# 6 Feature Importance Analysis

## 6.1 Motivation

Imagine that you are given ten puzzles, each of a thousand pieces, where all of the pieces have been shuffled into the same box. You are asked to solve one particular puzzle out of the ten. A reasonable way to proceed is to divide your task into two steps. In the first step, you try to isolate the one thousand pieces that are important to your problem, and discard the nine thousands pieces that are irrelevant. For example, you may notice that about one tenth of the pieces are made of plastic, and the rest are made of paper. Regardless of the pattern shown on the pieces, you know that discarding all paper pieces will isolate a single puzzle. In the second step, you try to fit a structure on the one thousand pieces that you have isolated. Now you may make a guess of what the pattern is, and organize the pieces around it.

Now consider a researcher interested in modeling a dynamic system as a function of many different candidate explanatory variables. Only a small subset of those candidate variables are expected to be relevant, however the researcher does not know in advance which. The approach generally followed in the financial literature is to try to fit a guessed algebraic function on a guessed subset of variables, and see which variables appear to be statistically significant (subject to that guessed algebraic function being correct, including all interaction effects among variables). Such an approach is counterintuitive and likely to miss important variables that would have been revealed by unexplored specifications. Instead, researchers could follow the same steps that they would apply to the problem of solving a puzzle: first, isolate the important variables, irrespective of any functional form, and only then try to fit those variables to a particular specification that is consistent with those isolated variables. ML techniques allow us to disentangle the specification search from the variable search.

In this section, we demonstrate that ML provides intuitive and effective tools for researchers who work on the development of theories. Our exposition runs counter to the popular myth that supervised ML models are black-boxes. According to that view, supervised ML algorithms find predictive patterns, however researchers have no understanding of those findings. In other words, the algorithm has learned something, not the researcher. This criticism is unwarranted.

Even if a supervised ML algorithm does not yield a closed-form algebraic solution (like, for example, a regression method would do), an analysis of its forecasts can tell us what variables are critically involved in a particular phenomenon, what variables are redundant, what variables are useless, and how the relevant variables interact with each other. This kind of analysis is known as "feature importance," and harnessing its power will require us to use everything we have learned in the previous sections.

## 6.2 *p*-Values

The classical regression framework makes a number of assumptions regarding the fitted model, such as correct model specification, mutually uncorrelated regressors, or white noise residuals. Conditional on those assumptions being true, researchers aim to determine the importance of an explanatory variable through a hypothesis test.[15] A popular way of expressing a variable's significance is through its *p*-value, a concept that dates back to the 1700s (Brian and Jaisson 2007). The *p*-value quantifies the probability that, if the true coefficient associated with that variable is zero, we could have obtained a result equal or more extreme than the one we have estimated. It indicates how incompatible the data are with a specified statistical model. However, a *p*-value does not measure the probability that neither the null nor the alternative hypothesis is true, or that the data are random. And a *p*-value does not measure the size of an effect, or the significance of a result.[16] The misuse of *p*-values is so widespread that the American Statistical Association has discouraged their application going forward as a measure of statistical significance (Wasserstein et al. 2019). This casts a doubt over decades of empirical research in Finance. In order to search for alternatives to the *p*-value, first we must understand its pitfalls.

### 6.2.1 A Few Caveats of p-Values

A first caveat of *p*-values is that they rely on the strong assumptions outlined earlier. When those assumptions are inaccurate, a *p*-value could be low even though the true value of the coefficient is zero (a false positive), and the *p*-value could be high even though the true value of the coefficient is not zero (a false negative).

A second caveat of *p*-values is that, for highly multicollinear (mutually correlated) explanatory variables, *p*-values cannot be robustly estimated. In multicollinear systems, traditional regression methods cannot discriminate among redundant explanatory variables, leading to substitution effects between related *p*-values.

A third caveat of *p*-values is that they evaluate a probability that is not entirely relevant. Given a null hypothesis $H_0$ and an estimated coefficient $\hat{\beta}$, the *p*-value estimates the probability of obtaining a result equal or more extreme than $\hat{\beta}$, subject to $H_0$ being true. However, researchers are often more interested in a different probability, namely, the probability of $H_0$ being true, subject to having observed $\hat{\beta}$. This probability can be computed using Bayes theorem, alas at the expense of making additional assumptions (Bayesian priors).[17]

[15] Some significance tests also demand that the residuals follow a Gaussian distribution.

[16] For additional details, read the "Statement on Statistical Significance and P-Values" by the American Statistical Association (2016) and Wasserstein and Lazar (2016).

[17] We revisit this argument in Section 8.2.

A fourth caveat of *p*-values is that it assesses significance in-sample. The entire sample is used to solve two tasks: estimating the coefficients and determining their significance. Accordingly, *p*-values may be low (i.e., significant) for variables that have no out-of-sample explanatory (i.e., forecasting) value. Running multiple in-sample tests on the same data set is likely to produce a false discovery, a practice known as *p*-hacking.

In summary, *p*-values require that we make many assumptions (caveat #1) in order to produce a noisy estimate (caveat #2) of a probability that we do not really need (caveat #3), and that may not be generalizable out-of-sample (caveat #4). These are not superfluous concerns. In theory, a key advantage of classical methods is that they provide a transparent attribution of significance among explanatory variables. But since that classical attribution has so many caveats in practice, perhaps classical methods could use some help from modern computational techniques that overcome those caveats.

### 6.2.2 A Numerical Example

Consider a binary random classification problem composed of forty features, where five are informative, thirty are redundant, and five are noise. Code Snippet 6.1 implements function getTestData, which generates informative, redundant, and noisy features. Informative features (marked with the "I_" prefix) are those used to generate labels. Redundant features (marked with the "R_" prefix) are those that are formed by adding Gaussian noise to a randomly chosen informative feature (the lower the value of sigmaStd, the greater the substitution effect). Noise features (marked with the "N_" prefix) are those that are not used to generate labels.

Figure 6.1 plots the *p*-values that result from a logit regression on those features. The horizontal bars report the *p*-values, and the vertical dashed line marks the 5% significance level. Only four out of the thirty-five nonnoise features are deemed statistically significant: I_1, R_29, R_27, I_3. Noise features are ranked as relatively important (with positions 9, 11, 14, 18, and 26). Fourteen of the features ranked as least important are not noise. In short, these *p*-values misrepresent the ground truth, for the reasons explained earlier.

Unfortunately, financial data sets tend to be highly multicollinear, as a result of common risk factors shared by large portions of the investment universe: market, sector, rating, value, momentum, quality, duration, etc. Under these circumstances, financial researchers should cease to rely exclusively on *p*-values. It is important for financial researchers to become familiar with additional methods to determine what variables contain information in a particular phenomenon.

SNIPPET 6.1 GENERATING A SET OF INFORMED, REDUNDANT, AND NOISE EXPLANATORY VARIABLES

```
def getTestData(n_features=100,n_informative=25,n_redundant=25,
  n_samples=10000,random_state=0,sigmaStd=.0):
  # generate a random dataset for a classification problem
  from sklearn.datasets import make_classification
  np.random.seed(random_state)
  X,y=make_classification(n_samples=n_samples,
    n_features=n_features-n_redundant,
    n_informative=n_informative,n_redundant=0,shuffle=False,
    random_state=random_state)
  cols=['I_'+str(i) for i in xrange(n_informative)]
  cols+=['N_'+str(i) for i in xrange(n_features-n_informative- \
    n_redundant)]
  X,y=pd.DataFrame(X,columns=cols),pd.Series(y)
  i=np.random.choice(xrange(n_informative),size=n_redundant)
  for k,j in enumerate(i):
    X['R_'+str(k)]=X['I_'+str(j)]+np.random.normal(size= \
      X.shape[0])*sigmaStd
  return X,y
#------------------------------------------------------
import numpy as np,pandas as pd,seaborn as sns
import statsmodels.discrete.discrete_model as sm
X,y=getTestData(40,5,30,10000,sigmaStd=.1)
ols=sm.Logit(y,X).fit()
```

## 6.3 Feature Importance

In this section, we study how two of ML's feature importance methods address the caveats of *p*-values, with minimal assumptions, using computational techniques. Other examples of ML interpretability methods are accumulated local effects (Apley 2016) and Shapley values (Štrumbelj 2014).

