append,True
            else:out=copy.deepcopy(out_)
        else:
            if reduxInPlace:redux(out,out_,**reduxArgs)
            else:out=redux(out,out_,**reduxArgs)
        reportProgress(i,len(jobs),time0,task)
    pool.close();pool.join() # this is needed to prevent memory leaks
    if isinstance(out,(pd.Series,pd.DataFrame)):out=out.sort_index()
    return out
```

SNIPPET 20.13 ENHANCING `mpPandasObj` TO PERFORM ON-THE-FLY OUTPUT REDUCTION

```
def mpJobList(func,argList,numThreads=24,mpBatches=1,linMols=True,redux=None,
              reduxArgs={},reduxInPlace=False,**kargs):
    if linMols:parts=linParts(len(argList[1]),numThreads*mpBatches)
    else:parts=nestedParts(len(argList[1]),numThreads*mpBatches)
    jobs=[]
    for i in xrange(1,len(parts)):
        job={argList[0]:argList[1][parts[i-1]:parts[i]],'func':func}
        job.update(kargs)
        jobs.append(job)
    out=processJobsRedux(jobs,redux=redux,reduxArgs=reduxArgs,
                         reduxInPlace=reduxInPlace,numThreads=numThreads)
    return out
```

$$P = Z\tilde{W} = \sum_{b=1}^{B} Z_b \tilde{W}_b$$

Equation 96

SNIPPET 20.14 PRINCIPAL COMPONENTS FOR A SUBSET OF THE COLUMNS

```
pcs=mpJobList(getPCs,('molecules',fileNames),numThreads=24,mpBatches=1,
    path=path,eVec=eVec,redux=pd.DataFrame.add)
#———————————————————————————————————
def getPCs(path,molecules,eVec):
    # get principal components by loading one file at a time
    pcs=None
    for i in molecules:
        df0=pd.read_csv(path+i,index_col=0,parse_dates=True)
        if pcs is None:pcs=np.dot(df0.values,eVec.loc[df0.columns].values)
        else:pcs+=np.dot(df0.values,eVec.loc[df0.columns].values)
    pcs=pd.DataFrame(pcs,index=df0.index,columns=eVec.columns)
    return pcs
```

$$SR[r] = \frac{\sum_{h=1}^{H} \mu_h^{'} \omega_h - \tau_h[\omega]}{\sqrt{\sum_{h=1}^{H} \omega_h^{'} V_h \omega_h}}$$

Equation 97

$$\max_{\omega} \; SR[r]$$
$$\text{s.t.} : \sum_{i=1}^{N} |\omega_{i,h}| = 1, \; \forall h = 1, \ldots, H$$

Equation 98

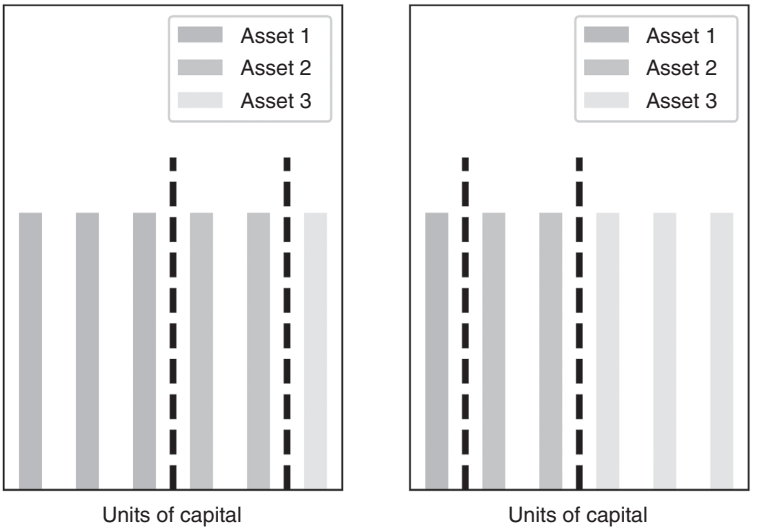


FIGURE 21.1 Partitions (1, 2, 3) and (3, 2, 1) must be treated as different

SNIPPET 21.1 PARTITIONS OF *k* OBJECTS INTO *n* SLOTS

```
from itertools import combinations_with_replacement
#———————————————————————————————
def pigeonHole(k,n):
    # Pigeonhole problem (organize k objects in n slots)
    for j in combinations_with_replacement(xrange(n),k):
        r=[0]*n
        for i in j:
            r[i]+=1
        yield r
```

SNIPPET 21.2 SET Ω OF ALL VECTORS ASSOCIATED WITH ALL PARTITIONS

```
import numpy as np
from itertools import product
#————————————————————————————————
def getAllWeights(k,n):
    #1) Generate partitions
