C# Machine Learning Projects

Nine real-world projects to build robust and high-performing machine learning models with C#

www.packt.com

By Yoon Hyup Hwang

C# Machine Learning Projects

Nine real-world projects to build robust and high-performing machine learning models with C#

Yoon Hyup Hwang

Packt

BIRMINGHAM - MUMBAI

C# Machine Learning Projects

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Tushar Gupta
Content Development Editor: Aaryaman Singh
Technical Editor: Sayli Nikalje
Copy Editors: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Tania Dutta
Production Coordinator: Arvindkumar Gupta

First published: June 2018

Production reference: 1130618

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-640-2

www.packtpub.com

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Yoon Hyup Hwang is a seasoned data scientist with several years of experience in marketing and finance industries with expertise in predictive modeling, machine learning, statistical analysis, and data engineering. He holds M.S.E. in Computer and Information Technology from the University of Pennsylvania, and a B.A. in Economics from the University of Chicago.

In his free time, he enjoys training different martial arts (Krav Maga, BJJ, and Muay Thai), snowboarding, and roasting coffee.

To my wife, Sunyoung, thank you for your endless support and love. Sacrificing our weekends and family time, you played a critical role in publishing this book. To my family, thank you for your consistent support and trust in me. To my publishing team at Packt, I would like to thank everyone, especially Aaryaman, Sayli, and Tushar. Also, I would like to thank everyone reading this book, and I hope you enjoy it. Cheers!

About the reviewer

Dirk Strauss is a software developer and Microsoft .NET MVP from South Africa with over 13 years of programming experience. He has extensive experience in SYSPRO customization, with C# and web development being his main focus. He currently works for Embrace as a full-stack developer.

He authored *C# Programming Cookbook* in 2016 and *C# 7 and .NET Core 2.0 Blueprints* in 2017 (published by Packt). He has also written for Syncfusion, contributing to the Succinctly series of eBooks. He blogs whenever he gets a chance. You can follow him on Twitter at `@DirkStrauss`.

Prakash Tripathi, is a technical lead in an MNC by profession, and an author and speaker by passion. He has extensive experience in the design, development, maintenance, and support of enterprise applications, mainly using Microsoft technologies and platforms.

Active in technical communities, he has been awarded **Most Valuable Professional** (**MVP**) by Microsoft for 2016-17 and 2017-18, and by c-sharpcorner (a leading developers portal) three times. He holds a master's in computer applications from MANIT-Bhopal, India.

I would like to thank my parents for bringing me into the world and teaching me life skills, and my wife, Aradhana, for her continuous support and motivation throughout the journey of this book. I cannot forget to thank Microsoft and the c-sharpcorner technical communities. They've provided a platform to demonstrate and excel in this fast-growing and ever-changing field.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Summary

Preface

In the era of data, it is hard to overlook the importance of **machine learning** (**ML**) and data science. ML has been used and adopted widely across many industries, and its adoption rate is growing faster than ever. Not only big technology companies, such as Google, Microsoft, and Apple, but also many non-technology companies such as Bloomberg and Goldman Sachs, invest heavily in ML. From searching for what to eat for dinner tonight on search engines to getting approvals for new credit cards, the applications of ML are everywhere in our daily lives. As a fellow data scientist and ML practitioner, I cannot emphasize enough the importance of ML in the current era of data, especially of big data.

If you are looking for resources to learn applied ML, you have come to the right place. For many aspiring data scientists and ML practitioners, the amount of resources for applied ML in C# is lacking in the ML books out there. You can easily find books that have detailed explanations on the theory behind ML. You can also easily find books that touch the practical aspects of ML in different programming languages, such as Python. However, as you might have noticed, there are not many books that cover how to build practical ML models and applications using C#.

In this book, we are going to focus on the practical side of ML and dive right into building ML models and applications for various real-world projects that are actively being researched and built in many different industries. By going through real-life examples of ML with real-world datasets, you will understand how other data scientists and ML practitioners actually build ML models and applications for their production systems. This book is unique in the sense that each chapter is an individual ML project that has a real-world business use case.

In this book, C# is the choice of language for the ML projects that we are going to work on. You might ask, *Why C#?* The answer is actually quite straightforward. As you might know already, C# is one of the most popular and widely used languages in the industry. Especially among finance companies, C# is one of the very few programming languages that is commonly accepted and used for their production applications.

Personally, I would have benefited from books like this one when I was starting my career in data science. At that time, there was a discrepancy between what was taught in schools and what really works in real life (and how). In this book, I would like to share the knowledge and experience that I had to learn the hard way. In this book, we are going to discuss things that are not frequently talked about, such as how ML projects typically start, how ML models are built and tested in different industries, how ML applications are then deployed on production systems, and how those ML models running in production systems are monitored and evaluated. We are going to work together throughout this book to help you get ready for any ML projects that you may come across in the future. By the end of this book, you will be able to build robust and well-performing ML models and applications using C#.

Who this book is for

This book is for those who know how to write code using C# and have a basic understanding of ML. Even if you do not have in-depth knowledge of the theories behind ML algorithms, don't worry! It is okay. This book will help you understand how you can use different learning algorithms for different use cases. If you have studied ML, maybe at school, on online courses, or at data science boot camps, then this book will be great for you. This book will show you how to actually apply the ML theories and concepts you have learned by going through nine real-world ML projects with real-world datasets. If you are already an ML practitioner, you will still greatly benefit from this book! By going through various real-world examples of applied ML, this book will help you expand your knowledge and experience of applying ML to various other business cases for many different industries.