### 6.3.1 Mean-Decrease Impurity

Suppose that you have a learning sample of size $N$, composed of $F$ features, $\{X_f\}_{f=1,\ldots,F}$ and one label per observation. A tree-based classification (or regression) algorithm splits at each node $t$ its labels into two samples: for a given feature $X_f$, labels in node $t$ associated with a $X_f$ below a threshold $\tau$ are placed in the left sample, and the rest are placed in the right sample. For each of these samples, we can evaluate their impurity as the entropy of the distribution of labels, as the Gini index, or following some other criterion. Intuitively, a sample is purest when it contains only labels of one kind, and it is most impure

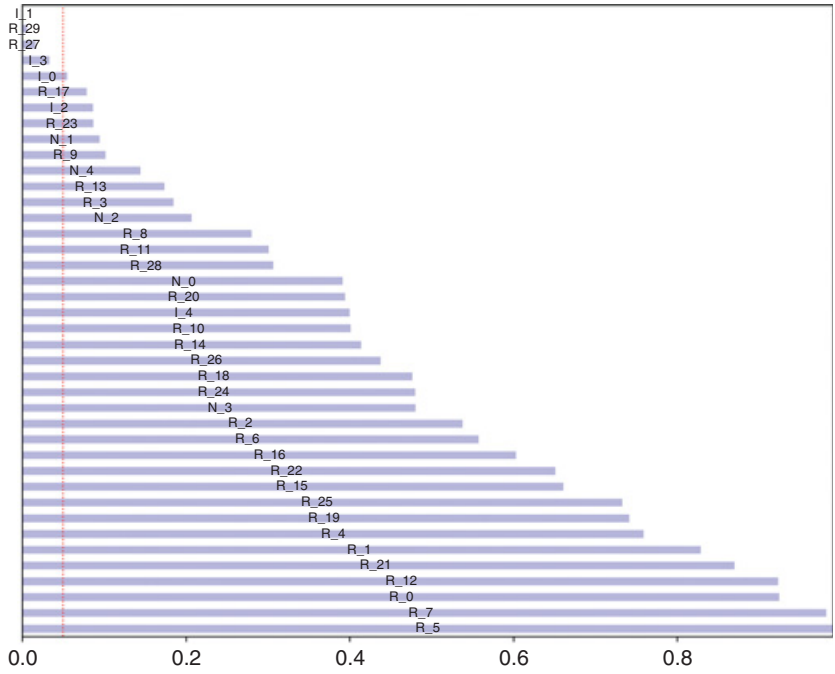


**Figure 6.1** $p$-Values computed on a set of explanatory variables.

when its labels follow a uniform distribution. The information gain that results from a split is measured in terms of the resulting reduction in impurity,

$$\Delta g[t,f] = i[t] - \frac{N_t^{(0)}}{N_t} i[t^{(0)}] - \frac{N_t^{(1)}}{N_t} i[t^{(1)}],$$

where $i[t]$ is the impurity of labels at node $t$ (before the split), $i[t^{(0)}]$ is the impurity of labels in the left sample, and $i[t^{(1)}]$ is the impurity of labels in the right sample. At each node $t$, the classification algorithm evaluates $\Delta g[t,f]$ for various features in $\{X_f\}_{f=1,\ldots,F}$, determines the optimal threshold $\tau$ that maximizes $\Delta g[t,f]$ for each of them, and selects the feature $f$ associated with greatest $\Delta g[t,f]$. The classification algorithm continues splitting the samples further until no additional information gains can be produced, or some early-stopping condition is met, such as achieving an impurity below the maximum acceptable limit.

The importance of a feature can be computed as the weighted information gain ($\Delta g[t,f]$) across all nodes where that feature was selected. This tree-based feature importance concept, introduced by Breiman (2001), is known as mean-decrease impurity (MDI). By construction, the MDI value associated with each feature is bounded between 0 and 1, and all combined add up to 1. In the

presence of $F$ features where all are uninformative (or equally informed), each MDI value is expected to be $1/F$. For algorithms that combine ensembles of trees, like random forests, we can further estimate the mean and variance of MDI values for each feature across all trees. These mean and variance estimates, along with the central limit theorem, are useful in testing the significance of a feature against a user-defined null hypothesis. Code Snippet 6.2 implements an ensemble MDI procedure. See López de Prado (2018a) for practical advice on how to use MDI.

SNIPPET 6.2 IMPLEMENTATION OF AN ENSEMBLE MDI METHOD

```
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) # CLT
   imp/=imp['mean'].sum()
   return imp
#-----------------------------------------------------------
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
X,y=getTestData(40,5,30,10000,sigmaStd=.1)
clf=DecisionTreeClassifier(criterion='entropy',max_features=1,
   class_weight='balanced',min_weight_fraction_leaf=0)
clf=BaggingClassifier(base_estimator=clf,n_estimators=1000,
   max_features=1.,max_samples=1.,oob_score=False)
fit=clf.fit(X,y)
imp=featImpMDI(fit,featNames=X.columns)
```

Figure 6.2 plots the result of applying MDI to the same random classification problem discussed in Figure 6.1. The horizontal bars indicate the mean of MDI values across 1,000 trees in a random forest, and the lines indicate the standard deviation around that mean. The more trees we add to the forest, the smaller becomes the standard deviation around the mean. MDI does a good job, in the sense that all of the nonnoisy features (either informed or redundant) are ranked higher than the noise features. Still, a small number of nonnoisy features appear to be much more important than their peers. This is the kind of substitution effects that we anticipated to find in the presence of redundant features. Section 6.5 proposes a solution to this particular concern.

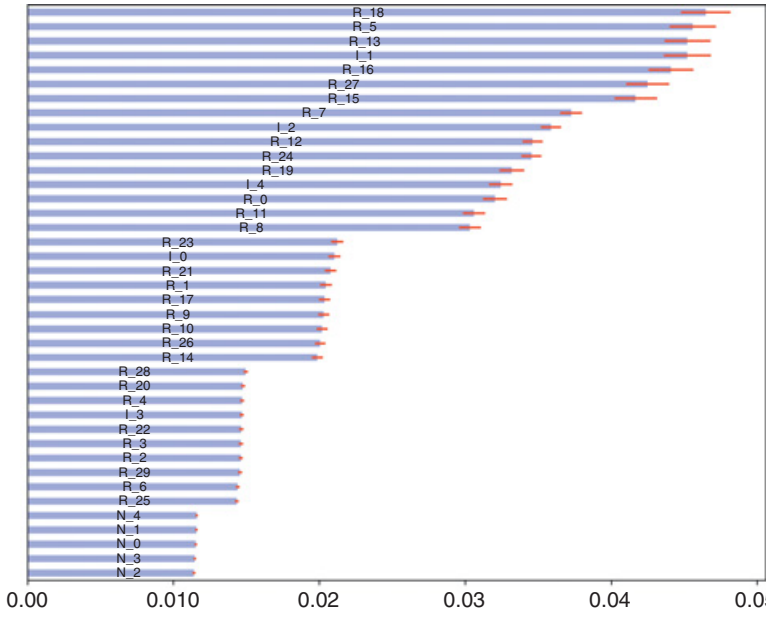


**Figure 6.2** Example of MDI results.

Out of the four caveats of *p*-values, the MDI method deals with three: (1) MDI's computational nature circumvents the need for strong distributional assumptions that could be false (caveat #1) – we are not imposing a particular tree structure or algebraic specification, or relying on stochastic or distributional characteristics of residuals. (2) Whereas betas are estimated on a single sample, ensemble MDIs are derived from a bootstrap of trees. Accordingly, the variance of MDI estimates can be reduced by increasing the number of trees in ensemble methods in general, or in a random forest in particular (caveat 2). This reduces the probability of false positives caused by overfitting. Also, unlike *p*-values, MDI's estimation does not require the inversion of a possibly ill-conditioned matrix. (3) The goal of the tree-based classifiers is not to estimate the coefficients of a given algebraic equation, thus estimating the probability of a particular null hypothesis is irrelevant. In other words, MDI corrects for caveat 3 by finding the important features in general, irrespective of any particular parametric specification.

An ensemble estimate of MDI will exhibit low variance given a sufficient number of trees, hence reducing the concern of *p*-hacking. But still, the procedure itself does not involve cross-validation. Therefore, the one caveat of *p*-values that MDI does not fully solve is that MDI is also computed in-sample (caveat #4). To confront this final caveat, we need to introduce the concept of mean-decrease accuracy.

### 6.3.2 Mean-Decrease Accuracy

A disadvantage of both *p*-values and MDI is that a variable that appears to be significant for explanatory purposes (in-sample) may be irrelevant for forecasting purposes (out-of-sample). To solve this problem (caveat #4), Breiman (2001) introduced the mean-decrease accuracy (MDA) method.[18] MDA works as follows: first, it fits a model and computes its cross-validated performance; second, it computes the cross-validated performance of the same fitted model, with the only difference that it shuffles the observations associated with one of the features. That gives us one modified cross-validated performance per feature. Third, it derives the MDA associated with a particular feature by comparing the cross-validated performance before and after shuffling. If the feature is important, there should be a significant decay in performance caused by the shuffling, as long as the features are independent. An important attribute of MDA is that, like ensemble MDIs, it is not the result of a single estimate, but rather the average of multiple estimates (one for each testing set in a k-fold cross-validation).