    parts,w=pigeonHole(k,n),None
    #2) Go through partitions
    for part_ in parts:
        w_=np.array(part_)/float(k) # abs(weight) vector
        for prod_ in product([-1,1],repeat=n): # add sign
            w_signed_=(w_*prod_).reshape(-1,1)
            if w is None:w=w_signed_.copy()
            else:w=np.append(w,w_signed_,axis=1)
    return w
```

SNIPPET 21.3 EVALUATING ALL TRAJECTORIES

```
import numpy as np
from itertools import product
#———————————————————————————————
def evalTCosts(w,params):
    # Compute t-costs of a particular trajectory
    tcost=np.zeros(w.shape[1])
    w_=np.zeros(shape=w.shape[0])
    for i in range(tcost.shape[0]):
        c_=params[i]['c']
        tcost[i]=(c_*abs(w[:,i]-w_)**.5).sum()
        w_=w[:,i].copy()
    return tcost
#———————————————————————————————
def evalSR(params,w,tcost):
    # Evaluate SR over multiple horizons
    mean,cov=0,0
    for h in range(w.shape[1]):
        params_=params[h]
        mean+=np.dot(w[:,h].T,params_['mean'])[0]-tcost[h]
        cov+=np.dot(w[:,h].T,np.dot(params_['cov'],w[:,h]))
    sr=mean/cov**.5
    return sr
#———————————————————————————————
def dynOptPort(params,k=None):
    # Dynamic optimal portfolio
    #1) Generate partitions
    if k is None:k=params[0]['mean'].shape[0]
    n=params[0]['mean'].shape[0]
    w_all,sr=getAllWeights(k,n),None
    #2) Generate trajectories as cartesian products
    for prod_ in product(w_all.T,repeat=len(params)):
        w_=np.array(prod_).T # concatenate product into a trajectory
        tcost_=evalTCosts(w_,params)
        sr_=evalSR(params,w_,tcost_) # evaluate trajectory
        if sr is None or sr<sr_: # store trajectory if better
            sr,w=sr_,w_.copy()
    return w
```

SNIPPET 21.4 PRODUCE A RANDOM MATRIX OF A GIVEN RANK

```
import numpy as np
#————————————————————————————————
def rndMatWithRank(nSamples,nCols,rank,sigma=0,homNoise=True):
    # Produce random matrix X with given rank
    rng=np.random.RandomState()
    U,_,_=np.linalg.svd(rng.randn(nCols,nCols))
    x=np.dot(rng.randn(nSamples,rank),U[:,:rank].T)
    if homNoise:
        x+=sigma*rng.randn(nSamples,nCols) # Adding homoscedastic noise
    else:
        sigmas=sigma*(rng.rand(nCols)+.5) # Adding heteroscedastic noise
        x+=rng.randn(nSamples,nCols)*sigmas
    return x
```

SNIPPET 21.5 GENERATE THE PROBLEM'S PARAMETERS

```
import numpy as np
#———————————————————————————————
def genMean(size):
    # Generate a random vector of means
    rMean=np.random.normal(size=(size,1))
    return rMean
#———————————————————————————————
    #1) Parameters
    size,horizon=3,2
    params=[]
    for h in range(horizon):
        x=rndMatWithRank(1000,3,3,0.)