This book is really for anyone with a passion for applied ML. If you want to be able to build ML models and applications that can be used in production systems from day 1, then this book is for you!

What this book covers

`Chapter 1`, *Basics of Machine Learning Modeling*, discusses some of the real-world examples of ML applications that we can easily find around ourselves. It also covers the essential steps in building ML models and how to set up a C# development environment for the upcoming real-world ML projects.

`Chapter 2`, *Spam Email Filtering*, covers feature engineering techniques for text datasets and how to build classification models using logistic regression and Naive Bayes learning algorithms. This chapter also discusses some of the basic model validation methods for classification models.

`Chapter 3`, *Twitter Sentiment Analysis*, describes some of the commonly used **natural language processing** (**NLP**) techniques for feature engineering and how to build multi-class classification models. This chapter also covers how to build Naive Bayes and random forest classifiers in C#, along with more advanced model evaluation metrics for classification models.

`Chapter 4`, *Foreign Exchange Rate Forecast*, dives into regression problems, where the target variables are continuous variables. This chapter discusses some of the frequently used technical indicators in the foreign exchange market and how to use them as features for building foreign exchange rate forecasting models. It also covers how to build linear regression and **Support Vector Machines** (**SVMs**) for foreign exchange rate forecasting.

`Chapter 5`, *Fair Value of House and Property*, covers a regression problem with mixed types of features in the dataset. This chapter discusses using different kernel methods for SVM models. It also describes some of the fundamental model evaluation metrics for regression models and how to use them to compare the models built.

`Chapter 6`, *Customer Segmentation*, describes an unsupervised learning problem, where there is no labeled target variable. It discusses how to use a k-means clustering algorithm to draw insights into customer behaviors from an e-commerce dataset. This chapter also discusses a metric that can be used to evaluate how well each cluster or segment is formed.

Chapter 7, *Music Genre Recommendation*, introduces a ranking problem, where the number of outputs of a ML model is more than one. This chapter covers how to build ML models for recommending music genres and how to evaluate the recommendation results from these models.

Chapter 8, *Handwritten Digit Recognition*, discusses an image recognition problem, where the goal is to build ML models to recognize handwritten digits. It covers one of the dimensionality reduction techniques and how it can be used for image dataset. This chapter introduces neural network models for image recognition.

Chapter 9, *Cyber Attack Detection*, dives into an anomaly detection problem. in this chapter we will try to build ML models to detect cyber attacks. It covers how to use the dimensionality reduction technique called **Principal Component Analysis** (**PCA**) to build an anomaly detection model that can identify cyber attacks.

Chapter 10, *Credit Card Fraud Detection*, continues with the anomaly detection problem. This chapter discusses how to build ML models to detect credit card frauds. It introduces a new ML algorithm, one-class SVM, for an anomaly detection model.

Chapter 11, *What's Next?*, is the closing chapter of this book. It reviews everything that was discussed throughout the book. It then covers some of the frequently appearing challenges in real-life ML projects. This chapter also discusses a few commonly used technologies and tools for data science tasks.

To get the most out of this book

To get the most out of this book, I recommend that you thoroughly follow each of the steps laid out in each chapter. Following through the code samples and running them on your own will help you understand better and get more comfortable with building ML models faster. I also recommend that you be adventurous and mix up the techniques and learning algorithms discussed in different chapters. When you are done with this book, it will be even better if you go through the projects yourself again from the beginning and start building your own versions of ML models for the individual projects.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/CSharp-Machine-Learning-Projects. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/CSharpMachineLearningProjects_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Open your Visual Studio and create a new `Console Application` under the Visual C# category. Use the preceding command to install the `Deedle` library through `NuGet` and add references to your project."

A block of code is set as follows:

```
var barChart = DataBarBox.Show(
new string[] { "Ham", "Spam" },
new double[] {
hamEmailCount,
spamEmailCount
}
);
barChart.SetTitle("Ham vs. Spam in Sample Set");
```

Any command-line input or output is written as follows: **PM> Install-Package Deedle**

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Open the package manager (Tools | NuGet Package Manager | Package Manager Console) and install `Deedle` using the following command."

ℹ *Warnings or important notes appear like this.*

TIP *Tips and tricks appear like this.*

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Basics of Machine Learning Modeling

It can be difficult to see how **machine learning** (**ML**) affects the daily lives of ordinary people. In fact, ML is everywhere! In the process of searching for a restaurant for dinner, you almost certainly used ML. In the search for a dress to wear for a dinner party, you would have used ML. On your way to your dinner appointment, you probably used ML as well if you used one of the ride-sharing apps. ML has become so widely used that it has become an essential part of our lives, although it is usually unnoticeable. With ever-growing data and its accessibility, the applications and needs for ML are rapidly rising across various industries. However, the pace of the growth in trained data scientists has yet to meet the pace of growth ML needs in businesses, despite abundant resources and software libraries that make building ML models easier, due to the fact that it takes time and experience for a data scientist and ML engineer to master such skill sets. This book will prepare such individuals with real-world projects based on real-world datasets.