When features are not independent, MDA may underestimate the importance of interrelated features. At the extreme, given two highly important but identical features, MDA may conclude that both features are relatively unimportant, because the effect of shuffling one may be partially compensated by not shuffling the other. We address this concern in Section 6.5.

MDA values are not bounded, and shuffling a feature could potentially improve the cross-validated performance, when the feature is uninformative to the point of being detrimental. Because MDA involves a cross-validation step, this method can be computationally expensive. Code Snippet 6.3 implements MDA. See López de Prado (2018a) for practical advice on how to use MDA.

Figure 6.3 plots the result of applying MDA to the same random classification problem we discussed in Figure 6.2. We can draw similar conclusions as we did in the MDI example. First, MDA does a good job overall at separating noise features from the rest. Noise features are ranked last. Second, noise features are also deemed unimportant in magnitude, with MDA values of essentially zero. Third, although substitution effects contribute to higher variances in MDA importance, none is high enough to question the importance of the nonnoisy features.

Despite its name, MDA does not necessarily rely on accuracy to evaluate the cross-validated performance. MDA can be computed on other performance scores. In fact, in the particular case of finance, accuracy is not a particularly good choice. The reason is, accuracy scores a classifier in terms of its proportion of correct predictions. This has the disadvantage that probabilities are not taken into account.

[18] This is sometimes also known as permutation importance.

SNIPPET 6.3 IMPLEMENTATION OF MDA

```
def featImpMDA(clf,X,y,n_splits=10):
  # feat importance based on OOS score reduction
  from sklearn.metrics import log_loss
  from sklearn.model_selection._split import KFold
  cvGen=KFold(n_splits=n_splits)
  scr0,scr1=pd.Series(),pd.DataFrame(columns=X.columns)
  for i,(train,test) in enumerate(cvGen.split(X=X)):
    X0,y0=X.iloc[train,:],y.iloc[train]
    X1,y1=X.iloc[test,:],y.iloc[test]
    fit=clf.fit(X=X0,y=y0) # the fit occurs here
    prob=fit.predict_proba(X1) # prediction before shuffling
    scr0.loc[i]=-log_loss(y1,prob,labels=clf.classes_)
    for j in X.columns:
      X1_=X1.copy(deep=True)
      np.random.shuffle(X1_[j].values) # shuffle one column
      prob=fit.predict_proba(X1_) # prediction after shuffling
      scr1.loc[i,j]=-log_loss(y1,prob,labels=clf.classes_)
  imp=(-1*scr1).add(scr0,axis=0)
  imp=imp/(-1*scr1)
  imp=pd.concat({'mean':imp.mean(),
    'std':imp.std()*imp.shape[0]**-.5},axis=1) # CLT
  return imp
#-----------------------------------------------------
X,y=getTestData(40,5,30,10000,sigmaStd=.1)
clf=DecisionTreeClassifier(criterion='entropy',max_features=1,
  class_weight='balanced',min_weight_fraction_leaf=0)
clf=BaggingClassifier(base_estimator=clf,n_estimators=1000,
  max_features=1.,max_samples=1.,oob_score=False)
imp=featImpMDA(clf,X,y,10)
```

For example, a classifier may achieve high accuracy even though it made good predictions with low confidence and bad predictions with high confidence. In the following section, we introduce a scoring function that addresses this concern.

## 6.4 Probability-Weighted Accuracy

In financial applications, a good alternative to accuracy is log-loss (also known as cross-entropy loss). Log-loss scores a classifier in terms of the average log-likelihood of the true labels (for a formal definition, see section 9.4 of López de Prado 2018a). One disadvantage, however, is that log-loss scores are not easy to interpret and compare. A possible solution is to compute the negative average likelihood of the true labels (NegAL),

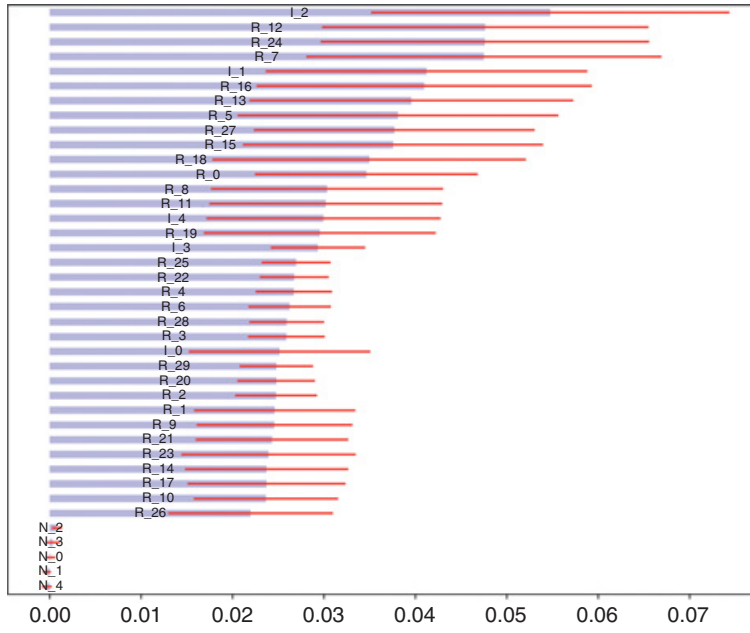


**Figure 6.3** Example of MDA results.

$$\text{NegAL} = -N^{-1}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} y_{n,k}p_{n,k},$$

where $p_{n,k}$ is the probability associated with prediction $n$ of label $k$ and $y_{n,k}$ is an indicator function, $y_{n,k} \in \{0,1\}$, where $y_{n,k} = 1$ when observation $n$ was assigned label $k$ and $y_{n,k} = 0$ otherwise. This is very similar to log-loss, with the difference that it averages likelihoods rather than log-likelihoods, so that NegAL still ranges between 0 and 1.

Alternatively, we can define the probability-weighted accuracy (PWA) as

$$\text{PWA} = \sum_{n=0}^{N-1} y_n\left(p_n - K^{-1}\right) \Bigg/ \sum_{n=0}^{N-1}\left(p_n - K^{-1}\right),$$

where $p_n = \max_k\{p_{n,k}\}$ and $y_n$ is an indicator function, $y_n \in \{0,1\}$, where $y_n = 1$ when the prediction was correct, and $y_n = 0$ otherwise.[19] This is equivalent to standard accuracy when the classifier has absolute conviction in every prediction ($p_n = 1$ for all $n$). PWA punishes bad predictions made with high confidence more severely than accuracy, but less severely than log-loss.

[19] The idea of PWA is the fruit of joint work with my colleagues Lee Cohn, Michael Lock, and Yaxiong Zeng.

## 6.5 Substitution Effects

Substitution effects arise when two features share predictive information. Substitution effects can bias the results from feature importance methods. In the case of MDI, the importance of two identical features will be halved, as they are randomly chosen with equal probability. In the case of MDA, two identical features may be considered relatively unimportant, even if they are critical, because the effect of shuffling one may be compensated by the other.

### *6.5.1 Orthogonalization*

When features are highly codependent, their importance cannot be adjudicated in a robust manner. Small changes in the observations may have a dramatic impact on their estimated importance. However, this impact is not random: given two highly codependent features, the drop in importance from one is compensated with the raise in importance of the other. In other words, codependence causes substitution effects when evaluating the importance of features.

One solution to multicollinearity is to apply PCA on the features, derive their orthogonal principal components, and then run MDI or MDA on those principal components (for additional details, see chapter 8 of López de Prado 2018a). Features orthogonalized in this way may be more resilient to substitution effects, with three caveats: (1) redundant features that result from nonlinear combinations of informative ones will still cause substitution effects; (2) the principal components may not have an intuitive explanation; (3) the principal components are defined by eigenvectors that do not necessarily maximize the model's out-of-sample performance (Witten et al. 2013).

### *6.5.2 Cluster Feature Importance*

A better approach, which does not require a change of basis, is to cluster similar features and apply the feature importance analysis at the cluster level. By construction, clusters are mutually dissimilar, hence taming the substitution effects. Because the analysis is done on a partition of the features, without a change of basis, results are usually intuitive.

Let us introduce one algorithm that implements this idea. The clustered feature importance (CFI) algorithm involves two steps: (1) finding the number and constituents of the clusters of features; (2) applying the feature importance analysis on groups of similar features rather than on individual features.

### Step 1: Features Clustering

First, we project the observed features into a metric space, resulting in a matrix $\{X_f\}_{f=1,\ldots,F}$. To form this matrix, one possibility is to follow the correlation-based approach described in Section 4.4.1. Another possibility is to apply information-theoretic concepts (such as variation of information; see Section 3) to represent those features in a metric space. Information-theoretic metrics have the advantage of recognizing redundant features that are the result of nonlinear combinations of informative features.[20]

Second, we apply a procedure to determine the optimal number and composition of clusters, such as the ONC algorithm (see Section 4). Remember that ONC finds the optimal number of clusters as well as the composition of those clusters, where each feature belongs to one and only one cluster. Features that belong to the same cluster share a large amount of information, and features that belong to different clusters share only a relatively small amount of information.

