CHAPTER 6

Supervised Learning: Classification

Here are some of the key questions that financial analysts attempt to solve:

- Is a borrower going to repay their loan or default on it?
- Will the instrument price go up or down?
- Is this credit card transaction a fraud or not?

All of these problem statements, in which the goal is to predict the categorical class labels, are inherently suitable for classification-based machine learning.

Classification-based algorithms have been used across many areas within finance that require predicting a qualitative response. These include fraud detection, default prediction, credit scoring, directional forecasting of asset price movement, and buy/sell recommendations. There are many other use cases of classification-based supervised learning in portfolio management and algorithmic trading.

In this chapter we cover three such classification-based case studies that span a diverse set of areas, including fraud detection, loan default probability, and formulating a trading strategy.

In "Case Study 1: Fraud Detection" on page 153, we use a classification-based algorithm to predict whether a transaction is fraudulent. The focus of this case study is also to deal with an unbalanced dataset, given that the fraud dataset is highly unbalanced with a small number of fraudulent observations.

In "Case Study 2: Loan Default Probability" on page 166, we use a classification-based algorithm to predict whether a loan will default. The case study focuses on various techniques and concepts of data processing, feature selection, and exploratory analysis.

In "Case Study 3: Bitcoin Trading Strategy" on page 179, we use classification-based algorithms to predict whether the current trading signal of bitcoin is to buy or sell depending on the relationship between the short-term and long-term price. We predict the trend of bitcoin's price using technical indicators. The prediction model can easily be transformed into a trading bot that can perform buy, sell, or hold actions without human intervention.

In addition to focusing on different problem statements in finance, these case studies will help you understand:

- How to develop new features such as technical indicators for an investment strategy using feature engineering, and how to improve model performance.
- How to use data preparation and data transformation, and how to perform feature reduction and use feature importance.
- How to use data visualization and exploratory data analysis for feature reduction and to improve model performance.
- How to use algorithm tuning and grid search across various classification-based models to improve model performance.
- How to handle unbalanced data.
- How to use the appropriate evaluation metrics for classification.

This Chapter's Code Repository

A Python-based master template for supervised classification model, along with the Jupyter notebook for the case studies presented in this chapter, is included in the folder *Chapter 6 - Sup. Learning - Classification models* in the code repository for this book. All of the case studies presented in this chapter use the standardized seven-step model development process presented in Chapter 2.[1]

For any new classification-based problem, the master template from the code repository can be modified with the elements specific to the problem. The templates are designed to run on cloud infrastructure (e.g., Kaggle, Google Colab, or AWS). In order to run the template on the local machine, all the packages used within the template must be installed successfully.

1 There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

Case Study 1: Fraud Detection

Fraud is one of the most significant issues the finance sector faces. It is incredibly costly. According to one study, it is estimated that the typical organization loses 5% of its annual revenue to fraud each year. When applied to the 2017 estimated Gross World Product of $79.6 trillion, this translates to potential global losses of up to $4 trillion.

Fraud detection is a task inherently suitable for machine learning, as machine learning–based models can scan through huge transactional datasets, detect unusual activity, and identify all cases that might be prone to fraud. Also, the computations of these models are faster compared to traditional rule-based approaches. By collecting data from various sources and then mapping them to trigger points, machine learning solutions are able to discover the rate of defaulting or fraud propensity for each potential customer and transaction, providing key alerts and insights for the financial institutions.

In this case study, we will use various classification-based models to detect whether a transaction is a normal payment or a fraud.

The focuses of this case study are:

- Handling unbalanced data by downsampling/upsampling the data.
- Selecting the right evaluation metric, given that one of the main goals is to reduce false negatives (cases in which fraudulent transactions incorrectly go unnoticed).

Blueprint for Using Classification Models to Determine Whether a Transaction Is Fraudulent

1. Problem definition

In the classification framework defined for this case study, the response (or target) variable has the column name "Class." This column has a value of 1 in the case of fraud and a value of 0 otherwise.

The dataset used is obtained from Kaggle. This dataset holds transactions by European cardholders that occurred over two days in September 2013, with 492 cases of fraud out of 284,807 transactions.

The dataset has been anonymized for privacy reasons. Given that certain feature names are not provided (i.e., they are called V1, V2, V3, etc.), the visualization and feature importance will not give much insight into the behavior of the model.

By the end of this case study, readers will be familiar with a general approach to fraud modeling, from gathering and cleaning data to building and tuning a classifier.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of the libraries used for data loading, data analysis, data preparation, model evaluation, and model tuning are shown below. The packages used for different purposes have been separated in the Python code below. The details of most of these packages and functions have been provided in Chapter 2 and Chapter 4:

`Packages for data loading, data analysis, and data preparation`

```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot

from pandas import read_csv, set_option
from pandas.plotting import scatter_matrix
from sklearn.preprocessing import StandardScaler
```

`Packages for model evaluation and classification models`

```
from sklearn.model_selection import train_test_split, KFold,\
 cross_val_score, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier,
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.metrics import classification_report, confusion_matrix,\
  accuracy_score
```

`Packages for deep learning models`

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
```

```
Packages for saving the model
```

```
from pickle import dump
from pickle import load
```

3. Exploratory data analysis

The following sections walk through some high-level data inspection.

3.1. Descriptive statistics. The first thing we must do is gather a basic sense of our data. Remember, except for the transaction and amount, we do not know the names of other columns. The only thing we know is that the values of those columns have been scaled. Let's look at the shape and columns of the data:

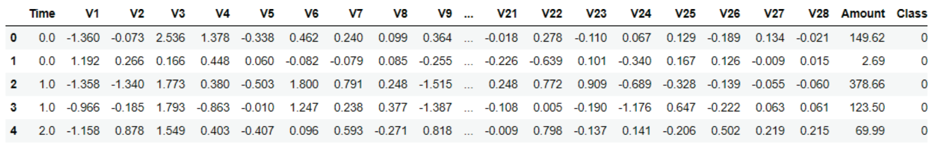
```
# shape
dataset.shape
```

```
Output
```

```
(284807, 31)
```

```
#peek at data
set_option('display.width', 100)
dataset.head(5)
```

```
Output
```



	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.360	-0.073	2.536	1.378	-0.338	0.462	0.240	0.099	0.364	...	-0.018	0.278	-0.110	0.067	0.129	-0.189	0.134	-0.021	149.62	0
1	0.0	1.192	0.266	0.166	0.448	0.060	-0.082	-0.079	0.085	-0.255	...	-0.226	-0.639	0.101	-0.340	0.167	0.126	-0.009	0.015	2.69	0
2	1.0	-1.358	-1.340	1.773	0.380	-0.503	1.800	0.791	0.248	-1.515	...	0.248	0.772	0.909	-0.689	-0.328	-0.139	-0.055	-0.060	378.66	0
3	1.0	-0.966	-0.185	1.793	-0.863	-0.010	1.247	0.238	0.377	-1.387	...	-0.108	0.005	-0.190	-1.176	0.647	-0.222	0.063	0.061	123.50	0
4	2.0	-1.158	0.878	1.549	0.403	-0.407	0.096	0.593	-0.271	0.818	...	-0.009	0.798	-0.137	0.141	-0.206	0.502	0.219	0.215	69.99	0

```
5 rows × 31 columns
```

As shown, the variable names are nondescript (*V1*, *V2*, etc.). Also, the data type for the entire dataset is `float`, except `Class`, which is of type integer.

How many are fraud and how many are not fraud? Let us check:

```
class_names = {0:'Not Fraud', 1:'Fraud'}
print(dataset.Class.value_counts().rename(index = class_names))
```

```
Output
```

```
Not Fraud    284315
Fraud           492
Name: Class, dtype: int64
```

Notice the stark imbalance of the data labels. Most of the transactions are nonfraud. If we use this dataset as the base for our modeling, most models will not place enough emphasis on the fraud signals; the nonfraud data points will drown out any weight the fraud signals provide. As is, we may encounter difficulties modeling the prediction of fraud, with this imbalance leading the models to simply assume *all* transactions are nonfraud. This would be an unacceptable result. We will explore some ways of dealing with this issue in the subsequent sections.

3.2. Data visualization. Since the feature descriptions are not provided, visualizing the data will not lead to much insight. This step will be skipped in this case study.

4. Data preparation

This data is from Kaggle and is already in a cleaned format without any empty rows or columns. Data cleaning or categorization is unnecessary.

5. Evaluate models

Now we are ready to split the data and evaluate the models.

5.1. Train-test split and evaluation metrics. As described in Chapter 2, it is a good idea to partition the original dataset into training and test sets. The test set is a sample of the data that we hold back from our analysis and modeling. We use it at the end of our project to confirm the accuracy of our final model. It is the final test that gives us confidence in our estimates of accuracy on unseen data. We will use 80% of the dataset for model training and 20% for testing:

```
Y= dataset["Class"]
X = dataset.loc[:, dataset.columns != 'Class']
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Checking models. In this step, we will evaluate different machine learning models. To optimize the various hyperparameters of the models, we use ten-fold cross validation and recalculate the results ten times to account for the inherent randomness in some of the models and the CV process. All of these models, including cross validation, are described in Chapter 4.