In this chapter, we will learn about some of the real-life examples and applications of ML, the essential steps in building ML models, and how to set up our C# environment for ML. After this brief introductory chapter, we will dive immediately into building classification ML models using text datasets in `Chapter 2`, *Spam Email Filtering*, and `Chapter 3`, *Twitter Sentiment Analysis*. Then, we will use financial and real estate property data to build regression models in `Chapter 4`, *Foreign Exchange Rate Forecast*, and `Chapter 5`, *Fair Value of House and Property*. In `Chapter 6`, *Customer Segmentation*, we will use a clustering algorithm to gain insight into customer behavior using e-commerce data. In `Chapter 7`, *Music Genre Recommendation*, and `Chapter 8`, *Handwritten Digit Recognition*, we will build recommendation and image recognition models using audio and image data. Lastly, we will use semi-supervised learning techniques to detect anomalies in `Chapter 9`, *Cyber Attack Detection* and `Chapter 10`, *Credit Card Fraud Detection*.

In this chapter, we will cover the following topics:

- Key ML tasks and applications

- Steps in building ML models
- Setting up a C# environment for ML

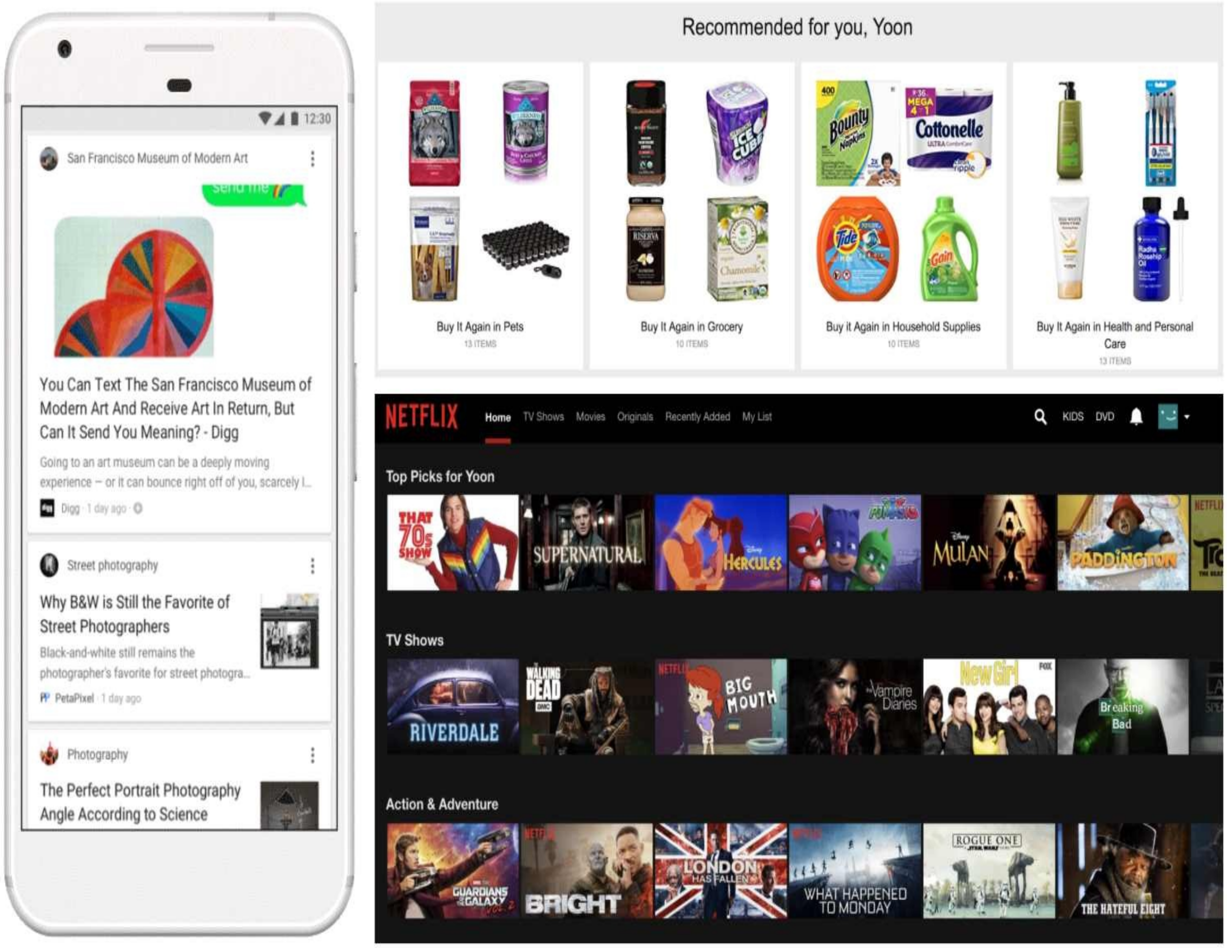
Key ML tasks and applications

There are many areas where ML is used in our daily lives without being noticed. Media companies use ML to recommend the most relevant content, such as news articles, movies, or music, for you to read, watch, or listen to. The e-commerce companies use ML to suggest the items that are of interest and that you are most likely to purchase. Game companies use ML to detect your motion and joint movements for their motion sensor games. Some other common uses of ML in the industry include face detection on cameras for better focusing, automated question answering where chat bots or virtual assistants interact with customers to answer questions and requests, and detecting and preventing fraudulent transactions. In this section, we will take a look at some of the applications we use in our daily lives that utilize ML heavily:

- **Google News feed**: Google News feed uses ML to generate a personalized stream of articles based on the user's interests and other profile data. Collaborative filtering algorithms are frequently used for such recommendation systems and are built from the view history data of their user base. Media companies use such personalized recommendation systems to attract more traffic to their websites and increase the number of subscribers.
- **Amazon product recommendations**: Amazon uses user browse and order history data to train a ML model to recommend products that a user is most likely to purchase. This is a good use case for supervised learning in the e-commerce industry. These recommendation algorithms help e-commerce companies maximize their profit by displaying items that are the most relevant to each user's interests.
- **Netflix movie recommendation**: Netflix uses movie ratings, view history, and preference profiles to recommend other movies that a user might like. They train collaborative filtering algorithms with data to make personalized recommendations. Considering that *More than 80 per cent of the TV shows people watch on Netflix are discovered through the platform's recommendation system* according to an article on Wired (`http://www.wired.co.uk/article/how-do-netflixs-algorithms-work-machine-learning-helps-to-predict-what-viewers-will-like`), this is a very useful and profitable example of ML at a media company.