Some silhouette scores may be low due one feature being a combination of multiple features across clusters. This is a problem, because ONC cannot assign one feature to multiple clusters. In this case, the following transformation may help reduce the multicollinearity of the system. For each cluster $k = 1, \ldots, K$, replace the features included in that cluster with residual features, where those residual features do not contain information from features *outside* cluster $k$. To be precise, let $D_k$ be the subset of index features $D = \{1, \ldots, F\}$ included in cluster $k$, where $D_k \subset \mathrm{D}$, $\| D_k \| > 0, \forall\, k$; $D_k \cap D_l = \varnothing, \forall k \neq l$; $\mathrm{U}_{k=1}^{K} D_k = D$. Then, for a given feature $X_i$ where $i \in D_k$, we compute the residual feature $\hat{\varepsilon}_i$ by fitting

$$X_{n,i} = \alpha_i + \sum_{j \in \left\{ \bigcup_{l<k} D_l \right\}} \beta_{i,j} X_{n,j} + \varepsilon_{n,i}$$

where $n = 1, \ldots, N$ is the index of observations per feature. If the degrees of freedom in the above regression is too low, one option is to use as regressors linear combinations of the features within each cluster (e.g., following a minimum variance weighting scheme), so that only $K - 1$ betas need to be estimated. One of the properties of OLS residuals is that they are orthogonal to the regressors. Thus, by replacing each feature $X_i$ with its residual equivalent $\hat{\varepsilon}_i$, we remove from cluster $k$ information that is already included in other clusters, while preserving the information that exclusively belongs to cluster $k$. Again, this transformation is not necessary if the silhouette scores clearly indicate that features belong to their respective clusters.

[20] For an example of features clustering with an information-theoretic distance metric, see https://ssrn.com/abstract=3517595

### Step 2: Clustered Importance

Step 1 has identified the number and composition of the clusters of features. We can use this information to apply MDI and MDA on groups of similar features, rather than on individual features. In the following, we assume that a partitional algorithm has clustered the features, however this notion of clustered feature importance can be applied to hierarchical clusters as well.

#### *Clustered MDI*

As we saw in Section 6.3.1, the MDI of a feature is the weighted impurity reduction across all nodes where that feature was selected. We compute the clustered MDI as the sum of the MDI values of the features that constitute that cluster. If there is one feature per cluster, then MDI and clustered MDI are the same. In the case of an ensemble of trees, there is one clustered MDI for each tree, which allows us to compute the mean clustered MDI, and standard deviation around the mean clustered MDI, similarly to how we did for the feature MDI. Code Snippet 6.4 implements the procedure that estimates the clustered MDI.

SNIPPET 6.4 CLUSTERED MDI

```
def groupMeanStd(df0,clstrs):
   out=pd.DataFrame(columns=['mean','std'])
   for i,j in clstrs.iteritems():
      df1=df0[j].sum(axis=1)
      out.loc['C_'+str(i),'mean']=df1.mean()
      out.loc['C_'+str(i),'std']=df1.std()*df1.shape[0]**-.5
  return out
#---------------------------------------------------------
def featImpMDI_Clustered(fit,featNames,clstrs):
   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=groupMeanStd(df0,clstrs)
   imp/=imp['mean'].sum()
   return imp
```

#### *Clustered MDA*

The MDA of a feature is computed by comparing the performance of an algorithm before and after shuffling that feature. When computing clustered MDA, instead of shuffling one feature at a time, we shuffle all of the features that constitute a given cluster. If there is one cluster per feature, then MDA and clustered MDA are the

SNIPPET 6.5 CLUSTERED MDA

```
def featImpMDA_Clustered(clf,X,y,clstrs,n_splits=10):
  from sklearn.metrics import log_loss
  from sklearn.model_selection._split import KFold
  cvGen=KFold(n_splits=n_splits)
  scr0,scr1=pd.Series(),pd.DataFrame(columns=clstrs.keys())
  for i,(train,test) in enumerate(cvGen.split(X=X)):
    X0,y0=X.iloc[train,:],y.iloc[train]
    X1,y1=X.iloc[test,:],y.iloc[test]
    fit=clf.fit(X=X0,y=y0)
    prob=fit.predict_proba(X1)
    scr0.loc[i]=-log_loss(y1,prob,labels=clf.classes_)
    for j in scr1.columns:
      X1_=X1.copy(deep=True)
      for k in clstrs[j]:
        np.random.shuffle(X1_[k].values) # shuffle cluster
      prob=fit.predict_proba(X1_)
      scr1.loc[i,j]=-log_loss(y1,prob,labels=clf.classes_)
  imp=(-1*scr1).add(scr0,axis=0)
  imp=imp/(-1*scr1)
  imp=pd.concat({'mean':imp.mean(),
    'std':imp.std()*imp.shape[0]**-.5},axis=1)
  imp.index=['C_'+str(i) for i in imp.index]
  return imp
```

same. Code Snippet 6.5 implements the procedure that estimates the clustered MDA.

## 6.6 Experimental Results

In this experiment we are going to test the clustered MDI and MDA procedures on the same data set we used on the nonclustered versions of MDI and MDA (see Sections 6.3.1 and 6.3.2). That data set consisted of forty features, of which five were informative, thirty were redundant, and five were noise. First, we apply the ONC algorithm to the correlation matrix of those features.[21] In a nonexperimental setting, the researcher should denoise and detone the correlation matrix before clustering, as explained in Section 2. We do not do so in this experiment as a matter of testing the robustness of the method (results are expected to be better on a denoised and detoned correlation matrix).

[21] As an exercise, we ask the reader to apply ONC on a metric projection of the features computed using the normalized variation of information.

Figure 6.4 shows that ONC correctly recognizes that there are six relevant clusters (one cluster for each informative feature, plus one cluster of noise features), and it assigns the redundant features to the cluster that contains the informative feature from which the redundant features were derived. Given the low correlation across clusters, there is no need to replace the features with their residuals (as proposed in Section 6.5.2.1). Code Snippet 6.6 implements this example.

SNIPPET 6.6 FEATURES CLUSTERING STEP

```
X,y=getTestData(40,5,30,10000,sigmaStd=.1)
corr0,clstrs,silh=clusterKMeansBase(X.corr(),maxNumClusters=10,
  n_init=10)
sns.heatmap(corr0,cmap='viridis')
```

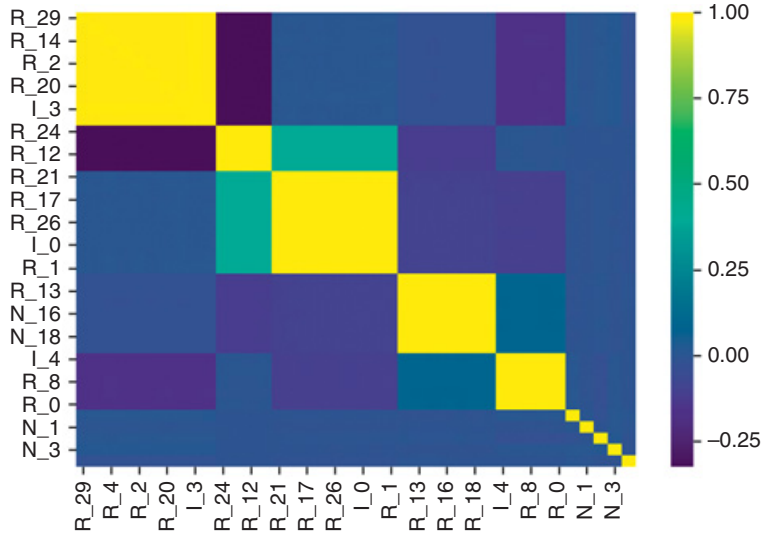


**Figure 6.4** ONC clusters together with informative and redundant features.

Next, we apply our clustered MDI method on that data set. Figure 6.5 shows the clustered MDI output, which we can compare with the unclustered output reported in Figure 6.2. The "C_" prefix indicates the cluster, and "C_5" is the cluster associated with the noise features. Clustered features "C_1" is the second least important, however its importance is more than double the importance of "C_5." This is in contrast with what we saw in Figure 6.2, where there was a small difference in importance between the noise features and some of the nonnoisy features. Thus, the clustered MDI method appears to work better than the standard MDI method. Code Snippet 6.7 shows how these results were computed.