Let us design our test harness. We will evaluate algorithms using the *accuracy metric*. This is a gross metric that will give us a quick idea of how correct a given model is. It is useful on binary classification problems.

```
# test options for classification
num_folds = 10
scoring = 'accuracy'
```

Let's create a baseline of performance for this problem and spot-check a number of different algorithms. The selected algorithms include:

Linear algorithms
: Logistic regression (LR) and linear discriminant analysis (LDA).

Nonlinear algorithms
: Classification and regression trees (CART) and *K*-nearest neighbors (KNN).

There are good reasons for selecting these models. These models are simpler and faster models with good interpretation for problems with large datasets. CART and KNN will be able to discern any nonlinear relationships between the variables. The key problem here is using an unbalanced sample. Unless we resolve that, more complex models, such as ensemble and ANNs, will have poor prediction. We will focus on addressing this later in the case study and then will evaluate the performance of these types of models.

```
# spot-check basic Classification algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
```

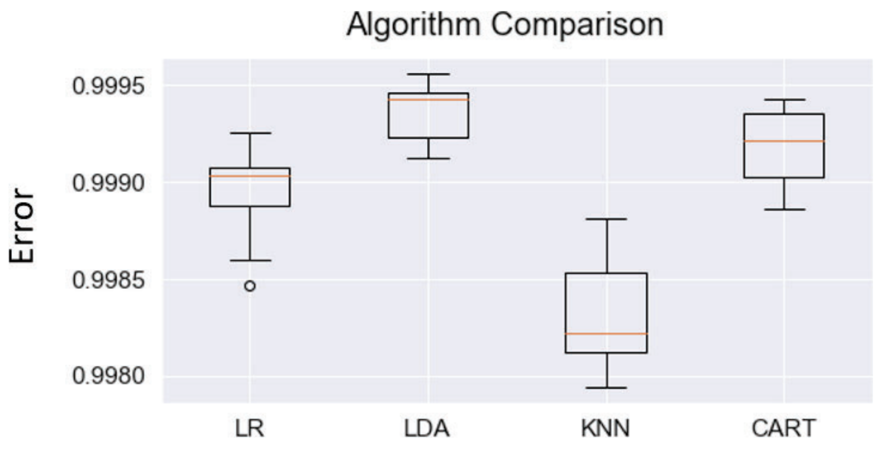
All the algorithms use default tuning parameters. We will display the mean and standard deviation of accuracy for each algorithm as we calculate and collect the results for use later.

```
results = []
names = []
for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, \
      scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

`Output`

```
LR: 0.998942 (0.000229)
LDA: 0.999364 (0.000136)
KNN: 0.998310 (0.000290)
CART: 0.999175 (0.000193)
```

```
# compare algorithms
fig = pyplot.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
pyplot.boxplot(results)
ax.set_xticklabels(names)
fig.set_size_inches(8,4)
pyplot.show()
```



The accuracy of the overall result is quite high. But let us check how well it predicts the fraud cases. Choosing one of the model CART from the results above and looking at the result on the test set:

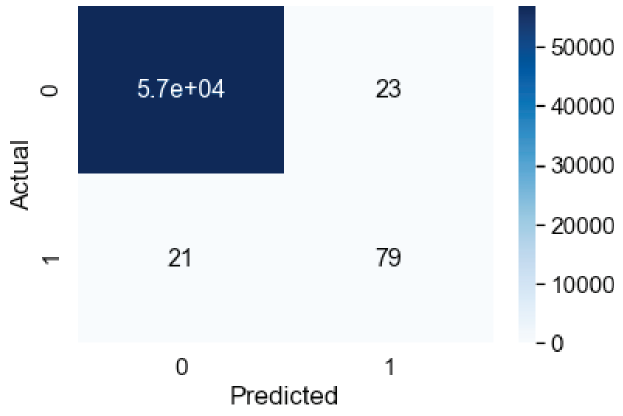
```
# prepare model
model = DecisionTreeClassifier()
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

`Output`

```
0.9992275552122467
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56862
           1       0.77      0.79      0.78       100

    accuracy                           1.00     56962
   macro avg       0.89      0.89      0.89     56962
weighted avg       1.00      1.00      1.00     56962
```

And producing the confusion matrix yields:

```
df_cm = pd.DataFrame(confusion_matrix(Y_validation, predictions), \
columns=np.unique(Y_validation), index = np.unique(Y_validation))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})
```



Overall accuracy is strong, but the confusion metrics tell a different story. Despite the high accuracy level, 21 out of 100 instances of fraud are missed and incorrectly predicted as nonfraud. The *false negative* rate is substantial.

The intention of a fraud detection model is to minimize these false negatives. To do so, the first step would be to choose the right evaluation metric.

In Chapter 4, we covered the evaluation metrics, such as accuracy, precision, and recall, for a classification-based problem. Accuracy is the number of correct predictions made as a ratio of all predictions made. Precision is the number of items correctly identified as positive out of total items identified as positive by the model. Recall is the total number of items correctly identified as positive out of total true positives.

For this type of problem, we should focus on recall, the ratio of true positives to the sum of true positives and false negatives. So if false negatives are high, then the value of recall will be low.

In the next step, we perform model tuning, select the model using the recall metric, and perform under-sampling.

6. Model tuning

The purpose of the model tuning step is to perform the grid search on the model selected in the previous step. However, since we encountered poor model performance in the previous section due to the unbalanced dataset, we will focus our attention on that. We will analyze the impact of choosing the correct evaluation metric and see the impact of using an adjusted, balanced sample.

6.1. Model tuning by choosing the correct evaluation metric. As mentioned in the preceding step, if false negatives are high, then the value of recall will be low. Models are ranked according to this metric:

```
scoring = 'recall'
```

Let us spot-check some basic classification algorithms for recall:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
```

Running cross validation:

```
results = []
names = []
for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, \
      scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

`Output`

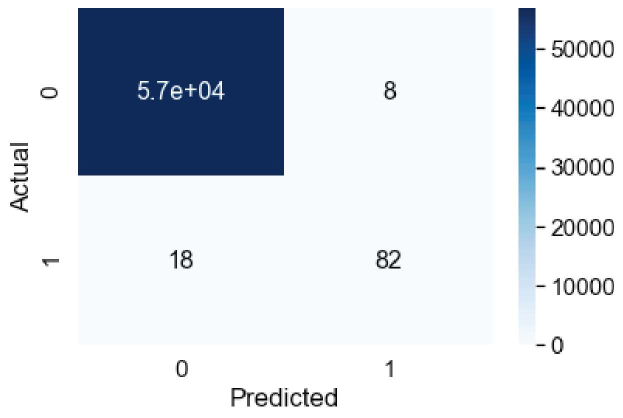
```
LR: 0.595470 (0.089743)
LDA: 0.758283 (0.045450)
KNN: 0.023882 (0.019671)
CART: 0.735192 (0.073650)
```

We see that the LDA model has the best recall of the four models. We continue by evaluating the test set using the trained LDA:

```
# prepare model
model = LinearDiscriminantAnalysis()
model.fit(X_train, Y_train)
# estimate accuracy on validation set

predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

`Output`