        mean_,cov_=genMean(size),np.cov(x,rowvar=False)
        c_=np.random.uniform(size=cov_.shape[0])*np.diag(cov_)**.5
        params.append({'mean':mean_,'cov':cov_,'c':c_})
```

SNIPPET 21.6 COMPUTE AND EVALUATE THE STATIC SOLUTION

```
import numpy as np
#———————————————————————————————
def statOptPortf(cov,a):
    # Static optimal porftolio
    # Solution to the "unconstrained" portfolio optimization problem
    cov_inv=np.linalg.inv(cov)
    w=np.dot(cov_inv,a)
    w/=np.dot(np.dot(a.T,cov_inv),a) # np.dot(w.T,a)==1
    w/=abs(w).sum() # re-scale for full investment
    return w
#———————————————————————————————
#2) Static optimal portfolios
w_stat=None
for params_ in params:
    w_=statOptPortf(cov=params_['cov'],a=params_['mean'])
    if w_stat is None:w_stat=w_.copy()
    else:w_stat=np.append(w_stat,w_,axis=1)
tcost_stat=evalTCosts(w_stat,params)
sr_stat=evalSR(params,w_stat,tcost_stat)
print 'static SR:',sr_stat
```

SNIPPET 21.7 COMPUTE AND EVALUATE THE DYNAMIC SOLUTION

```
import numpy as np
#————————————————————————————————
#3) Dynamic optimal portfolios
w_dyn=dynOptPort(params)
tcost_dyn=evalTCosts(w_dyn,params)
sr_dyn=evalSR(params,w_dyn,tcost_dyn)