SNIPPET 6.7 CALLING THE FUNCTIONS FOR CLUSTERED MDI

```
clf=DecisionTreeClassifier(criterion='entropy',max_features=1,
  class_weight='balanced',min_weight_fraction_leaf=0)
clf=BaggingClassifier(base_estimator=clf,n_estimators=1000,
  max_features=1.,max_samples=1.,oob_score=False)
fit=clf.fit(X,y)
imp=featImpMDI_Clustered(fit,X.columns,clstrs)
```

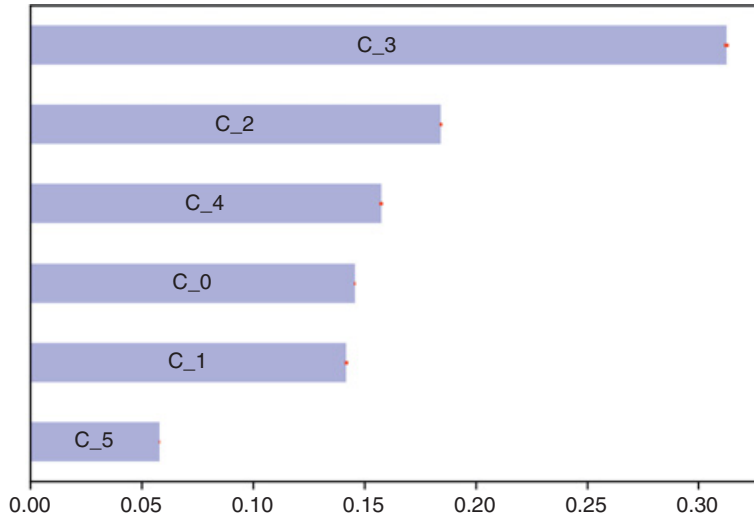


**Figure 6.5** Clustered MDI.

Finally, we apply our clustered MDA method on that data set. Figure 6.6 shows the clustered MDA output, which we can compare with the unclustered output reported in Figure 6.3. Again, "C_5" is the cluster associated with the noise features, and all other clusters are associated with informative and redundant features. This analysis has reached two correct conclusions: (1) "C_5" has essentially zero importance, and should be discarded as irrelevant; and (2) all other clusters have very similar importance. This is in contrast with what we saw in Figure 6.3, where some nonnoise features appeared to be much more important than others, even after taking into consideration the standard derivation around the mean values. Code Snippet 6.8 shows how these results were computed.

## 6.7 Conclusions

Most researchers use *p*-values to evaluate the significance of explanatory variables. However, as we saw in this section, *p*-values suffer from four major flaws. ML offers feature importance methods that overcome most or all of those flaws.

SNIPPET 6.8 CALLING THE FUNCTIONS FOR CLUSTERED MDA

```
clf=DecisionTreeClassifier(criterion='entropy',max_features=1,
  class_weight='balanced',min_weight_fraction_leaf=0)
clf=BaggingClassifier(base_estimator=clf,n_estimators=1000,
  max_features=1.,max_samples=1.,oob_score=False)
imp=featImpMDA_Clustered(clf,X,y,clstrs,10)
```

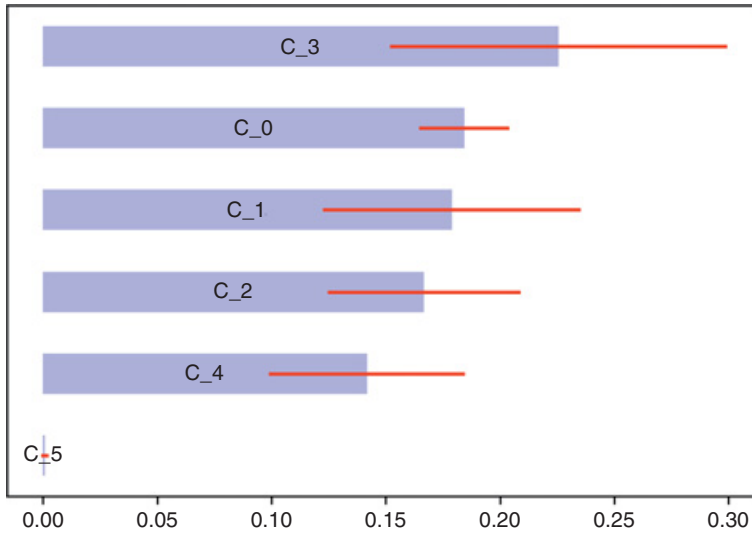


**Figure 6.6** Clustered MDA.

The MDI and MDA methods assess the importance of features robustly and without making strong assumptions about the distribution and structure of the data. Unlike *p*-values, MDA evaluates feature importance in cross-validated experiments. Furthermore, unlike *p*-values, clustered MDI and clustered MDA estimates effectively control for substitution effects. But perhaps the most salient advantage of MDI and MDA is that, unlike classical significance analyses, these ML techniques evaluate the importance of a feature irrespective of any particular specification. In doing so, they provide information that is extremely useful for the development of a theory. Once the researcher knows the variables involved in a phenomenon, she can focus her attention on finding the mechanism or specification that binds them together.

The implication is that classical statistical approaches, such as regression analysis, are not necessarily more transparent or insightful than their ML counterparts. The perception that ML tools are black-boxes and classical tools are white-boxes is false. Not only can ML feature importance methods be as helpful as *p*-values, but in some cases they can be more insightful and accurate.

A final piece of advice is to consider carefully what are we interested in explaining or predicting. In Section 5, we reviewed various labeling methods. The same features can yield various degrees of importance in explaining or predicting different types of labels. Whenever possible, it makes sense to apply these feature importance methods to all of the labeling methods discussed earlier, and see what combination of features and labels leads to the strongest theory. For instance, you may be indifferent between predicting the sign of the next trend or predicting the sign of the next 5% return, because you can build profitable strategies on either kind of prediction (as long as the feature importance analysis suggests the existence of a strong theoretical connection).

## 6.8 Exercises

1 Consider a medical test with a false positive rate $\alpha = \mathrm{P}[x > \tau | H_0]$, where $H_0$ is the null hypothesis (the patient is healthy), $x$ is the observed measurement, and $\tau$ is the significance threshold. A test is run on a random patient and comes back positive (the null hypothesis is rejected). What is the probability that the patient truly has the condition?
   a Is it $1 - \alpha = \mathrm{P}[x \leq \tau | H_0]$ (the confidence of the test)?
   b Is it $1 - \beta = \mathrm{P}[x > \tau | H_1]$ (the power, or recall, of the test)?
   c Or is it $\mathrm{P}[H_1 | x > \tau]$ (the precision of the test)?
   d Of the above, what do $p$-values measure?
   e In finance, the analogous situation is to test whether a variable is involved in a phenomenon. Do $p$-values tell us anything about the probability that the variable is relevant, given the observed evidence?
2 Consider a medical test where $\alpha = .01$, $\beta = 0$, and the probability of the condition is $\mathrm{P}[H_1] = .001$. The test has full recall and a very high confidence. What is the probability that a positive-tested patient is actually sick? Why is it much lower than $1 - \alpha$ and $1 - \beta$? What is the probability that a patient is actually sick after testing positive twice on independent tests?
3 Rerun the examples in Sections 6.3.1 and 6.3.2, where this time you pass an argument sigmaStd=0 to the getTestData function. How do Figures 6.2 and 6.3 look now? What causes the difference, if there is one?
4 Rerun the MDA analysis in Section 6.3.2, where this time you use probability-weighted accuracy (Section 6.4) as the scoring function. Are results materially different? Are they more intuitive or easier to explain? Can you think of other ways to represent MDA outputs using probability-weighted accuracy?
5 Rerun the experiment in Section 6.6, where this time the distance metric used to cluster the features is variation of information (Section 3).

# 7 Portfolio Construction

## 7.1 Motivation

The allocation of assets requires making decisions under uncertainty. Markowitz (1952) proposed one of the most influential ideas in modern financial history, namely, the representation of the problem of investing as a convex optimization program. Markowitz's Critical Line Algorithm (CLA) estimates an "efficient frontier" of portfolios that maximize the expected return subject to a given level of risk, where portfolio risk is measured in terms of the standard deviation of returns. In practice, mean-variance optimal solutions tend to be concentrated and unstable (De Miguel et al. 2009).

There are three popular approaches to reducing the instability in optimal portfolios. First, some authors attempted to regularize the solution, by injecting additional information regarding the mean or variance in the form of priors (Black and Litterman 1992). Second, other authors suggested reducing the solution's feasibility region by incorporating additional constraints (Clarke et al. 2002). Third, other authors proposed improving the numerical stability of the covariance matrix's inverse (Ledoit and Wolf 2004).

In Section 2, we discussed how to deal with the instability caused by the noise contained in the covariance matrix. As it turns out, the signal contained in the covariance matrix can also be a source of instability, which requires a specialized treatment. In this section, we explain why certain data structures (or types of signal) make mean-variance solutions unstable, and what we can do to address this second source of instability.