```
0.9995435553526912
```



LDA performs better, missing only 18 out of 100 cases of fraud. Additionally, we find fewer false positives as well. However, there is still improvement to be made.

6.2. Model tuning—balancing the sample by random under-sampling. The current data exhibits a significant class imbalance, where there are very few data points labeled "fraud." The issue of such class imbalance can result in a serious bias toward the majority class, reducing the classification performance and increasing the number of false negatives.

One of the remedies to handle such situations is to *under-sample* the data. A simple technique is to under-sample the majority class randomly and uniformly. This might lead to a loss of information, but it may yield strong results by modeling the minority class well.

Next, we will implement random under-sampling, which consists of removing data to have a more balanced dataset. This will help ensure that our models avoid overfitting.

The steps to implement random under-sampling are:

1. First, we determine the severity of the class imbalance by using `value_counts()` on the class column. We determine how many instances are considered fraud transactions (*fraud = 1*).
2. We bring the nonfraud transaction observation count to the same amount as fraud transactions. Assuming we want a 50/50 ratio, this will be equivalent to 492 cases of fraud and 492 cases of nonfraud transactions.
3. We now have a subsample of our dataframe with a 50/50 ratio with regards to our classes. We train the models on this subsample. Then we perform this iteration again to shuffle the nonfraud observations in the training sample. We keep

track of the model performance to see whether our models can maintain a certain accuracy every time we repeat this process:

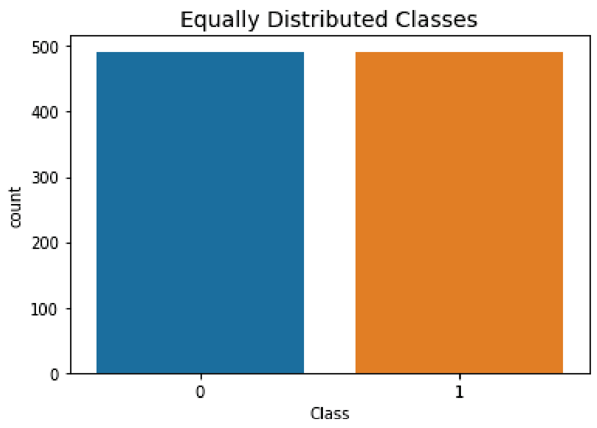
```
df = pd.concat([X_train, Y_train], axis=1)
# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
df_new = normal_distributed_df.sample(frac=1, random_state=42)
# split out validation dataset for the end
Y_train_new= df_new["Class"]
X_train_new = df_new.loc[:, dataset.columns != 'Class']
```

Let us look at the distribution of the classes in the dataset:

```
print('Distribution of the Classes in the subsample dataset')
print(df_new['Class'].value_counts()/len(df_new))
sns.countplot('Class', data=df_new)
pyplot.title('Equally Distributed Classes', fontsize=14)
pyplot.show()
```

`Output`

```
Distribution of the Classes in the subsample dataset
1    0.5
0    0.5
Name: Class, dtype: float64
```



The data is now balanced, with close to 1,000 observations. We will train all the models again, including an ANN. Now that the data is balanced, we will focus on accuracy as our main evaluation metric, since it considers both false positives and false negatives. Recall can always be produced if needed:

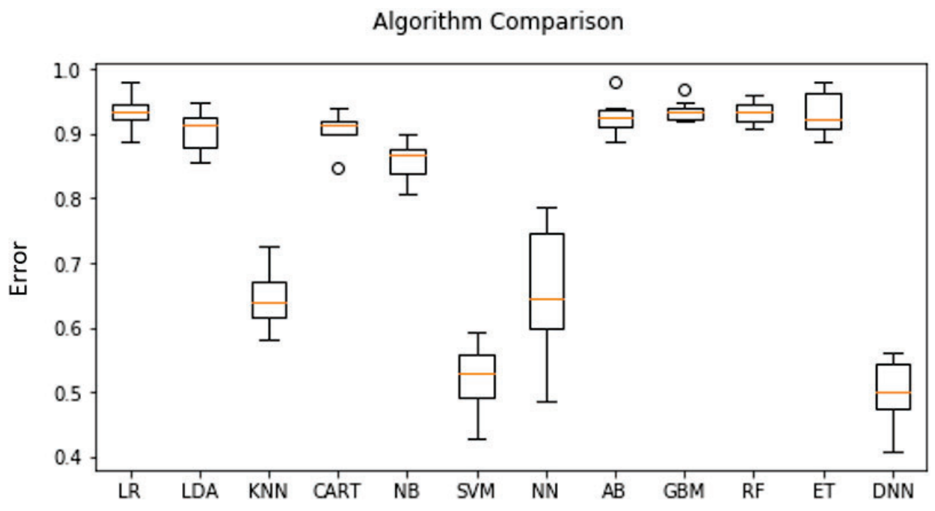
```
#setting the evaluation metric
scoring='accuracy'
# spot-check the algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
#Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier()))
models.append(('ET', ExtraTreesClassifier()))
```

The steps to define and compile an ANN-based deep learning model in Keras, along with all the terms (neurons, activation, momentum, etc.) mentioned in the following code, have been described in Chapter 3. This code can be leveraged for any deep learning–based classification model.

`Keras-based deep learning model:`

```
# Function to create model, required for KerasClassifier
def create_model(neurons=12, activation='relu', learn_rate = 0.01, momentum=0):
    # create model
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1], \
      activation=activation))
    model.add(Dense(32,activation=activation))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='binary_crossentropy', optimizer='adam', \
    metrics=['accuracy'])
    return model
models.append(('DNN', KerasClassifier(build_fn=create_model,\
epochs=50, batch_size=10, verbose=0)))
```

Running the cross validation on the new set of models results in the following:



Although a couple of models, including random forest (RF) and logistic regression (LR), perform well, GBM slightly edges out the other models. We select this for further analysis. Note that the result of the deep learning model using Keras (i.e., "DNN") is poor.

A grid search is performed for the GBM model by varying the number of estimators and maximum depth. The details of the GBM model and the parameters to tune for this model are described in Chapter 4.

```
# Grid Search: GradientBoosting Tuning
n_estimators = [20,180,1000]
max_depth= [2, 3,5]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth)
model = GradientBoostingClassifier()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
  cv=kfold)
grid_result = grid.fit(X_train_new, Y_train_new)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

`Output`

```
Best: 0.936992 using {'max_depth': 5, 'n_estimators': 1000}
```

In the next step, the final model is prepared, and the result on the test set is checked:

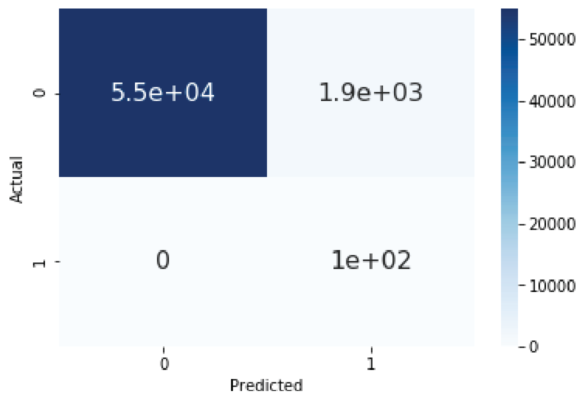
```
# prepare model
model = GradientBoostingClassifier(max_depth= 5, n_estimators = 1000)
model.fit(X_train_new, Y_train_new)
# estimate accuracy on Original validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

`Output`

```
0.9668199852533268
```

The accuracy of the model is high. Let's look at the confusion matrix:

`Output`



The results on the test set are impressive, with a high accuracy and, importantly, no false negatives. However, we see that an outcome of using our under-sampled data is a propensity for false positives—cases in which nonfraud transactions are misclassified as fraudulent. This is a trade-off the financial institution would have to consider. There is an inherent cost balance between the operational overhead, and possible customer experience impact, from processing false positives and the financial loss resulting from missing fraud cases through false negatives.

Conclusion

In this case study, we performed fraud detection on credit card transactions. We illustrated how different classification machine learning models stack up against each other and demonstrated that choosing the right metric can make an important difference in model evaluation. Under-sampling was shown to lead to a significant improvement, as all fraud cases in the test set were correctly identified after applying under-sampling. This came with a trade-off, though. The reduction in false negatives came with an increase in false positives.

Overall, by using different machine learning models, choosing the right evaluation metrics, and handling unbalanced data, we demonstrated how the implementation of a simple classification-based model can produce robust results for fraud detection.

Case Study 2: Loan Default Probability

Lending is one of the most important activities of the finance industry. Lenders provide loans to borrowers in exchange for the promise of repayment with interest. That means the lender makes a profit only if the borrower pays off the loan. Hence, the two most critical questions in the lending industry are:

1. How risky is the borrower?
2. Given the borrower's risk, should we lend to them?

Default prediction could be described as a perfect job for machine learning, as the algorithms can be trained on millions of examples of consumer data. Algorithms can perform automated tasks such as matching data records, identifying exceptions, and calculating whether an applicant qualifies for a loan. The underlying trends can be assessed with algorithms and continuously analyzed to detect trends that might influence lending and underwriting risk in the future.

The goal of this case study is to build a machine learning model to predict the probability that a loan will default.

In most real-life cases, including loan default modeling, we are unable to work with clean, complete data. Some of the potential problems we are bound to encounter are missing values, incomplete categorical data, and irrelevant features. Although data cleaning may not be mentioned often, it is critical for the success of machine learning applications. The algorithms that we use can be powerful, but without the relevant or appropriate data, the system may fail to yield ideal results. So one of the focus areas of this case study will be data preparation and cleaning. Various techniques and concepts of data processing, feature selection, and exploratory analysis are used for data cleaning and organizing the feature space.

In this case study, we will focus on:

- Data preparation, data cleaning, and handling a large number of features.
- Data discretization and handling categorical data.
- Feature selection and data transformation.

Blueprint for Creating a Machine Learning Model for Predicting Loan Default Probability

1. Problem definition

In the classification framework for this case study, the predicted variable is *charge-off*, a debt that a creditor has given up trying to collect on after a borrower has missed payments for several months. The predicted variable takes a value of 1 in case of charge-off and a value of 0 otherwise.

We will analyze data for loans from 2007 to 2017Q3 from Lending Club, available on Kaggle. Lending Club is a US peer-to-peer lending company. It operates an online lending platform that enables borrowers to obtain a loan and investors to purchase notes backed by payments made on these loans. The dataset contains more than 887,000 observations with 150 variables containing complete loan data for all loans issued over the specified time period. The features include income, age, credit scores, home ownership, borrower location, collections, and many others. We will investigate these 150 predictor variables for feature selection.

By the end of this case study, readers will be familiar with a general approach to loan default modeling, from gathering and cleaning data to building and tuning a classifier.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The standard Python packages are loaded in this step. The details have been presented in the previous case studies. Please refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. The loan data for the time period from 2007 to 2017Q3 is loaded:

```
# load dataset
dataset = pd.read_csv('LoansData.csv.gz', compression='gzip', \
low_memory=True)
```

3. Data preparation and feature selection

In the first step, let us look at the size of the data:

```
dataset.shape
```

`Output`

```
(1646801, 150)
```

Given that there are 150 features for each loan, we will first focus on limiting the feature space, followed by the exploratory analysis.

3.1. Preparing the predicted variable. Here, we look at the details of the predicted variable and prepare it. The predicted variable will be derived from the `loan_status` column. Let's check the value distributions:[2]

```
dataset['loan_status'].value_counts(dropna=False)
```

`Output`

```
Current                                                788950
Fully Paid                                             646902
Charged Off                                            168084
Late (31-120 days)                                      23763
In Grace Period                                         10474
Late (16-30 days)                                        5786
Does not meet the credit policy. Status:Fully Paid       1988
Does not meet the credit policy. Status:Charged Off       761
Default                                                    70
NaN                                                        23
Name: loan_status, dtype: int64
```

From the data definition documentation:

Fully Paid
: Loans that have been fully repaid.

Default
: Loans that have not been current for 121 days or more.

Charged Off
: Loans for which there is no longer a reasonable expectation of further payments.

A large proportion of observations show a status of `Current`, and we do not know whether those will be `Charged Off`, `Fully Paid`, or `Default` in the future. The observations for `Default` are tiny in number compared to `Fully Paid` or `Charged Off` and are not considered. The remaining categories of `loan status` are not of prime importance for this analysis. So, in order to convert this to a binary classification problem and to analyze in detail the effect of important variables on the loan status, we will consider only two major categories—Charged Off and Fully Paid:

2 The predicted variable is further used for correlation-based feature reduction.

```
dataset = dataset.loc[dataset['loan_status'].isin(['Fully Paid', 'Charged Off'])]
dataset['loan_status'].value_counts(normalize=True, dropna=False)
```

`Output`

```
Fully Paid     0.793758
Charged Off    0.206242
Name: loan_status, dtype: float64
```

About 79% of the remaining loans have been fully paid and 21% have been charged off, so we have a somewhat unbalanced classification problem, but it is not as unbalanced as the dataset of fraud detection we saw in the previous case study.

In the next step, we create a new binary column in the dataset, where we categorize Fully Paid as 0 and Charged Off as 1. This column represents the predicted variable for this classification problem. A value of 1 in this column indicates the borrower has defaulted:

```
dataset['charged_off'] = (dataset['loan_status'] == 'Charged Off').apply(np.uint8)
dataset.drop('loan_status', axis=1, inplace=True)
```

3.2. Feature selection—limit the feature space. The full dataset has 150 features for each loan, but not all features contribute to the prediction variable. Removing features of low importance can improve accuracy and reduce both model complexity and overfitting. Training time can also be reduced for very large datasets. We'll eliminate features in the following steps using three different approaches:

- Eliminating features that have more than 30% missing values.
- Eliminating features that are unintuitive based on subjective judgment.
- Eliminating features with low correlation with the predicted variable.

3.2.1. Feature elimination based on significant missing values. First, we calculate the percentage of missing data for each feature:

```
missing_fractions = dataset.isnull().mean().sort_values(ascending=False)

#Drop the missing fraction
drop_list = sorted(list(missing_fractions[missing_fractions > 0.3].index))
dataset.drop(labels=drop_list, axis=1, inplace=True)
dataset.shape
```

`Output`

```
(814986, 92)
```

This dataset has 92 columns remaining once some of the columns with a significant number of missing values are dropped.

3.2.2. Feature elimination based on intuitiveness. To filter the features further we check the description in the data dictionary and keep the features that intuitively contribute to the prediction of default. We keep features that contain the relevant credit detail of the borrower, including annual income, FICO score, and debt-to-income ratio. We also keep those features that are available to investors when considering an investment in the loan. These include features in the loan application and any features added by Lending Club when the loan listing was accepted, such as loan grade and interest rate.

The list of the features retained are shown in the following code snippet:

```
keep_list = ['charged_off','funded_amnt','addr_state', 'annual_inc', \
'application_type','dti', 'earliest_cr_line', 'emp_length',\
'emp_title', 'fico_range_high',\
'fico_range_low', 'grade', 'home_ownership', 'id', 'initial_list_status', \
'installment', 'int_rate', 'loan_amnt', 'loan_status',\
'mort_acc', 'open_acc', 'pub_rec', 'pub_rec_bankruptcies', \
'purpose', 'revol_bal', 'revol_util', \
'sub_grade', 'term', 'title', 'total_acc',\
'verification_status', 'zip_code','last_pymnt_amnt',\
'num_actv_rev_tl', 'mo_sin_rcnt_rev_tl_op',\
'mo_sin_old_rev_tl_op',"bc_util","bc_open_to_buy",\
"avg_cur_bal","acc_open_past_24mths" ]

drop_list = [col for col in dataset.columns if col not in keep_list]
dataset.drop(labels=drop_list, axis=1, inplace=True)
dataset.shape
```

`Output`

```
(814986, 39)
```

After removing the features in this step, 39 columns remain.

3.2.3. Feature elimination based on the correlation. The next step is to check the correlation with the predicted variable. Correlation gives us the interdependence between the predicted variable and the feature. We select features with a moderate-to-strong relationship with the target variable and drop those that have a correlation of less than 3% with the predicted variable:

```
correlation = dataset.corr()
correlation_chargeOff = abs(correlation['charged_off'])
drop_list_corr = sorted(list(correlation_chargeOff\
  [correlation_chargeOff < 0.03].index))
print(drop_list_corr)
```

`Output`

```
['pub_rec', 'pub_rec_bankruptcies', 'revol_bal', 'total_acc']
```

The columns with low correlation are dropped from the dataset, and we are left with only 35 columns:

```
dataset.drop(labels=drop_list_corr, axis=1, inplace=True)
```

4. Feature selection and exploratory analysis

In this step, we perform the exploratory data analysis of the feature selection. Given that many features had to be eliminated, it is preferable that we perform the exploratory data analysis after feature selection to better visualize the relevant features. We will also continue the feature elimination by visually screening and dropping those features deemed irrelevant.

4.1. Feature analysis and exploration. In the following sections, we take a deeper dive into the dataset features.

4.1.1. Analyzing the categorical features. Let us look at the some of the categorical features in the dataset.

First, let's look at the `id`, `emp_title`, `title`, and `zip_code` features:

```
dataset[['id','emp_title','title','zip_code']].describe()
```

`Output`

	id	emp_title	title	zip_code
count	814986	766415	807068	814986
unique	814986	280473	60298	925
top	14680062	Teacher	Debt consolidation	945xx
freq	1	11351	371874	9517

IDs are all unique and irrelevant for modeling. There are too many unique values for employment titles and titles. Occupation and job title may provide some information for default modeling; however, we assume much of this information is embedded in the verified income of the customer. Moreover, additional cleaning steps on these features, such as standardizing or grouping the titles, would be necessary to extract any marginal information. This work is outside the scope of this case study but could be explored in subsequent iterations of the model.

Geography could play a role in credit determination, and zip codes provide a granular view of this dimension. Again, additional work would be necessary to prepare this feature for modeling and was deemed outside the scope of this case study.