print 'dynamic SR:',sr_dyn
```

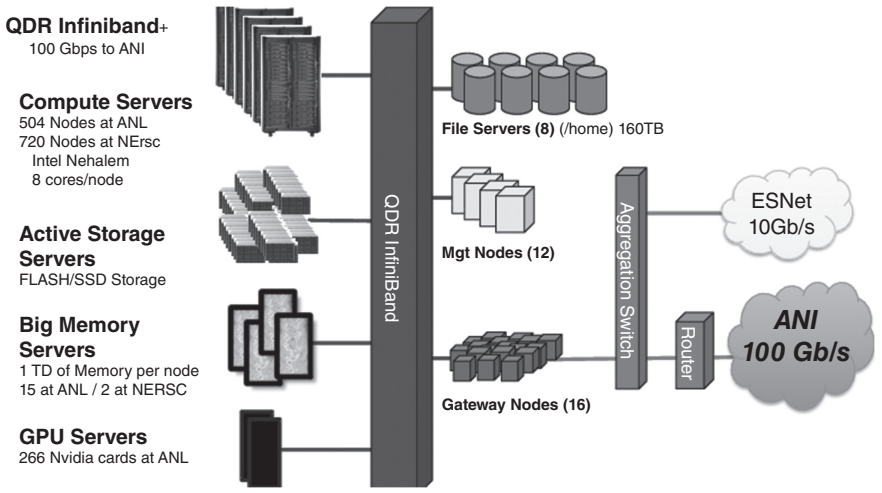


FIGURE 22.1 Schematic of the Magellan cluster (circa 2010), an example of HPC computer cluster

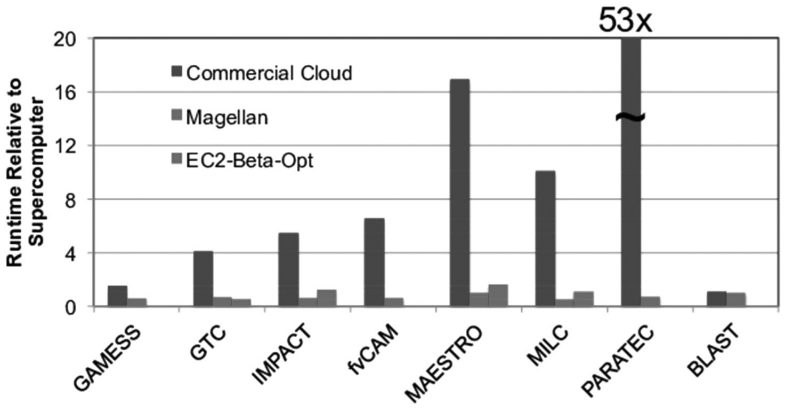


FIGURE 22.2 The cloud ran scientific applications considerably slower than on HPC systems (circa 2010)

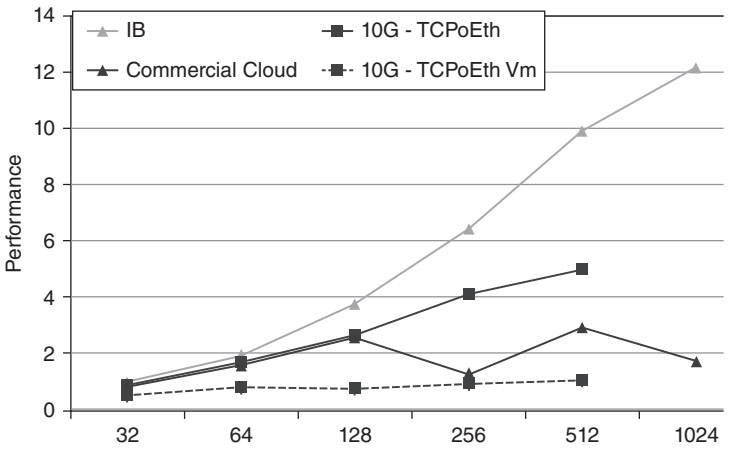


FIGURE 22.3 As the number of cores increases (horizontal axis), the virtualization overhead becomes much more significant (circa 2010)