- **Face detection on cameras**: Cameras detect faces for better focusing and light metering. This is the most frequently used example of computer vision and classification. Also, some photo management software uses clustering algorithms to group similar faces in your images together so that you can search photos by certain people in them later.
- **Alexa – Virtual assistant**: Virtual assistant systems, such as Alexa, can answer questions such as *What's the weather in New York?* or complete certain tasks, such as *Turn on the living room lights*. These kinds of virtual assistant system are typically built using speech recognition, **natural language understanding** (**NLU**), deep learning, and various other machine learning technologies.
- **Microsoft Xbox Kinect**: Kinect can sense how far each object is from the sensor and detect joint positions. Kinect is trained with a randomized decision forest algorithm that builds lots of individual decision trees from depth images.

The following screenshot shows different examples of recommendation systems using ML:



12:30

San Francisco Museum of Modern Art

You Can Text The San Francisco Museum of Modern Art And Receive Art In Return, But Can It Send You Meaning? - Digg

Going to an art museum can be a deeply moving experience — or it can bounce right off of you, scarcely l...

Digg · 1 day ago

Street photography

Why B&W is Still the Favorite of Street Photographers

Black-and-white still remains the photographer's favorite for street photogra...

PetaPixel · 1 day ago

Photography

The Perfect Portrait Photography Angle According to Science

Recommended for you, Yoon

Buy It Again in Pets
13 ITEMS

Buy It Again in Grocery
10 ITEMS

Buy it Again in Household Supplies
10 ITEMS

Buy It Again in Health and Personal Care
13 ITEMS

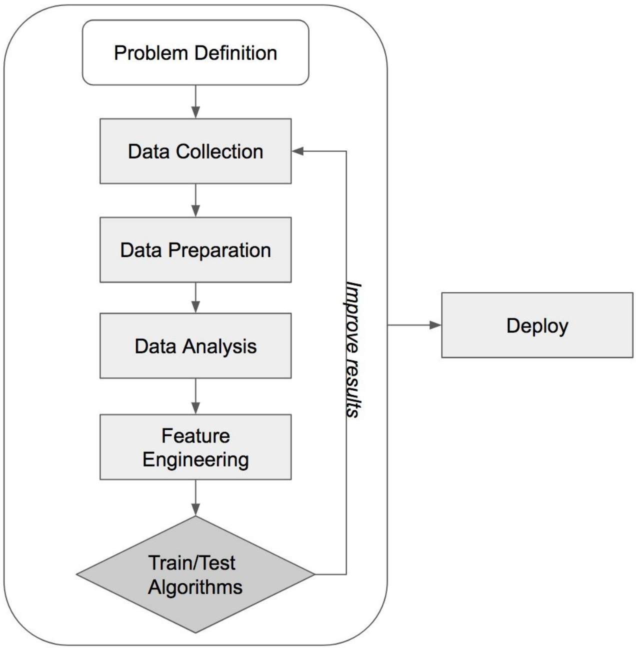
Left: Google News Feed, top-right: Amazon product recommendation, bottom-right: Netflix movie recommendation The following screenshot depicts a few other examples of ML applications:

Alexa, What's the weather like in New York?
Alexa, Turn on the living room lights
Alexa, What's on the news?
amazon
KINECT SPORTS

Left: Face detection, middle: Amazon Alexa, right: Microsoft Xbox Kinect

Steps in building ML models

Now that we have seen some examples of the ML applications that are out there, the question is, *How do we go about building such ML applications and systems?* Books about ML and ML courses that are taught in universities typically start by covering the mathematics and theories behind ML algorithms and then apply those algorithms to a given dataset. This approach is great for people who are completely new to this subject and are looking to learn the foundations of ML. However, aspiring data scientists with some prior knowledge and experience and who are looking to apply their knowledge to real ML projects often stumble about where to start and how to approach a given ML project. In this section, we will discuss a typical workflow for building a ML application, which we will follow throughout the book. The following figure summarizes our approach to developing an application using ML and we will discuss this in more detail in the following subsections:



Steps in building ML models

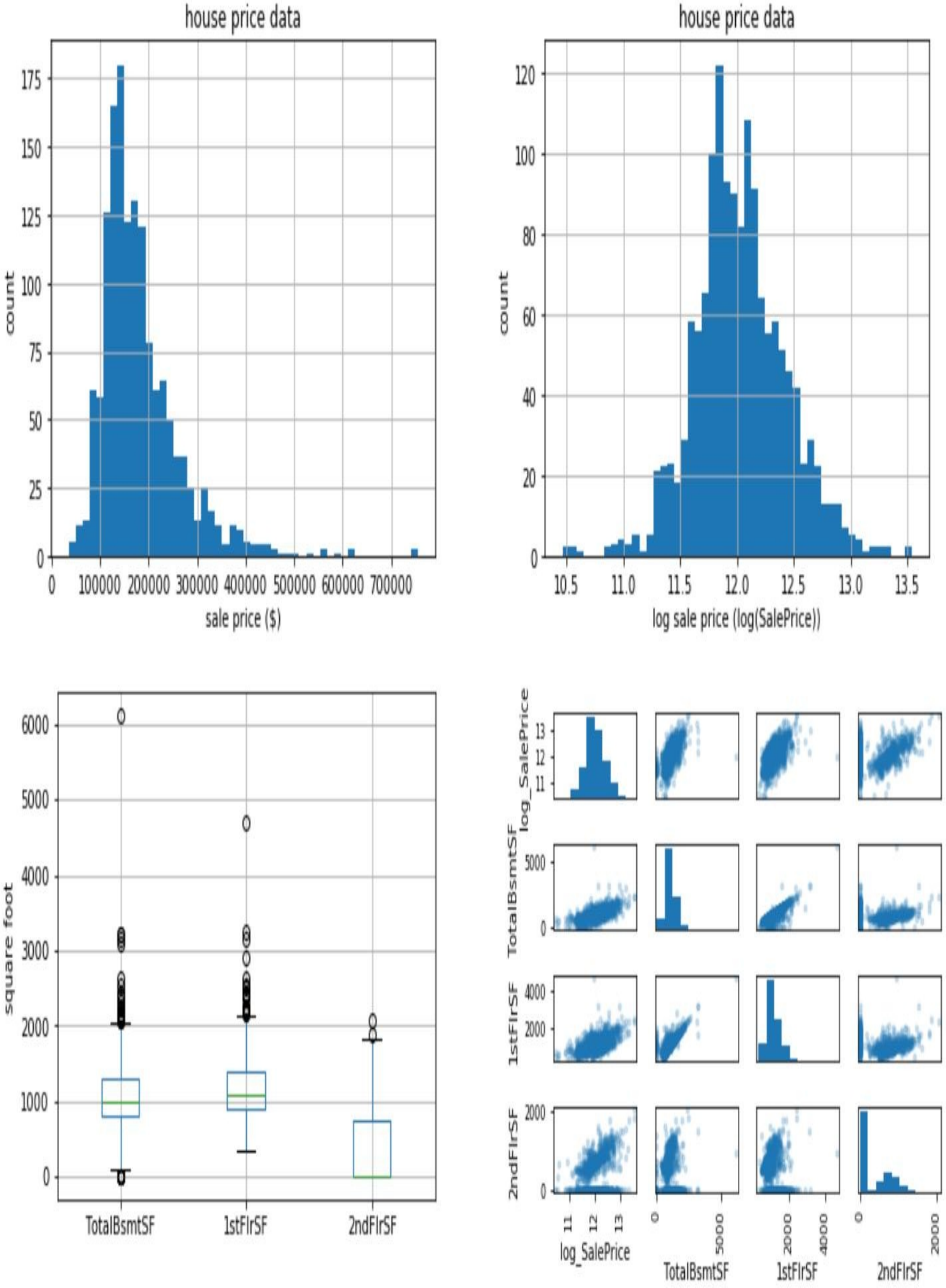
As seen in the preceding diagram, the steps that are to be followed for building learning models are as follows:

- **Problem definition**: The first step in starting any project is not only understanding the problem, but also defining the problem that you are trying to solve using ML. Poor definition of a problem will result in a meaningless ML system, since the models will have been trained and optimized for a problem that you are not actually trying to solve. This first step is unarguably the most important step in building useful ML models and applications. You should at least answer the following four questions before you jump into building ML models:
    - What is the problem? This is where you describe and state the problem that you are trying to solve. For example, a problem description might be *need a system to assess a small business owner's ability to pay back a loan* for a small business lending project.
    - Why is it a problem? It is important to define why such a problem is actually a problem and why the new ML model is going to be useful. Maybe you have a working model already and you have noticed it is performing worse than before; you might have obtained new data sources that you can use for building a new prediction model; or maybe you want your existing model to produce prediction results more quickly. There can be multiple reasons why you think this is a problem and why you need a new model. Defining why it is a problem will help you stay on the right track while you are building a new ML model.
    - What are some of the approaches to solving this problem? This is where you brainstorm your approaches to solve the given problem. You should think about how this model is going to be used (do you need this to be a real-time system or is this going to be run as a batch process?), what type of problem it is (is it a classification problem, regression, clustering, or something else?), and what types of data you would need for your model. This will provide a good basis for future steps in building your machine learning model.
    - What are the success criteria? This is where you define your checkpoints. You should think about what metrics you will look at and what your target model performance should look like. If you are building a model that is going to be used in a real-time system, then you can also set the target execution speed and data availability at runtime as part of your success criteria. Setting these success criteria will help you keep moving forward without being stuck at a certain step.
- **Data collection**: Having data is the most essential and critical part of

building a ML model, preferably lots of data. No data, no model. Depending on your project, your approaches to collecting data can vary. You can purchase existing data sources from other vendors, you can scrape websites and extract data from there, you can use publicly available data, or you can also collect your own data. There are multiple ways you can gather the data you need for your ML model, but you need to keep in mind these two elements of your data when you are in the process of data collection—the target variable and feature variables. The target variable is the answer for your predictions and feature variables are the factors that your models will use to learn how to predict the target variable. Often, target variables are not present in a labeled form. For example, when you are dealing with Twitter data to predict the sentiment of each tweet, you might not have labeled sentiment data for each tweet. In this case, you will have to take an extra step to label your target variables. Once you have your data collected, you can move on to the data preparation step.