```
dataset.drop(['id','emp_title','title','zip_code'], axis=1, inplace=True)
```

Let's look at the `term` feature.

Term refers to the number of payments on the loan. Values are in months and can be either 36 or 60. The 60-month loans are more likely to charge off.

Let’s convert term to integers and group by the term for further analysis:

```
dataset['term'] = dataset['term'].apply(lambda s: np.int8(s.split()[0]))
dataset.groupby('term')['charged_off'].value_counts(normalize=True).loc[:,1]
```

`Output`

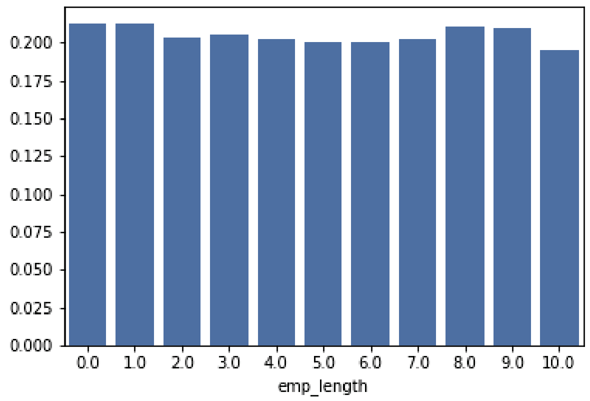
```
term
36    0.165710
60    0.333793
Name: charged_off, dtype: float64
```

Loans with five-year periods are more than twice as likely to charge-off as loans with three-year periods. This feature seems to be important for the prediction.

Let’s look at the `emp_length` feature:

```
dataset['emp_length'].replace(to_replace='10+ years', value='10 years',\
  inplace=True)

dataset['emp_length'].replace('< 1 year', '0 years', inplace=True)

def emp_length_to_int(s):
    if pd.isnull(s):
        return s
    else:
        return np.int8(s.split()[0])

dataset['emp_length'] = dataset['emp_length'].apply(emp_length_to_int)
charge_off_rates = dataset.groupby('emp_length')['charged_off'].value_counts\
  (normalize=True).loc[:,1]
sns.barplot(x=charge_off_rates.index, y=charge_off_rates.values)
```

`Output`



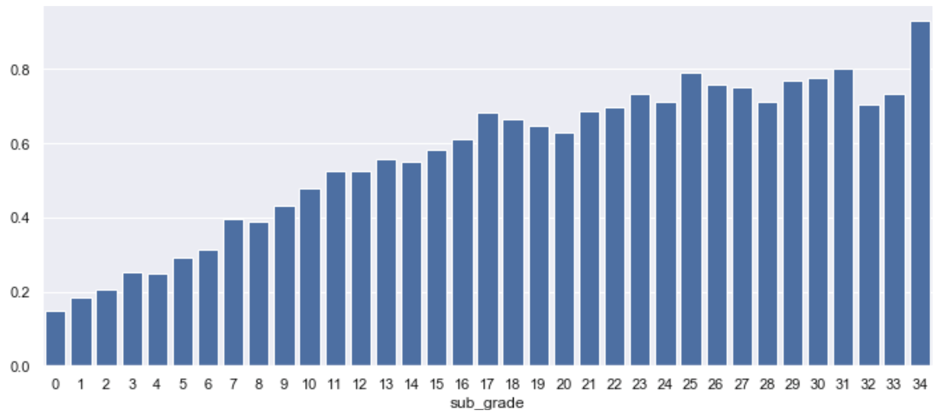
Loan status does not appear to vary much with employment length (on average); hence this feature is dropped:

```
dataset.drop(['emp_length'], axis=1, inplace=True)
```

Let's look at the `sub_grade` feature:

```
charge_off_rates = dataset.groupby('sub_grade')['charged_off'].value_counts\
(normalize=True).loc[:,1]
sns.barplot(x=charge_off_rates.index, y=charge_off_rates.values)
```

`Output`



As shown in the chart, there's a clear trend of higher probability of charge-off as the sub-grade worsens, and so it is considered to be a key feature.

4.1.2. Analyzing the continuous features. Let's look at the `annual_inc` feature:

```
dataset[['annual_inc']].describe()
```

`Output`

	annual_inc
count	8.149860e+05
mean	7.523039e+04
std	6.524373e+04
min	0.000000e+00
25%	4.500000e+04
50%	6.500000e+04
75%	9.000000e+04
max	9.550000e+06

Annual income ranges from $0 to $9,550,000, with a median of $65,000. Because of the large range of incomes, we use a log transform of the annual income variable:

```
dataset['log_annual_inc'] = dataset['annual_inc'].apply(lambda x: np.log10(x+1))
dataset.drop('annual_inc', axis=1, inplace=True)
```

Let's look at the FICO score (`fico_range_low`, `fico_range_high`) feature:

```
dataset[['fico_range_low','fico_range_high']].corr()
```

`Output`

	fico_range_low	fico_range_high
fico_range_low	1.0	1.0
fico_range_high	1.0	1.0

Given that the correlation between FICO low and high is 1, it is preferred that we keep only one feature, which we take as the average of FICO scores:

```
dataset['fico_score'] = 0.5*dataset['fico_range_low'] +\
 0.5*dataset['fico_range_high']

dataset.drop(['fico_range_high', 'fico_range_low'], axis=1, inplace=True)
```

4.2. Encoding categorical data. In order to use a feature in the classification models, we need to convert the categorical data (i.e., text features) to its numeric representation. This process is called encoding. There can be different ways of encoding. However, for this case study we will use a *label encoder*, which encodes labels with a value between 0 and *n*, where *n* is the number of distinct labels. The `LabelEncoder` function from sklearn is used in the following step, and all the categorical columns are encoded at once:

```
from sklearn.preprocessing import LabelEncoder
# Categorical boolean mask
categorical_feature_mask = dataset.dtypes==object
# filter categorical columns using mask and turn it into a list
categorical_cols = dataset.columns[categorical_feature_mask].tolist()
```

Let us look at the categorical columns:

```
categorical_cols
```

`Output`

```
['grade',
 'sub_grade',
 'home_ownership',
 'verification_status',
 'purpose',
 'addr_state',
 'initial_list_status',
 'application_type']
```

4.3. Sampling data. Given that the loan data is skewed, it is sampled to have an equal number of charge-off and no charge-off observations. Sampling leads to a more balanced dataset and avoids overfitting:[3]

```
loanstatus_0 = dataset[dataset["charged_off"]==0]
loanstatus_1 = dataset[dataset["charged_off"]==1]
subset_of_loanstatus_0 = loanstatus_0.sample(n=5500)
subset_of_loanstatus_1 = loanstatus_1.sample(n=5500)
dataset = pd.concat([subset_of_loanstatus_1, subset_of_loanstatus_0])
dataset = dataset.sample(frac=1).reset_index(drop=True)
print("Current shape of dataset :",dataset.shape)
```

Although sampling may have its advantages, there might be some disadvantages as well. Sampling may exclude some data that might not be homogeneous to the data that is taken. This affects the level of accuracy in the results. Also, selection of the proper size of samples is a difficult job. Hence, sampling should be performed with caution and should generally be avoided in the case of a relatively balanced dataset.

5. Evaluate algorithms and models

5.1. Train-test split. Splitting out the validation dataset for the model evaluation is the next step:

```
Y= dataset["charged_off"]
X = dataset.loc[:, dataset.columns != 'charged_off']
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation = \
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Test options and evaluation metrics. In this step, the test options and evaluation metrics are selected. The `roc_auc` evaluation metric is selected for this classification. The details of this metric were provided in Chapter 4. This metric represents a model's ability to discriminate between positive and negative classes. An `roc_auc` of 1.0 represents a model that made all predictions perfectly, and a value of 0.5 represents a model that is as good as random.

```
num_folds = 10
scoring = 'roc_auc'
```

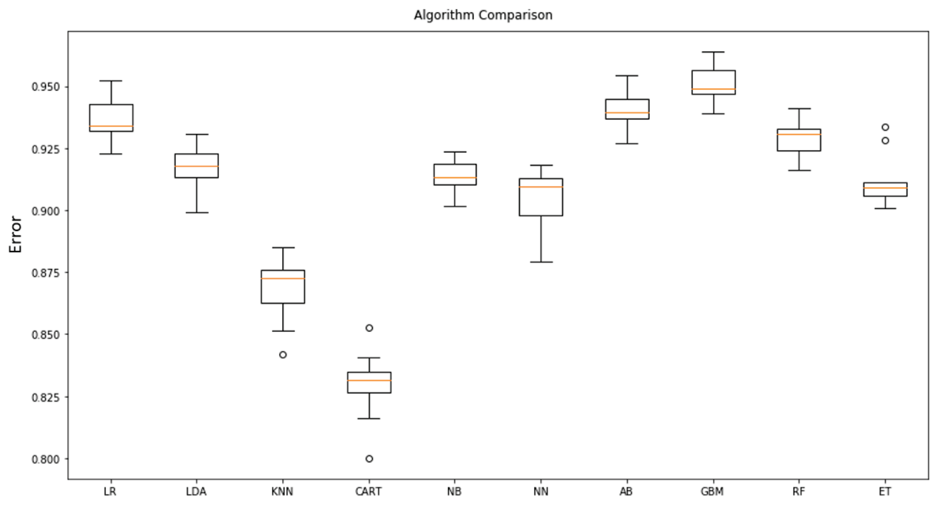
The model cannot afford to have a high amount of false negatives as that leads to a negative impact on the investors and the credibility of the company. So we can use recall as we did in the fraud detection use case.