FIGURE 22.4 Supernova SN 2011fe was discovered 11 hours after first evidence of explosion, as a result of the extensive automation in classification of astronomical observations

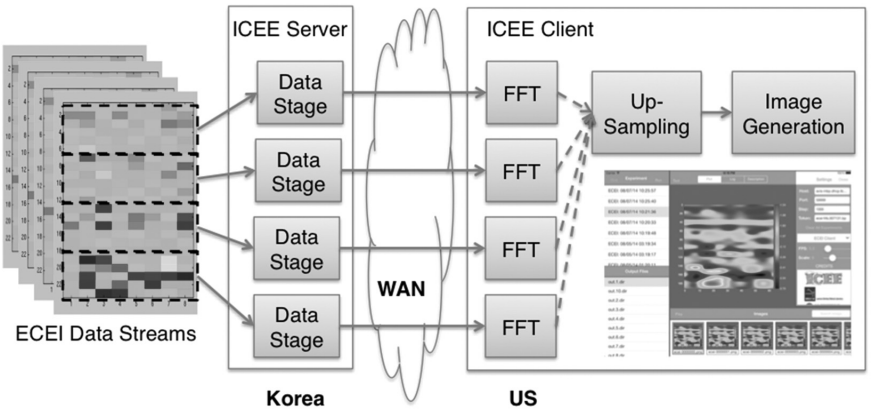
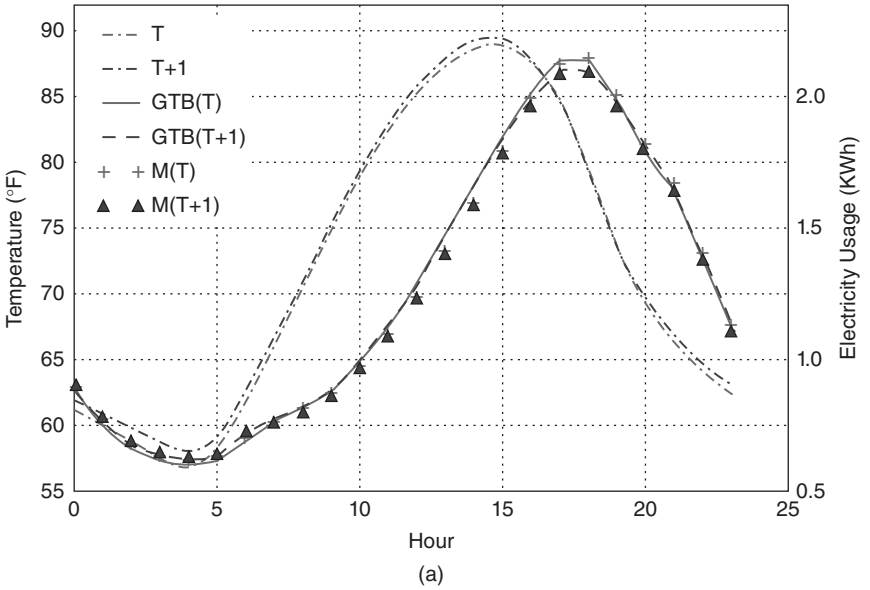


FIGURE 22.5 A distributed workflow for studying fusion plasma dynamics



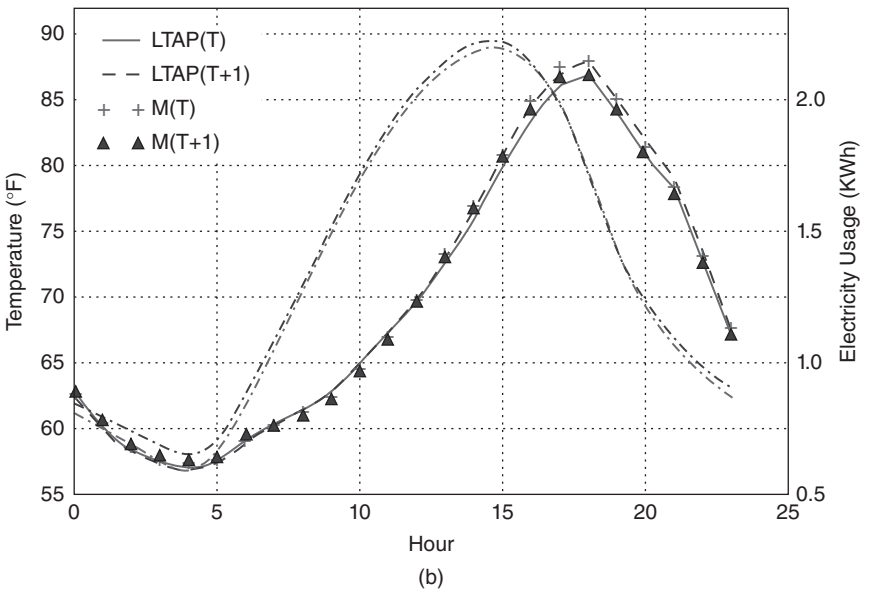
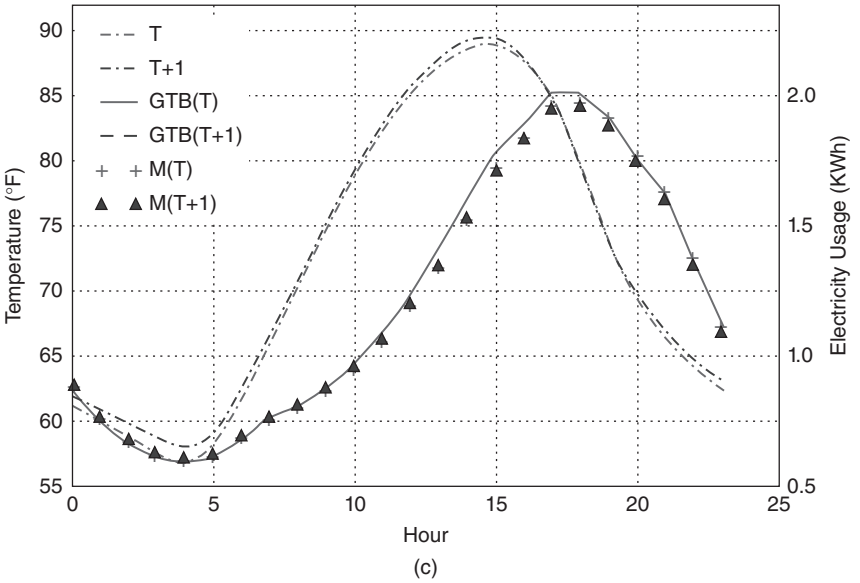


FIGURE 22.6 Gradient tree boosting (GBT) appears to follow recent usage too closely and therefore not able to predict the baseline usage as well as the newly develop method named LTAP. (a) GTB on Control group. (b) LTAP on Control group. (c) GTB on