## 7.2 Convex Portfolio Optimization

Consider a portfolio of $N$ holdings, where its returns in excess of the risk-free rate have an expected value $\mu$ and an expected covariance $V$. Markowitz's insight was to formulate the classical asset allocation problem as a quadratic program,

$$\min_{\omega} \frac{1}{2}\omega' V \omega$$

$$\text{s.t.} : \omega' a = 1,$$

where $a$ characterizes the portfolio's constraints. This problem can be expressed in Lagrangian form as

$$L[\omega, \lambda] = \frac{1}{2}\omega' V \omega - \lambda(\omega' a - 1)$$

with first-order conditions

$$\frac{\partial L[\omega,\lambda]}{\partial \omega} = V\omega - \lambda a$$

$$\frac{\partial L[\omega,\lambda]}{\partial \lambda} = \omega' a - 1.$$

Setting the first-order (necessary) conditions to zero, we obtain that $V\omega - \lambda a = 0 \Rightarrow \omega = \lambda V^{-1} a$ and $\omega' a = a'\omega = 1 \Rightarrow \lambda a' V^{-1} a = 1 \Rightarrow \lambda = 1/\left(a' V^{-1} a\right)$, thus

$$\omega^* = \frac{V^{-1} a}{a' V^{-1} a}.$$

The second-order (sufficient) condition confirms that this solution is the minimum of the Lagrangian:

$$\begin{vmatrix} \frac{\partial L^2[\omega,\lambda]}{\partial \omega^2} & \frac{\partial L^2[\omega,\lambda]}{\partial \omega \partial \lambda} \\ \frac{\partial L^2[\omega,\lambda]}{\partial \lambda \partial \omega} & \frac{\partial L^2[\omega,\lambda]}{\partial \lambda^2} \end{vmatrix} = \begin{vmatrix} V' & -a' \\ a & 0 \end{vmatrix} = a' a \geq 0.$$

Let us now turn our attention to a few formulations of the characteristic vector, $a$:

1 For $a = 1_N$ and $V = \sigma I_N$, where $\sigma \in \mathbb{R}^+$, $1_N$ is a vector of ones of size $N$, and $I_N$ is an identity matrix of size $N$, then the solution is the equal weights portfolio (known as the "1/N" portfolio, or the "naïve" portfolio), because $\omega^* = 1_N \sigma^{-1} / \left(N\sigma^{-1}\right) = 1_N / N$.
2 For $a = 1_N$ and $V$ is a diagonal matrix with unequal entries ($V_{i,j} = 0$, for all $i \neq j$), then the solution is the inverse-variance portfolio, because $\omega^* = \frac{1}{\sum_{n=1}^{N} \frac{1}{V_{n,n}}} \left\{\frac{1}{V_{n,n}}\right\}_{n=1,\ldots,N}$.
3 For $a = 1_N$, the solution is the minimum variance portfolio.
4 For $a = \mu$, the solution maximizes the portfolio's Sharpe ratio, $\omega'\mu / \sqrt{\omega' V \omega}$, and the market portfolio is $V^{-1}\mu / (1_N' V^{-1} \mu)$ (Grinold and Kahn 1999).

## 7.3 The Condition Number

Certain covariance structures can make the mean-variance optimization solution unstable. To understand why, we need to introduce the concept of condition number of a covariance matrix. Consider a correlation matrix between two securities,

$$C = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix},$$

where $\rho$ is the correlation between their returns. Matrix $C$ can be diagonalized as $CW = W\Lambda$ as follows. First, we set the eigenvalue equation $|C - I\lambda| = 0$. Operating,

$$\begin{vmatrix} 1-\lambda & \rho \\ \rho & 1-\lambda \end{vmatrix} = 0 \Rightarrow \quad (1-\lambda)^2 - \rho^2 = 0.$$

This equation has roots in $\lambda = 1 \pm \rho$, hence the diagonal elements of $\Lambda$ are

$$\Lambda_{1,1} = 1 + \rho$$

$$\Lambda_{2,2} = 1 - \rho.$$

Second, the eigenvector associated with each eigenvalue is given by the solution to the system

$$\begin{bmatrix} 1-\Lambda_{1,1} & \rho \\ \rho & 1-\Lambda_{2,2} \end{bmatrix} \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

If $C$ is not already a diagonal matrix, then $\rho \neq 0$, in which case the above system has solutions in

$$\begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix},$$

and it is easy to verify that

$$W\Lambda W' = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1+\rho & 0 \\ 0 & 1-\rho \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}' = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix} = C.$$

The trace of $C$ is $tr(C) = \Lambda_{1,1} + \Lambda_{2,2} = 2$, so $\rho$ sets how big one eigenvalue gets at the expense of the other. The determinant of $C$ is given by $|C| = \Lambda_{1,1}\Lambda_{2,2} = (1+\rho)(1-\rho) = 1 - \rho^2$. The determinant reaches its maximum at $\Lambda_{1,1} = \Lambda_{2,2} = 1$, which corresponds to the uncorrelated case, $\rho = 0$. The determinant reaches its minimum at $\Lambda_{1,1} = 0$ or $\Lambda_{2,2} = 0$, which corresponds to the perfectly correlated case, $|\rho| = 1$. The inverse of $C$ is

$$C^{-1} = W\Lambda^{-1}W' = \frac{1}{|C|}\begin{bmatrix} 1 & -\rho \\ -\rho & 1 \end{bmatrix}.$$

The implication is that the more $\rho$ deviates from zero, the bigger one eigenvalue becomes relative to the other, causing $|C|$ to approach zero, which makes the values of $C^{-1}$ explode.

More generally, the instability caused by covariance structure can be measured in terms of the magnitude between the two extreme eigenvalues. Accordingly, the condition number of a covariance or correlation (or normal, thus diagonalizable) matrix is defined as the absolute value of the ratio between its maximal and minimal (by moduli) eigenvalues. In the above example,

$$\lim_{\rho \to 1^-} \frac{\Lambda_{1,1}}{\Lambda_{2,2}} = +\infty$$

$$\lim_{\rho \to -1^+} \frac{\Lambda_{2,2}}{\Lambda_{1,1}} = +\infty.$$

## 7.4 Markowitz's Curse

Matrix $C$ is just a standardized version of $V$, and the conclusions we drew on $C^{-1}$ apply to the $V^{-1}$ used to compute $\omega^*$. When securities within a portfolio are highly correlated ($-1 < \rho \ll 0$ or $0 \ll \rho < 1$), $C$ has a high condition number, and the values of $V^{-1}$ explode. This is problematic in the context of portfolio optimization, because $\omega^*$ depends on $V^{-1}$, and unless $\rho \approx 0$, we must expect an unstable solution to the convex optimization program. In other words, Markowitz's solution is guaranteed to be numerically stable only if $\rho \approx 0$, which is precisely the case when we don't need it! The reason we needed Markowitz was to handle the $\rho \not\approx 0$ case, but the more we need Markowitz, the more numerically unstable is the estimation of $\omega^*$. This is Markowitz's curse.

López de Prado (2016) introduced an ML-based asset allocation method called hierarchical risk parity (HRP). HRP outperforms Markowitz and the naïve allocation in out-of-sample Monte Carlo experiments. The purpose of HRP was not to deliver an optimal allocation, but merely to demonstrate the potential of ML approaches. In fact, HRP outperforms Markowitz out-of-sample even though HRP is by construction suboptimal in-sample. In the next section we analyze further why standard mean-variance optimization is relatively easy to beat.

## 7.5 Signal as a Source of Covariance Instability

In Section 2, we saw that the covariance instability associated with noise is regulated by the $N/T$ ratio, because the lower bound of the Marcenko–Pastur distribution, $\lambda_-$, gets smaller as $N/T$ grows,[22] while the upper bound, $\lambda_+$, increases as $N/T$ grows. In this section, we are dealing with a different source of covariance instability, caused by the structure of the data (signal). As we saw in the $2 \times 2$ matrix example, $\rho$ regulates the matrix's condition number, regardless and independently from $N/T$. Signal-induced instability is structural, and cannot be reduced by sampling more observations.

There is an intuitive explanation for how signal makes mean-variance optimization unstable. When the correlation matrix is an identity matrix, the eigenvalue function is a horizontal line, and the condition number is 1. Outside that ideal case, the condition number is impacted by irregular correlation structures. In the particular case of finance, when a subset of securities exhibits greater correlation among themselves than to the rest of the investment universe, that subset forms a cluster within the correlation matrix. Clusters appear naturally, as a consequence of hierarchical relationships. When $K$ securities form a cluster, they are more heavily exposed to a common eigenvector, which implies that the associated eigenvalue explains a greater amount of variance. But because the trace of the correlation matrix is exactly $N$, that means that an eigenvalue can only increase at the expense of the other $K - 1$ eigenvalues in that cluster, resulting in a condition number greater than 1. Consequently, the greater the intracluster correlation is, the higher the condition number becomes. This source of instability is distinct and unrelated to $N/T \to 1$.