3 Sampling is covered in detail in "Case Study 1: Fraud Detection" on page 153.

5.3. Compare models and algorithms. Let us spot-check the classification algorithms. We include ANN and ensemble models in the list of models to be checked:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
# Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier()))
models.append(('ET', ExtraTreesClassifier()))
```

After performing the *k*-fold cross validation on the models shown above, the overall performance is as follows:



The gradient boosting method (GBM) model performs best, and we select it for grid search in the next step. The details of GBM along with the model parameters are described in Chapter 4.

6. Model tuning and grid search

We tune the number of estimator and maximum depth hyperparameters, which were discussed in Chapter 4:

```
# Grid Search: GradientBoosting Tuning
n_estimators = [20,180]
max_depth= [3,5]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth)
model = GradientBoostingClassifier()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
  cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

`Output`

```
Best: 0.952950 using {'max_depth': 5, 'n_estimators': 180}
```

A GBM model with `max_depth` of 5 and number of estimators of 150 results in the best model.

7. Finalize the model

Now, we perform the final steps for selecting a model.

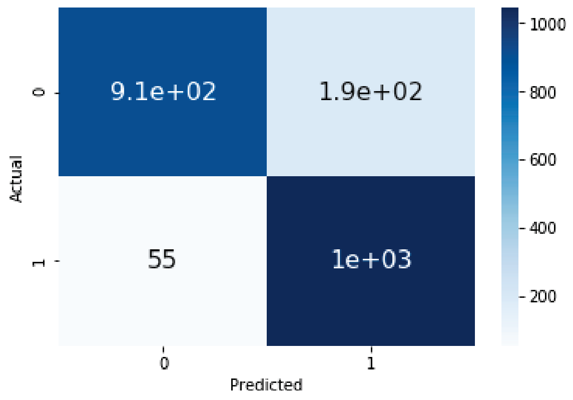
7.1. Results on the test dataset. Let us prepare the GBM model with the parameters found during the grid search step and check the results on the test dataset:

```
model = GradientBoostingClassifier(max_depth= 5, n_estimators= 180)
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

`Output`

```
0.889090909090909
```

The accuracy of the model is a reasonable 89% on the test set. Let us examine the confusion matrix:

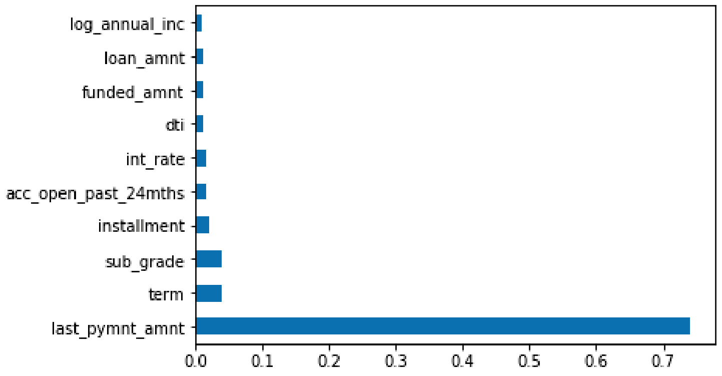


Looking at the confusion matrix and the overall result of the test set, both the rate of false positives and the rate of false negatives are lower; the overall model performance looks good and is in line with the training set results.

7.2. Variable intuition/feature importance. In this step, we compute and display the variable importance of our trained model:

```
print(model.feature_importances_) #use inbuilt class feature_importances
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
#plot graph of feature importances for better visualization
feat_importances.nlargest(10).plot(kind='barh')
pyplot.show()
```

```
Output
```



The results of the model importance are intuitive. The last payment amount seems to be the most important feature, followed by loan term and sub-grade.

Conclusion

In this case study, we introduced the classification-based tree algorithm applied to loan default prediction. We showed that data preparation is one of the most important steps. We addressed this by performing feature elimination using different techniques, such as feature intuition, correlation analysis, visualization, and data quality checks of the features. We illustrated that there can be different ways of handling and analyzing the categorical data and converting categorical data into a usable format for the models.

We emphasized that performing data processing and establishing an understanding of variable importance is key in the model development process. A focus on these steps led to the implementation of a simple classification-based model that produced robust results for default prediction.

Case Study 3: Bitcoin Trading Strategy

First released as open source in 2009 by the pseudonymous Satoshi Nakamoto, bitcoin is the longest-running and most well-known cryptocurrency.

A major drawback of cryptocurrency trading is the volatility of the market. Since cryptocurrency markets trade 24/7, tracking cryptocurrency positions against quickly changing market dynamics can rapidly become an impossible task to manage. This is where automated trading algorithms and trading bots can assist.

Various machine learning algorithms can be used to generate trading signals in an attempt to predict the market's movement. One could use machine learning algorithms to classify the next day's movement into three categories: market will rise (take a long position), market will fall (take a short position), or market will move sideways (take no position). Since we know the market direction, we can decide the optimum entry and exit points.

Machine learning has one key aspect called *feature engineering*. It means that we can create new, intuitive features and feed them to a machine learning algorithm in order to achieve better results. We can introduce different technical indicators as features to help predict future prices of an asset. These technical indicators are derived from market variables such as price or volume and have additional information or signals embedded in them. There are many different categories of technical indicators, including trend, volume, volatility, and momentum indicators.

In this case study, we will use various classification-based models to predict whether the current position signal is buy or sell. We will create additional trend and momentum indicators from market prices to leverage as additional features in the prediction.

In this case study, we will focus on:

- Building a trading strategy using classification (classification of long/short signals).
- Feature engineering and constructing technical indicators of trend, momentum, and mean reversion.
- Building a framework for backtesting results of a trading strategy.
- Choosing the right evaluation metric to assess a trading strategy.

Blueprint for Using Classification-Based Models to Predict Whether to Buy or Sell in the Bitcoin Market

1. Problem definition

The problem of predicting a buy or sell signal for a trading strategy is defined in the classification framework, where the predicted variable has a value of 1 for buy and 0 for sell. This signal is decided through the comparison of the short-term and long-term price trends.

The data used is from one of the largest bitcoin exchanges in terms of average daily volume, Bitstamp. The data covers prices from January 2012 to May 2017. Different trend and momentum indicators are created from the data and are added as features to enhance the performance of the prediction model.

By the end of this case study, readers will be familiar with a general approach to building a trading strategy, from cleaning data and feature engineering to model tuning and developing a backtesting framework.

2. Getting started—loading the data and Python packages

Let's load the packages and the data.

2.1. Loading the Python packages. The standard Python packages are loaded in this step. The details have been presented in the previous case studies. Refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. The bitcoin data fetched from the Bitstamp website is loaded in this step:

```
# load dataset
dataset = pd.read_csv('BitstampData.csv')
```

3. Exploratory data analysis

In this step, we will take a detailed look at this data.

3.1. Descriptive statistics. First, let us look at the shape of the data:

```
dataset.shape
```

Output

```
(2841377, 8)
```

```
# peek at data
set_option('display.width', 100)
dataset.tail(2)
```

Output

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
2841372	1496188560	2190.49	2190.49	2181.37	2181.37	1.700166	3723.784755	2190.247337
2841373	1496188620	2190.50	2197.52	2186.17	2195.63	6.561029	14402.811961	2195.206304

The dataset has minute-by-minute data of OHLC (Open, High, Low, Close) and traded volume of bitcoin. The dataset is large, with approximately 2.8 million total observations.

4. Data preparation

In this part, we will clean the data to prepare for modeling.

4.1. Data cleaning. We clean the data by filling the NaNs with the last available values:

```
dataset[dataset.columns.values] = dataset[dataset.columns.values].ffill()
```

The `Timestamp` column is not useful for modeling and is dropped from the dataset:

```
dataset=dataset.drop(columns=['Timestamp'])
```

4.2. Preparing the data for classification. As a first step, we will create the target variable for our model. This is the column that will indicate whether the trading signal is buy or sell. We define the short-term price as the 10-day rolling average and the long-term price as the 60-day rolling average. We attach a label of `1` (`0`) if the short-term price is higher (lower) than the long-term price:

```
# Create short simple moving average over the short window
dataset['short_mavg'] = dataset['Close'].rolling(window=10, min_periods=1,\
center=False).mean()
```

```
# Create long simple moving average over the long window
dataset['long_mavg'] = dataset['Close'].rolling(window=60, min_periods=1,\
center=False).mean()