Passive group. (d) LTAP on Passive group. (e) GTB on Active group. (f) LTAP on Active group



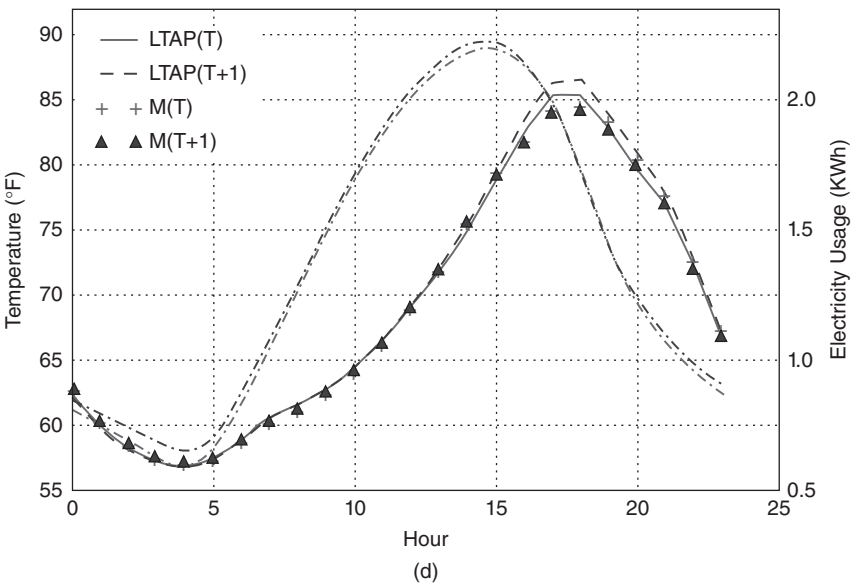
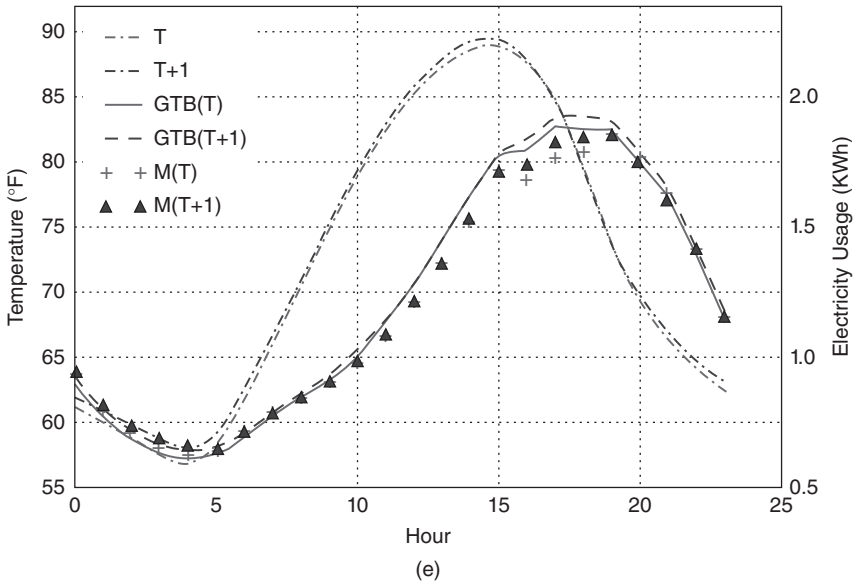


FIGURE 22.6 (*Continued*)



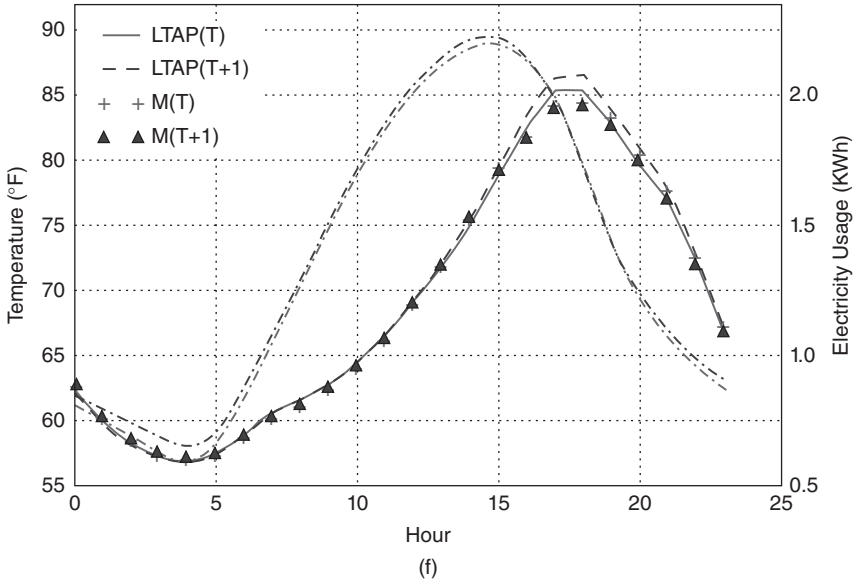


FIGURE 22.6 (*Continued*)

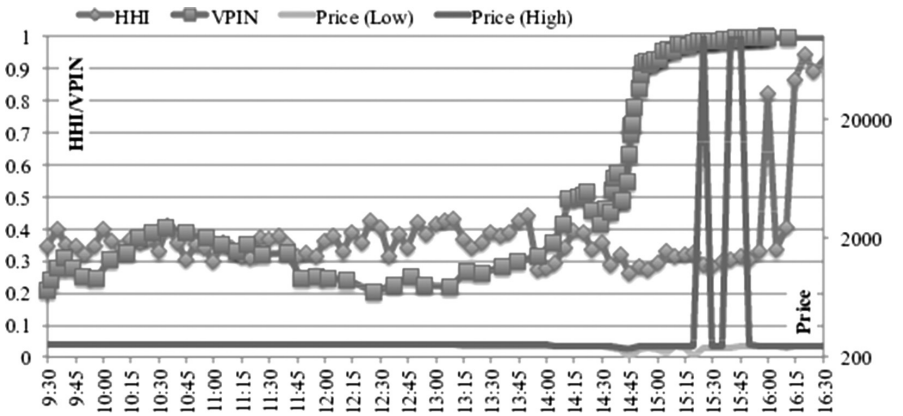


FIGURE 22.7 Apple Stock price on May 6, 2010, along with HHI and VPIN values computed every 5 minutes during the market hours

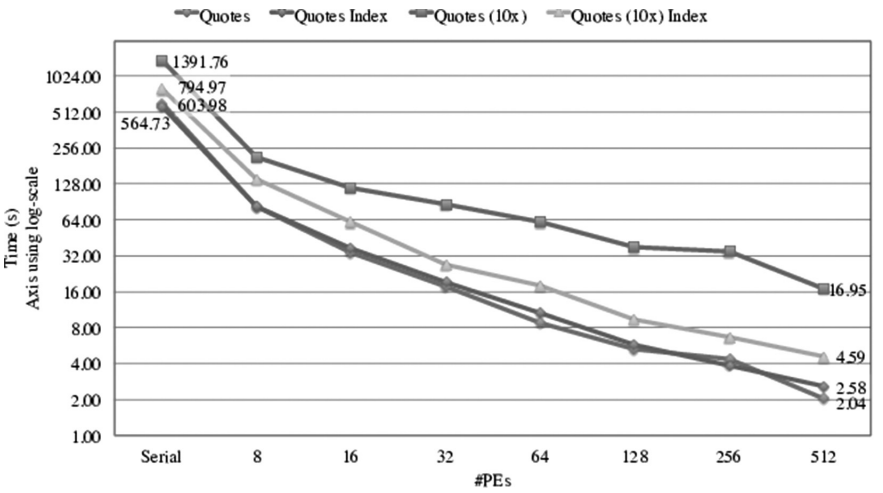


FIGURE 22.8 Time to process 10-year worth of SP500 quotes data stored in HDF5 files, which takes 21 