- **Data preparation**: Once you have gathered all of your input data, you need to prepare it so that it is in a useable format. This step is more important than you might think. If you have messy data and you did not clean it up for your learning algorithms, your algorithms will not learn well from your dataset and will not perform as expected. Also, even if you have high-quality data, if your data is not in a format that your algorithms can be trained with, then it is meaningless to have high-quality data. Bad data, bad model. You should at least handle some of the common problems listed as follows to have your data ready for the next steps:
    - **File format**: If you are getting your data from multiple data sources, you will most likely run into different formats for each data source. Some data might be in CSV format, while other data is in JSON or XML format. Some data might even be stored in a relational database. In order to train your ML model, you will need to first merge all these data sources in different formats into one standard format.
    - **Data format**: It can also be the case that data formats vary among different data sources. For example, some data might have the address field broken down into street address, city, state, and ZIP, while some others might not. Some data might have the date field in the American date format (mm/dd/yyyy), while some others may be in British format (dd/mm/yyyy). These data format discrepancies among data sources can cause issues when you are parsing the values. In order to train your ML model, you will need to have a uniform data format for each field.
    - **Duplicate records**: Often you will see same exact records repeating in

    your dataset. This problem can occur in the data collection process where you recorded a data point more than once or when you were merging different datasets in your data preparation process. Having duplicate records can adversely affect your model and it is good to check for duplicates in your dataset before you move on to the next steps.
    - **Missing values**: It is also common to see some records with empty or missing values in the data. This can also have an adverse effect when you are training your ML models. There are multiple ways to handle missing values in your data, but you will have to be careful and understand your data very well, as this can change your model performance dramatically. Some of the ways you can handle the missing values include dropping records with missing values, replacing missing values with the mean or median, replacing missing values with a constant, or replacing missing values with a dummy variable and an indicator variable for missing. It will be beneficial to study your data before you deal with the missing values.
- **Data analysis**: Now that your data is ready, it is time to actually look at the data and see if you can recognize any patterns and draw some insights from the data. Summary statistics and plots are two of the best ways to describe and understand your data. For continuous variables, looking at the minimum, maximum, mean, median, and quartiles is a good place to start. For categorical variables, you can look at the counts and percentages of categories. As you are looking at these summary statistics, you can also start plotting graphs to visualize the structures of your data. The following figure shows some commonly used charts for data analysis. Histograms are frequently used to show and inspect underlying distributions of variables, outliers, and skewness. Box plots are frequently used to visualize five-number summary, outliers, and skewness. Pairwise scatter plots are frequently used to detect obvious pairwise correlations among the variables:


house price data
count
175
150
125
100
75
50
25
0
0
100000
200000
300000
400000
500000
600000
700000
sale price ($)
house price data
count
120
100
80
60
40
20
0
10.5
11.0
11.5
12.0
12.5
13.0
13.5
log sale price (log(SalePrice))
square foot
6000
5000
4000
3000
2000
1000
0
TotalBsmtSF
1stFlrSF
2ndFlrSF
log_SalePrice
13
12
11
TotalBsmtSF
5000
0
1stFlrSF
4000
2000
2ndFlrSF
2000
0
11
12
13
log_SalePrice
0
5000
TotalBsmtSF
2000
4000
1stFlrSF
0
2000
2ndFlrSF

Data analysis and visualizations. Top-left: histogram of nominal house sale price, top-right: histogram of house sale price using the logarithmic scale, bottom-left: box plots of distributions of basement, first floor, and second floor square footage's, bottom-right: scatter plot between first and second floor square feet

- **Feature engineering**: Feature engineering is the most important part of the model building process in applied ML. However, this is one of the least discussed topics in many textbooks and ML courses. Feature engineering is the process of transforming raw input data into more informative data for your algorithms to learn from. For example, for your Twitter sentiment prediction model that we will build in `Chapter 3`, *Twitter Sentiment Analysis*, your raw input data may only contain a list of text in one column and a list of sentiment targets in another column. Your ML model will probably not learn how to predict each tweet's sentiment well with this raw data. However, if you transform this data so that each column represents the number of occurrences of each word in each tweet, then your learning algorithm can learn the relationship between the existence of certain words and sentiments more easily. You can also group each word with its adjacent word (bigram) and have the number of occurrences of each bigram in each tweet as another group of features. As you can see from this example, feature engineering is a way of making your raw data more representative and informative of the underlying problems. Feature engineering is not only a science, but also an art. Feature engineering requires good domain knowledge of the dataset, the creativity to build new features from raw input data, and multiple iterations for better results. As we work through this book, we will cover how to build text features using some **natural language processing** (**NLP**) techniques, how to build time series features, how to sub-select features to avoid overfitting issues, and how to use dimensionality reduction techniques to transform high-dimensional data into fewer dimensions.