# Create signals
dataset['signal'] = np.where(dataset['short_mavg'] >
dataset['long_mavg'], 1.0, 0.0)
```

4.3. Feature engineering. We begin feature engineering by analyzing the features we expect may influence the performance of the prediction model. Based on a conceptual understanding of key factors that drive investment strategies, the task at hand is to identify and construct new features that may capture the risks or characteristics embodied by these return drivers. For this case study, we will explore the efficacy of specific momentum technical indicators.

The current data of the bitcoin consists of date, open, high, low, close, and volume. Using this data, we calculate the following momentum indicators:

Moving average
: A moving average provides an indication of a price trend by cutting down the amount of noise in the series.

Stochastic oscillator %K
: A stochastic oscillator is a momentum indicator that compares the closing price of a security to a range of its previous prices over a certain period of time. *%K* and *%D* are slow and fast indicators. The fast indicator is more sensitive than the slow indicator to changes in the price of the underlying security and will likely result in many transaction signals.

Relative strength index (RSI)
: This is a momentum indicator that measures the magnitude of recent price changes to evaluate overbought or oversold conditions in the price of a stock or other asset. The RSI ranges from 0 to 100. An asset is deemed to be overbought once the RSI approaches 70, meaning that the asset may be getting overvalued and is a good candidate for a pullback. Likewise, if the RSI approaches 30, it is an indication that the asset may be getting oversold and is therefore likely to become undervalued.

Rate of change (ROC)
: This is a momentum oscillator that measures the percentage change between the current price and the *n* period past prices. Assets with higher ROC values are considered more likely to be overbought; those with lower ROC, more likely to be oversold.

Momentum (MOM)

This is the rate of acceleration of a security's price or volume—that is, the speed at which the price is changing.

The following steps show how to generate some useful features for prediction. The features for trend and momentum can be leveraged for other trading strategy models:

```
#calculation of exponential moving average
def EMA(df, n):
    EMA = pd.Series(df['Close'].ewm(span=n, min_periods=n).mean(), name='EMA_'\
     + str(n))
    return EMA
dataset['EMA10'] = EMA(dataset, 10)
dataset['EMA30'] = EMA(dataset, 30)
dataset['EMA200'] = EMA(dataset, 200)
dataset.head()

#calculation of rate of change
def ROC(df, n):
    M = df.diff(n - 1)
    N = df.shift(n - 1)
    ROC = pd.Series(((M / N) * 100), name = 'ROC_' + str(n))
    return ROC
dataset['ROC10'] = ROC(dataset['Close'], 10)
dataset['ROC30'] = ROC(dataset['Close'], 30)

#calculation of price momentum
def MOM(df, n):
    MOM = pd.Series(df.diff(n), name='Momentum_' + str(n))
    return MOM
dataset['MOM10'] = MOM(dataset['Close'], 10)
dataset['MOM30'] = MOM(dataset['Close'], 30)

#calculation of relative strength index
def RSI(series, period):
 delta = series.diff().dropna()
 u = delta * 0
 d = u.copy()
 u[delta > 0] = delta[delta > 0]
 d[delta < 0] = -delta[delta < 0]
 u[u.index[period-1]] = np.mean( u[:period] ) #first value is sum of avg gains
 u = u.drop(u.index[:(period-1)])
 d[d.index[period-1]] = np.mean( d[:period] ) #first value is sum of avg losses
 d = d.drop(d.index[:(period-1)])
 rs = u.ewm(com=period-1, adjust=False).mean() / \
 d.ewm(com=period-1, adjust=False).mean()
 return 100 - 100 / (1 + rs)
dataset['RSI10'] = RSI(dataset['Close'], 10)
dataset['RSI30'] = RSI(dataset['Close'], 30)
dataset['RSI200'] = RSI(dataset['Close'], 200)

#calculation of stochastic osillator.
```

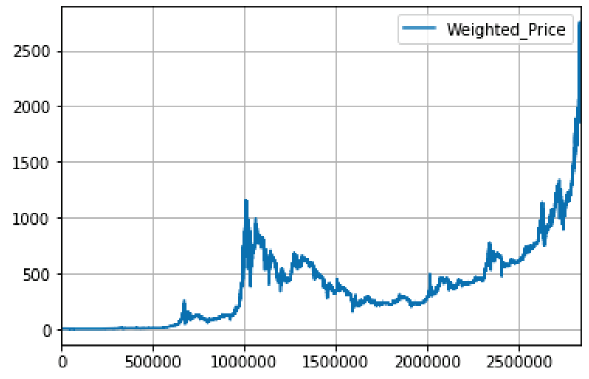
```
def STOK(close, low, high, n):
 STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
 low.rolling(n).min())) * 100
 return STOK

def STOD(close, low, high, n):
 STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
 low.rolling(n).min())) * 100
 STOD = STOK.rolling(3).mean()
 return STOD

dataset['%K10'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%D10'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%K30'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%D30'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%K200'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 200)
dataset['%D200'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 200)

#calculation of moving average
def MA(df, n):
    MA = pd.Series(df['Close'].rolling(n, min_periods=n).mean(), name='MA_'\
     + str(n))
    return MA
dataset['MA21'] = MA(dataset, 10)
dataset['MA63'] = MA(dataset, 30)
dataset['MA252'] = MA(dataset, 200)
```

With our features completed, we'll prepare them for use.

4.4. Data visualization. In this step, we visualize different properties of the features and the predicted variable:

```
dataset[['Weighted_Price']].plot(grid=True)
plt.show()
```

`Output`

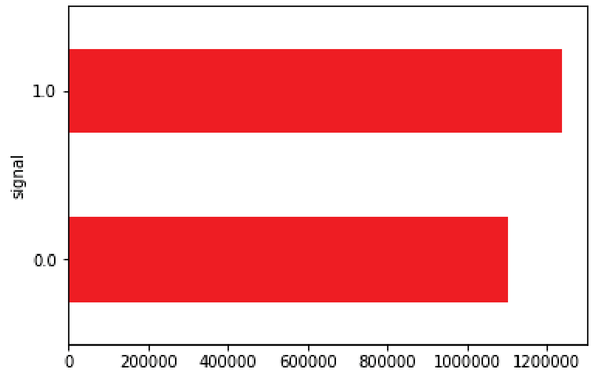


The chart illustrates a sharp rise in the price of bitcoin, increasing from close to $0 to around $2,500 in 2017. Also, high price volatility is readily visible.

Let us look at the distribution of the predicted variable:

```
fig = plt.figure()
plot = dataset.groupby(['signal']).size().plot(kind='barh', color='red')
plt.show()
```

`Output`



The predicted variable is 1 more than 52% of the time, meaning there are more buy signals than sell signals. The predicted variable is relatively balanced, especially as compared to the fraud dataset we saw in the first case study.

5. Evaluate algorithms and models

In this step, we will evaluate different algorithms.

5.1. Train-test split. We first split the dataset into training (80%) and test (20%) sets. For this case study we use 100,000 observations for a faster calculation. The next steps would be same in the event we want to use the entire dataset:

```
# split out validation dataset for the end
subset_dataset= dataset.iloc[-100000:]
Y= subset_dataset["signal"]
X = subset_dataset.loc[:, dataset.columns != 'signal']
validation_size = 0.2
seed = 1
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=1)
```

5.2. Test options and evaluation metrics. Accuracy can be used as the evaluation metric since there is not a significant class imbalance in the data:

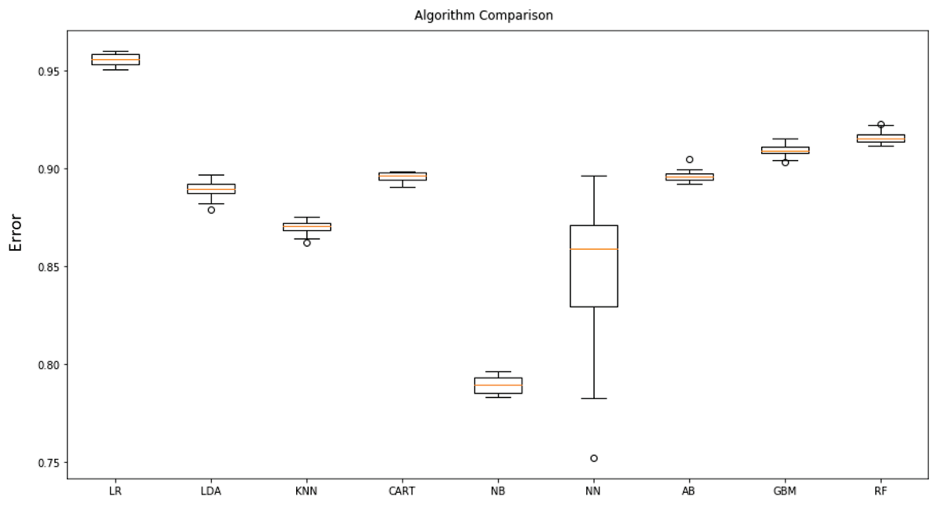
```
# test options for classification
num_folds = 10
scoring = 'accuracy'
```

5.3. Compare models and algorithms. In order to know which algorithm is best for our strategy, we evaluate the linear, nonlinear, and ensemble models.

5.3.1. Models. Checking the classification algorithms:

```
models = []
models.append(('LR', LogisticRegression(n_jobs=-1)))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
#Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier(n_jobs=-1)))
```

After performing the *k*-fold cross validation, the comparison of the models is as follows:



Although some of the models show promising results, we prefer an ensemble model given the huge size of the dataset, the large number of features, and an expected non-linear relationship between the predicted variable and the features. Random forest has the best performance among the ensemble models.

6. Model tuning and grid search

A grid search is performed for the random forest model by varying the number of estimators and maximum depth. The details of the random forest model and the parameters to tune are discussed in Chapter 4:

```
n_estimators = [20,80]
max_depth= [5,10]
criterion = ["gini","entropy"]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth, \
  criterion = criterion )
model = RandomForestClassifier(n_jobs=-1)
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, \
  scoring=scoring, cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_,\
  grid_result.best_params_))
```

`Output`

```
Best: 0.903438 using {'criterion': 'gini', 'max_depth': 10, 'n_estimators': 80}
```

7. Finalize the model

Let us finalize the model with the best parameters found during the tuning step and perform the variable intuition.

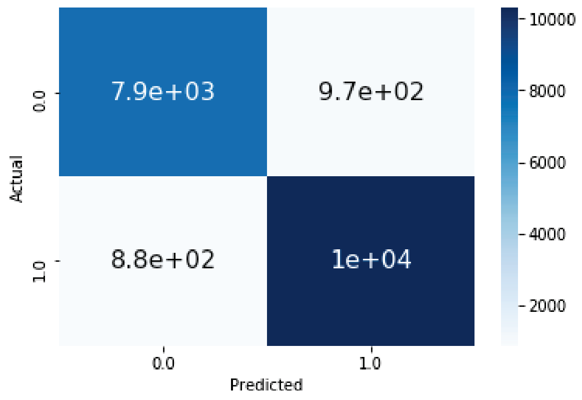
7.1. Results on the test dataset. In this step, we evaluate the selected model on the test set:

```
# prepare model
model = RandomForestClassifier(criterion='gini', n_estimators=80,max_depth=10)

#model = LogisticRegression()
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

`Output`

```
0.9075
```

The selected model performs quite well, with an accuracy of 90.75%. Let us look at the confusion matrix:

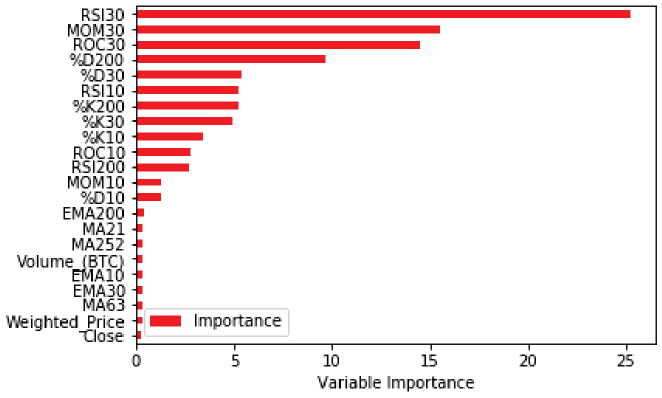


The overall model performance is reasonable and is in line with the training set results.

7.2. Variable intuition/feature importance. Let us look into the feature importance of the model:

```
Importance = pd.DataFrame({'Importance':model.feature_importances_*100},\
 index=X.columns)
Importance.sort_values('Importance', axis=0, ascending=True).plot(kind='barh', \
color='r' )
plt.xlabel('Variable Importance')
```

```
Output
```



The result of the variable importance looks intuitive, and the momentum indicators of RSI and MOM over the last 30 days seem to be the two most important features.

The feature importance chart corroborates the fact that introducing new features leads to an improvement in the model performance.

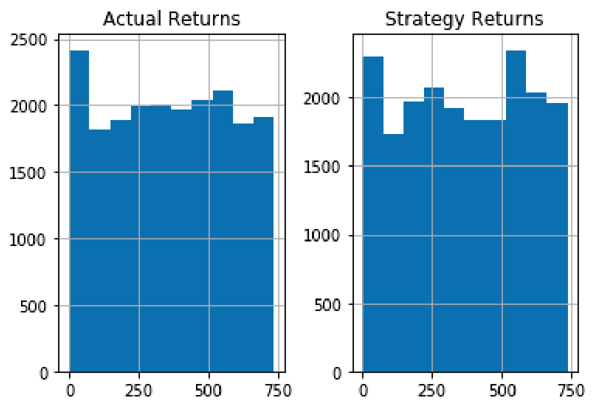
7.3. Backtesting results. In this additional step, we perform a backtest on the model we've developed. We create a column for strategy returns by multiplying the daily returns by the position that was held at the close of business the previous day and compare it against the actual returns.

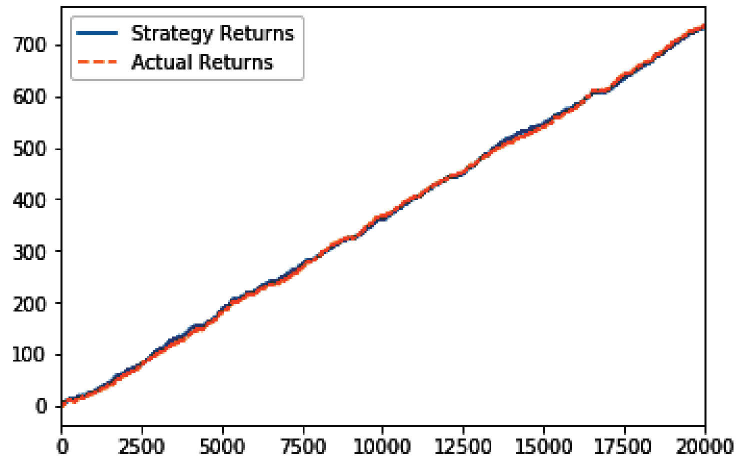
Backtesting Trading Strategies

A backtesting approach similar to the one presented in this case study can be used to quickly backtest any trading strategy.

```
backtestdata = pd.DataFrame(index=X_validation.index)
backtestdata['signal_pred'] = predictions
backtestdata['signal_actual'] = Y_validation
backtestdata['Market Returns'] = X_validation['Close'].pct_change()
backtestdata['Actual Returns'] = backtestdata['Market Returns'] *\
backtestdata['signal_actual'].shift(1)
backtestdata['Strategy Returns'] = backtestdata['Market Returns'] * \
backtestdata['signal_pred'].shift(1)
backtestdata=backtestdata.reset_index()
backtestdata.head()
backtestdata[['Strategy Returns','Actual Returns']].cumsum().hist()
backtestdata[['Strategy Returns','Actual Returns']].cumsum().plot()
```

```
Output
```





Looking at the backtesting results, we do not deviate significantly from the actual market return. Indeed, the achieved momentum trading strategy made us better at predicting the price direction to buy or sell in order to make profits. However, as our accuracy is not 100% (but more than 96%), we made relatively few losses compared to the actual returns.

Conclusion

This case study demonstrated that framing the problem is a key step when tackling a finance problem with machine learning. In doing so, it was detemined that transforming the labels according to an investment objective and performing feature engineering were required for this trading strategy. We demonstrated the efficiency of using intuitive features related to the trend and momentum of the price movement. This helped increase the predictive power of the model. Finally, we introduced a backtesting framework, which allowed us to simulate a trading strategy using historical data. This enabled us to generate results and analyze risk and profitability before risking any actual capital.

Chapter Summary

In "Case Study 1: Fraud Detection" on page 153, we explored the issue of an unbalanced dataset and the importance of having the right evaluation metric. In "Case Study 2: Loan Default Probability" on page 166, various techniques and concepts of data processing, feature selection, and exploratory analysis were covered. In "Case Study 3: Bitcoin Trading Strategy" on page 179, we looked at ways to create technical indicators as features in order to use them for model enhancement. We also prepared a backtesting framework for a trading strategy.

Overall, the concepts in Python, machine learning, and finance presented in this chapter can used as a blueprint for any other classification-based problem in finance.

Exercises

- Predict whether a stock price will go up or down using the features related to the stock or macroeconomic variables (use the ideas from the bitcoin-based case study presented in this chapter).
- Create a model to detect money laundering using the features of a transaction. A sample dataset for this exercise can be obtained from Kaggle.
- Perform a credit rating analysis of corporations using the features related to creditworthiness.