times longer when the same data is in ASCII files (603.98 seconds versus approximately 3.5 hours)

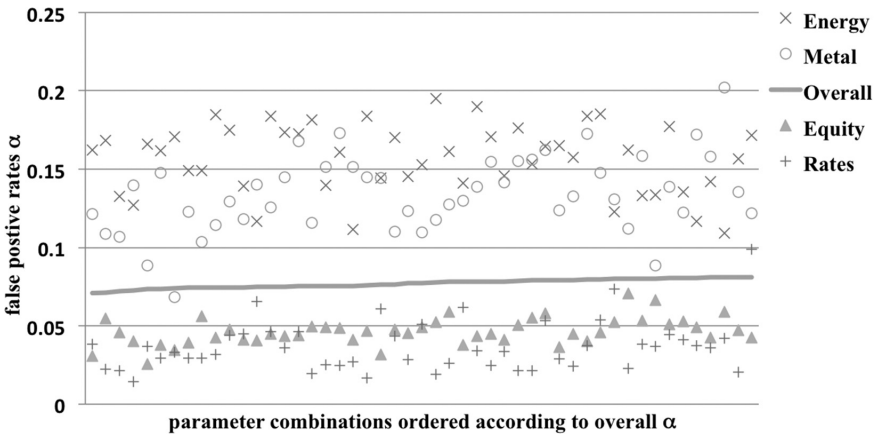


FIGURE 22.9 The average false positive rates (α) of different classes of futures contracts ordered according to their average.

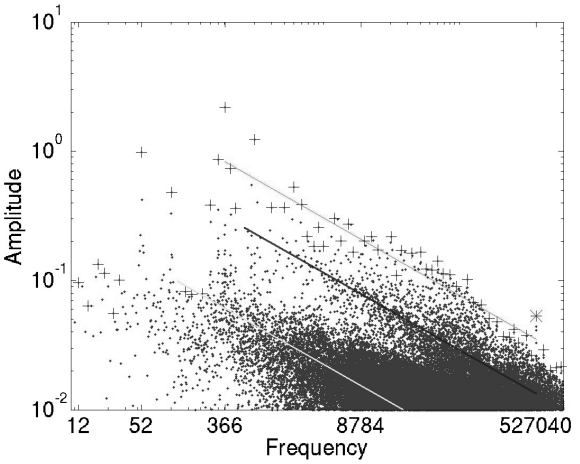


FIGURE 22.10 Fourier spectrum of trading prices of natural gas futures contracts in 2012. Non-uniform FFT identifies strong presence of activities happening once per day (frequency = 366), twice per day (frequency = 732), and once per minute (frequency = 527040 = 366*24*60).