Let us illustrate this intuition with a numerical example. Code Snippet 7.1 shows how to form a block-diagonal correlation matrix of different numbers of blocks, block sizes, and intrablock correlations. Figure 7.1 plots a block-diagonal

SNIPPET 7.1 COMPOSITION OF BLOCK-DIAGONAL CORRELATION MATRICES

```
import matplotlib.pyplot as mpl,seaborn as sns
import numpy as np
#-------------------------------------------------------
corr0=formBlockMatrix(2,2,.5)
eVal,eVec=np.linalg.eigh(corr0)
print max(eVal)/min(eVal)
sns.heatmap(corr0,cmap='viridis')
```

[22] As a reminder, in Section 2, variable $N$ denoted the number of columns in the covariance matrix, and variable $T$ denoted the number of independent observations used to compute the covariance matrix.

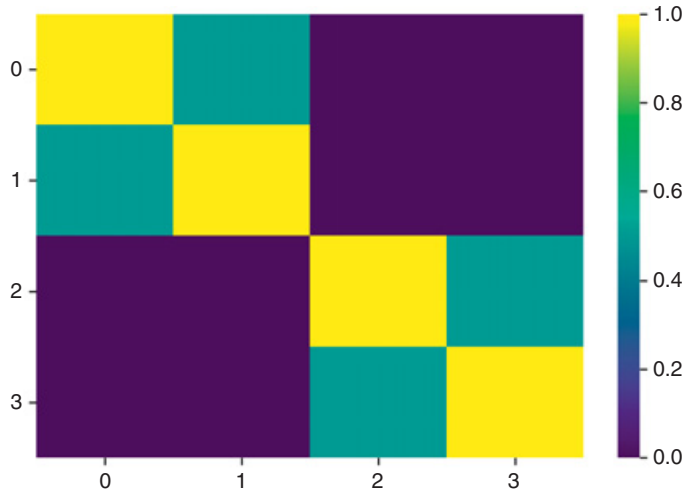


**Figure 7.1** Heatmap of a block-diagonal correlation matrix.

matrix of size 4*x*4, composed of two equal-sized blocks, where the intrablock correlation is 0.5 and the outer-block correlation is zero. Because of this block structure, the condition number is not 1, but 3. The condition number rises if (1) we make one block greater or (2) we increase the intrablock correlation. The reason is, in both cases one eigenvector explains more variance than the rest. For instance, if we increase the size of one block to three and reduce the size of the other to 1, the condition number becomes 4. If instead we increase the intrablock correlation to 0.75, the condition number becomes 7. A block-diagonal correlation matrix of size 500*x*500 with two equal-sized blocks, where the intrablock correlation is 0.5 has a condition number of 251, again as a result of having 500 eigenvectors where most of the variance is explained by only 2.

Code Snippet 7.2 demonstrates that bringing down the intrablock correlation in only one of the two blocks does not reduce the condition number. The reason is, the extreme eigenvalues are caused by the dominant block. So even though the high condition number may be caused by only one cluster, it impacts the entire correlation matrix. This observation has an important implication: the instability of Markowitz's solution can be traced back to a few dominant

SNIPPET 7.2 BLOCK-DIAGONAL CORRELATION MATRIX WITH A DOMINANT BLOCK

```
corr0=block_diag(formBlockMatrix(1,2,.5))
corr1=formBlockMatrix(1,2,.0)
corr0=block_diag(corr0,corr1)
eVal,eVec=np.linalg.eigh(corr0)
print max(eVal)/min(eVal)
```

clusters within the correlation matrix. We can contain that instability by optimizing the dominant clusters separately, hence preventing that the instability spreads throughout the entire portfolio.

## 7.6 The Nested Clustered Optimization Algorithm

The remainder of this section is dedicated to introducing a new ML-based method, named nested clustered optimization (NCO), which tackles the source of Markowitz's curse. NCO belongs to a class of algorithms known as "wrappers": it is agnostic as to what member of the efficient frontier is computed, or what set of constraints is imposed. NCO provides a strategy for addressing the effect of Markowitz's curse on an existing mean-variance allocation method.

### *7.6.1 Correlation Clustering*

The first step of the NCO algorithm is to cluster the correlation matrix. This operation involves finding the optimal number of clusters. One possibility is to apply the ONC algorithm (Section 4), however NCO is agnostic as to what particular algorithm is used for determining the number of clusters. For large matrices, where $T/N$ is relatively low, it is advisable to denoise the correlation matrix prior to clustering, following the method described in Section 2. Code Snippet 7.3 implements this procedure. We compute the denoised covariance matrix, cov1, using the deNoiseCov function introduced in Section 2. As a reminder, argument q informs the ratio between the number of rows and the number of columns in the observation matrix. When bWidth=0, the covariance matrix is not denoised. We standardize the resulting covariance matrix into a correlation matrix using the cov2corr function. Then we cluster the cleaned correlation matrix using the clusterKMeansBase function, which we introduced in Section 4. The argument maxNumClusters is set to half the number of columns in the correlation matrix. The reason is, single-item clusters do not cause an increase in the matrix's condition number, so we

SNIPPET 7.3 THE CORRELATION CLUSTERING STEP

```
import pandas as pd
cols=cov0.columns
cov1=deNoiseCov(cov0,q,bWidth=.01) # de-noise cov
cov1=pd.DataFrame(cov1,index=cols,columns=cols)
corr1=cov2corr(cov1)
corr1,clstrs,silh=clusterKMeansBase(corr1,
  maxNumClusters=corr0.shape[0]/2,n_init=10)
```

only need to consider clusters with a minimum size of two. If we expect fewer clusters, a lower maxNumClusters may be used to accelerate calculations.

A common question is whether we should cluster corr1 or corr1.abs(). When all correlations are nonnegative, clustering corr1 and corr1.abs() yields the same outcome. When some correlations are negative, the answer is more convoluted, and depends on the numerical properties of the observed inputs. I recommend that you try both, and see what clustering works better for your particular corr1 in Monte Carlo experiments.[23]

### 7.6.2 Intracluster Weights

The second step of the NCO algorithm is to compute optimal intracluster allocations, using the denoised covariance matrix, cov1. Code Snippet 7.4 implements this procedure. For simplicity purposes, we have defaulted to a minimum variance allocation, as implemented in the minVarPort function. However, nothing in the procedure prevents the use of alternative allocation methods. Using the estimated intracluster weights, we can derive the reduced covariance matrix, cov2, which reports the correlations between clusters.

SNIPPET 7.4 INTRACLUSTER OPTIMAL ALLOCATIONS

```
wIntra=pd.DataFrame(0,index=cov1.index,columns=clstrs.keys())
for i in clstrs:
   wIntra.loc[clstrs[i],i]=minVarPort(cov1.loc[clstrs[i],
     clstrs[i]]).flatten()
cov2=wIntra.T.dot(np.dot(cov1,wIntra)) # reduced covariance matrix
```

### 7.6.3 Intercluster Weights

The third step of the NCO algorithm is to compute optimal intercluster allocations, using the reduced covariance matrix, cov2. By construction, this covariance matrix is close to a diagonal matrix, and the optimization problem is close to the ideal Markowitz case. In other words, the clustering and intracluster optimization steps have allowed us to transform a "Markowitz-cursed" problem ($|\rho| \gg 0$) into a well-behaved problem ($\rho \approx 0$).

[23] As a rule of thumb, corr1.abs() tends to work better in long-short portfolio optimization problems where some correlations are negative. Intuitively, the ability to have negative weights is equivalent to flipping the sign of the correlation, which can induce considerable instability. Because negatively correlated variables will interact through the weights, it makes sense to cluster those variables together, thus containing that source of instability within each cluster.

Code Snippet 7.5 implements this procedure. It applies the same allocation procedure that was used in the intracluster allocation step (that is, in the case of Code Snippet 7.4, the minVarPort function). The final allocation per security is reported by the wAll0 data frame, which results from multiplying intracluster weights with the intercluster weights.

SNIPPET 7.5 INTERCLUSTER OPTIMAL ALLOCATIONS

```
wInter=pd.Series(minVarPort(cov2).flatten(),index=cov2.index)
wAll0=wIntra.mul(wInter,axis=1).sum(axis=1).sort_index()
```

## 7.7 Experimental Results

In this section we subject the NCO algorithm to controlled experiments, and compare its performance to Markowitz's approach. Like in Section 2, we discuss two characteristic portfolios of the efficient frontier, namely, the minimum variance and maximum Sharpe ratio solutions, since any member of the unconstrained efficient frontier can be derived as a convex combination of the two (a result sometimes known as the "separation theorem").

Code Snippet 7.6 implements the NCO algorithm introduced earlier in this section. When argument mu is None, function optPort_nco returns the minimum variance portfolio, whereas when mu is not None, function optPort_nco returns the maximum Sharpe ratio portfolio.