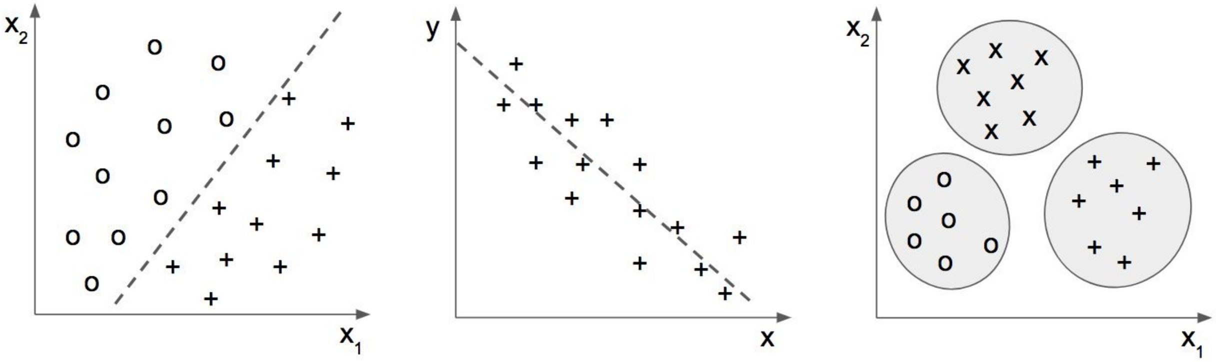
Coming up with features is difficult, time-consuming, requires expert knowledge. Applied machine learning is basically feature engineering.

-Andrew Ng

- **Train/test algorithms**: Once you have created your features, it is time to train and test some ML algorithms. Before you start training your models, it is good to think about performance metrics. Depending on the problem you are solving, your choice of performance measure will differ. For example, if you are building a stock price forecast model, you might want to minimize the difference between your prediction and the actual price and choose **root**

mean square error (**RMSE**) as your performance measure. If you are building a credit model to predict whether a person can be approved for a loan or not, you would want to use the precision rate as your performance measure, since incorrect loan approvals (false positives) will have a more negative impact than incorrect loan disapprovals (false negatives). As we work through the chapters, we will discuss more specific performance metrics for each project.

Once you have specific performance measures in mind for your model, you can now train and test various learning algorithms and their performance. Depending on your prediction target, your choice of learning algorithms will also vary. The following figure shows illustrations of some of the common machine learning problems. If you were solving classification problems, you would want to train classifiers, such as the logistic regression model, the Naive Bayes classifier, or the random forest classifier. On the other hand, if you had a continuous target variable, then you would want to train regressors, such as the linear regression model, k-nearest neighbor, or **Support Vector Machine** (**SVM**). If you would like to draw some insights from data by using unsupervised learning, you would want to use k-means clustering or mean shift algorithms:



Illustrations of ML problems. Left: classification, middle: regression, right: clustering

Lastly, we will have to think about how we test and evaluate the performance of the learning algorithms we tried. Splitting your dataset into train and test sets and running cross-validation are the two most commonly used methods of testing and comparing your ML models. The purpose of splitting a dataset into two subsets, one for training and another for testing, is to train a model on the train set without exposing it to the test set so that prediction results on the test set are indicative of the general model performance for the unforeseen data. K-fold cross-validation is another way to evaluate model performance. It first splits a dataset into equally sized K subsets and leaves one set out for testing and trains on the rest. For example, in 3-fold cross-validation, a dataset will first split into three equally sized subsets. In the first iteration, we will use folds #1 and #2 to train our model and test it on fold #3. In the second iteration, we will use folds #1 and #3 to train and test our model on fold #2, In the third iteration, we will use folds #2 and #3 to train and test our model on fold #1. Then, we will average the performance measures to estimate the model performance:

- **Improve results**: By now you will have one or two candidate models that perform reasonably well, but there might be still some room to improve. Maybe you noticed your candidate models are overfitting to some extent, maybe they do not meet your target performance, or maybe you have some more time to iterate on your models—regardless of your intent, there are multiple ways that you can improve the performance of your model and they are as follows:
    - **Hyperparameter tuning**: You can tune the configurations of your models to potentially improve the performance results. For example, for random forest models, you can tune the maximum height of the tree or number of trees in the forest. For SVMs, you can tune the kernels or cost values.
    - **Ensemble methods**: Ensembling is combining the results of multiple models to get better results. Bagging is where you train the same algorithm on different subsets of your dataset, boosting is combining different models that are trained on the same train set, and stacking is where the output of models is used as the input to a meta model that learns how to combine the results of the sub-models.
    - **More feature engineering**: Iterating on feature engineering is another way to improve model performance.
- **Deploy**: Time to put your models into action! Once you have your models ready, it is time to let them run in production. Make sure you test extensively before your models take full charge. It will also be beneficial to plan to develop monitoring tools for your models, since model performance can decrease over time as the input data evolves.

Setting up a C# environment for ML

Now that we have discussed the steps and approaches to building ML models that we will follow throughout this book, let's start setting up our C# environment for ML. We will first install and set up Visual Studio and then two packages (Accord.NET and Deedle) that we will frequently use for our projects in the following chapters.

Setting up Visual Studio for C#

Assuming you have some prior knowledge of C#, we will keep this part brief. In case you need to install Visual Studio for C#, go to `https://www.visualstudio.com/downloads/` and download one of the versions of Visual Studio. In this book, we use the Community Edition of Visual Studio 2017. If it prompts you to download .NET Framework before you install Visual Studio, go to `https://www.microsoft.com/en-us/download/details.aspx?id=53344` and install it first.