SNIPPET 7.6 FUNCTION IMPLEMENTING THE NCO ALGORITHM

```
def optPort_nco(cov,mu=None,maxNumClusters=None):
  cov=pd.DataFrame(cov)
  if mu is not None:mu=pd.Series(mu[:,0])
  corr1=cov2corr(cov)
  corr1,clstrs,_=clusterKMeansBase(corr1,maxNumClusters,
    n_init=10)
  wIntra=pd.DataFrame(0,index=cov.index,columns=clstrs.keys())
  for i in clstrs:
    cov_=cov.loc[clstrs[i],clstrs[i]].values
    if mu is None:mu_=None
    else:mu_=mu.loc[clstrs[i]].values.reshape(-1,1)
    wIntra.loc[clstrs[i],i]=optPort(cov_,mu_).flatten()
  cov_=wIntra.T.dot(np.dot(cov,wIntra)) # reduce covariance matrix
  mu_=(None if mu is None else wIntra.T.dot(mu))
  wInter=pd.Series(optPort(cov_,mu_).flatten(),index=cov_.index)
  nco=wIntra.mul(wInter,axis=1).sum(axis=1).values.reshape(-1,1)
  return nco
```

### 7.7.1 Minimum Variance Portfolio

Code Snippet 7.7 creates a random vector of means and a random covariance matrix that represent a stylized version of a fifty securities portfolio, grouped in ten blocks with intracluster correlations of 0.5. This vector and matrix characterize the "true" process that generates observations.[24] We set a seed for the purpose of reproducing and comparing results across runs with different parameters. Function formTrueMatrix was declared in Section 2.

SNIPPET 7.7 DATA-GENERATING PROCESS

```
nBlocks,bSize,bCorr=10,50,.5
np.random.seed(0)
mu0,cov0=formTrueMatrix(nBlocks,bSize,bCorr)
```

Code Snippet 7.8 uses function simCovMu to simulate a random empirical vector of means and a random empirical covariance matrix based on 1,000 observations drawn from the true process (declared in Section 2). When shrink=True, the empirical covariance matrix is subjected to Ledoit–Wolf shrinkage. Using that empirical covariance matrix, function optPort (also declared in Section 2) estimates the minimum variance portfolio according to Markowitz, and function optPort_nco estimates the minimum variance portfolio applying the NCO algorithm. This procedure is repeated on 1,000 different random empirical covariance matrices. Note that, because minVarPortf=True, the random empirical vectors of means are discarded.

SNIPPET 7.8 DRAWING AN EMPIRICAL VECTOR OF MEANS AND COVARIANCE MATRIX

```
nObs,nSims,shrink,minVarPortf=1000,1000,False,True
np.random.seed(0)
for i in range(nSims):
  mu1,cov1=simCovMu(mu0,cov0,nObs,shrink=shrink)
  if minVarPortf:mu1=None
  w1.loc[i]=optPort(cov1,mu1).flatten()
  w1_d.loc[i]=optPort_nco(cov1,mu1,
    int(cov1.shape[0]/2)).flatten()
```

[24] In practical applications, we do not need to simulate $\{\mu, V\}$, as these inputs are estimated from observed data. The reader can repeat this experiment on a pair of observed $\{\mu, V\}$ and evaluate via Monte Carlo the estimation error of alternative optimization methods on those particular inputs, thus finding out what method yields most robust estimates for a particular input.

Code Snippet 7.9 computes the true minimum variance portfolio, derived from the true covariance matrix. Using those allocations as benchmark, it then computes the root-mean-square errors (RMSE) across all weights. We can run Code Snippet 7.9 with and without shrinkage, thus obtaining the four combinations displayed in Figure 7.2.

SNIPPET 7.9 ESTIMATION OF ALLOCATION ERRORS

```
w0=optPort(cov0,None if minVarPortf else mu0)
w0=np.repeat(w0.T,w1.shape[0],axis=0) # true allocation
rmsd=np.mean((w1-w0).values.flatten()**2)**.5 # RMSE
rmsd_d=np.mean((w1_d-w0).values.flatten()**2)**.5 # RMSE
```

| | Markowitz | NCO |
|---|---|---|
| Raw | 7.95E-03 | 4.21E-03 |
| Shrunk | 8.89E-03 | 6.74E-03 |

**Figure 7.2** RMSE for the minimum variance portfolio.

NCO computes the minimum variance portfolio with 52.98% of Markowitz's RMSE, i.e., a 47.02% reduction in the RMSE. While Ledoit–Wolf shrinkage helps reduce the RMSE, that reduction is relatively small, around 11.81%. Combining shrinkage and NCO yields a 15.30% reduction in RMSE, which is better than shrinkage but worse than NCO alone.

The implication is that NCO delivers substantially lower RMSE than Markowitz's solution, even for a small portfolio of only fifty securities, and that shrinkage adds no value. It is easy to test that NCO's advantage widens for larger portfolios (we leave it as an exercise).

### 7.7.2 Maximum Sharpe Ratio Portfolio

By setting minVarPortf=False, we can rerun Code Snippets 7.8 and 7.9 to derive the RMSE associated with the maximum Sharpe ratio portfolio. Figure 7.3 reports the results from this experiment.

NCO computes the maximum Sharpe ratio portfolio with 45.17% of Markowitz's RMSE, i.e., a 54.83% reduction in the RMSE. The combination of shrinkage and NCO yields a 18.52% reduction in the RMSE of the maximum Sharpe ratio portfolio, which is better than shrinkage but worse than NCO. Once again, NCO delivers substantially lower RMSE than Markowitz's solution, and shrinkage adds no value.

| | Markowitz | NCO |
|---|---|---|
| Raw | 7.02E-02 | 3.17E-02 |
| Shrunk | 6.54E-02 | 5.72E-02 |

**Figure 7.3** RMSE for the maximum Sharpe ratio portfolio.

## 7.8 Conclusions

Markowitz's portfolio optimization framework is mathematically correct, however its practical application suffers from numerical problems. In particular, financial covariance matrices exhibit high condition numbers due to noise and signal. The inverse of those covariance matrices magnifies estimation errors, which leads to unstable solutions: changing a few rows in the observations matrix may produce entirely different allocations. Even if the allocations estimator is unbiased, the variance associated with these unstable solutions inexorably leads to large transaction costs than can erase much of the profitability of these strategies.

In this section, we have traced back the source of Markowitz's instability problems to the shape of the correlation matrix's eigenvalue function. Horizontal eigenvalue functions are ideal for Markowitz's framework. In finance, where clusters of securities exhibit greater correlation among themselves than to the rest of the investment universe, eigenvalue functions are not horizontal, which in turn is the cause for high condition numbers. Signal is the cause of this type of covariance instability, not noise.

We have introduced the NCO algorithm to address this source of instability, by splitting the optimization problem into several problems: computing one optimization per cluster, and computing one final optimization across all clusters. Because each security belongs to one cluster and one cluster only, the final allocation is the product of the intracluster and intercluster weights. Experimental results demonstrate that this dual clustering approach can significantly reduce Markowitz's estimation error. The NCO algorithm is flexible and can be utilized in combination with any other framework, such as Black–Litterman, shrinkage, reversed optimization, or constrained optimization approaches. We can think of NCO as a strategy for splitting the general optimization problem into subproblems, which can then be solved using the researcher's preferred method.

Like many other ML algorithms, NCO is flexible and modular. For example, when the correlation matrix exhibits a strongly hierarchical structure, with

clusters within clusters, we can apply the NCO algorithm within each cluster and subcluster, mimicking the matrix's tree-like structure. The goal is to contain the numerical instability at each level of the tree, so that the instability within a subcluster does not extend to its parent cluster or the rest of the correlation matrix.

We can follow the Monte Carlo approach outlined in this section to estimate the allocation error produced by various optimization methods on a particular set of input variables. The result is a precise determination of what method is most robust to a particular case. Thus, rather than relying always on one particular approach, we can apply opportunistically whatever optimization method is best suited in a particular setting.

## 7.9 Exercises

1 Add to Code Snippet 7.3 a detoning step, and repeat the experimental analysis conducted in Section 7.7. Do you see an additional improvement in NCO's performance? Why?
2 Repeat Section 7.7, where this time you generate covariance matrices without a cluster structure, using the function getRndCov listed in Section 2. Do you reach a qualitatively different conclusion? Why?
3 Repeat Section 7.7, where this time you replace the minVarPort function with the CLA class listed in Bailey and López de Prado (2013).
4 Repeat Section 7.7 for a covariance matrix of size ten and for a covariance matrix of size one hundred. How do NCO's results compare to Markowitz's as a function of the problem's size?
5 Repeat Section 7.7, where you purposely mislead the clusterKMeansBase algorithm by setting its argument maxNumClusters to a very small value, like 2. By how much does NCO's solution worsen? How is it possible that, even with only two clusters (instead of ten), NCO performs significantly better than Markowitz's solution?