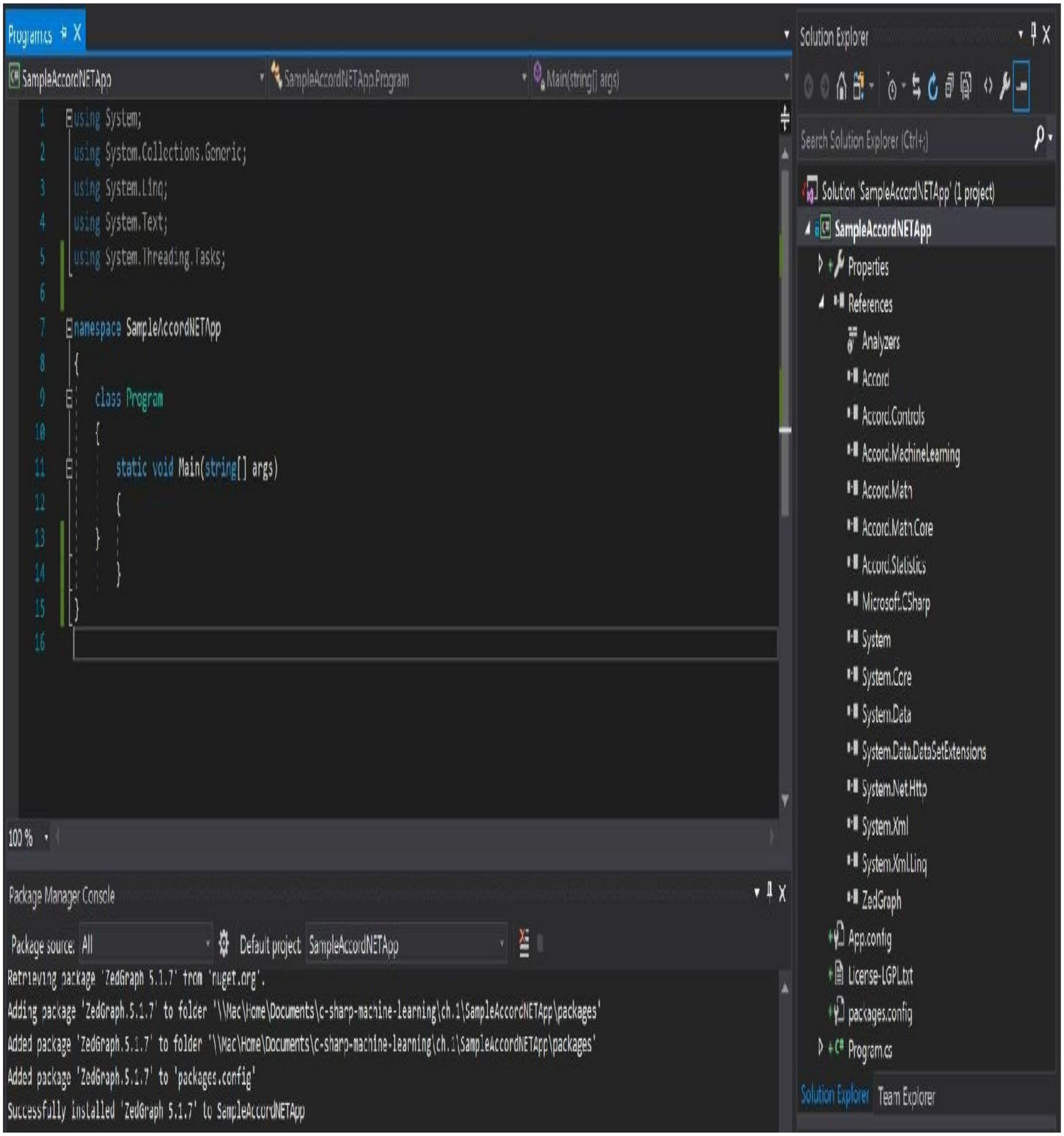
Installing Accord.NET

Accord.NET is a .NET ML framework. On top of ML packages, the Accord.NET framework also has mathematics, statistics, computer vision, computer audition, and other scientific computing modules. We are mainly going to use the ML package of the Accord.NET framework.

Once you have installed and set up your Visual Studio, let's start installing the ML framework for C#, Accord.NET. It is easiest to install it through NuGet. To install it, open the package manager (Tools | NuGet Package Manager | Package Manager Console) and install `Accord.MachineLearning` and `Accord.Controls` by typing in the following commands:

```
PM> Install-Package Accord.MachineLearning
PM> Install-Package Accord.Controls
```

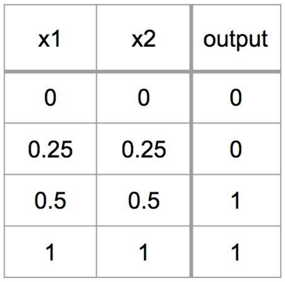
Now, let's build a sample ML application using these Accord.NET packages. Open your Visual Studio and create a new `Console Application` under the Visual C# category. Use the preceding commands to install those Accord.NET packages through `NuGet` and add references to our project. You should see some Accord.NET packages added to your references in your **Solutions Explorer** and the result should look something like the following screenshot:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SampleAccordNETApp
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

```
Retrieving package 'ZedGraph 5.1.7' from 'nuget.org'.
Adding package 'ZedGraph.5.1.7' to folder '\\Mac\Home\Documents\c-sharp-machine-learning\ch.1\SampleAccordNETApp\packages'
Added package 'ZedGraph.5.1.7' to folder '\\Mac\Home\Documents\c-sharp-machine-learning\ch.1\SampleAccordNETApp\packages'
Added package 'ZedGraph.5.1.7' to 'packages.config'
Successfully installed 'ZedGraph 5.1.7' to SampleAccordNETApp
```

The model we are going to build now is a very simple logistic regression model. Given two-dimensional arrays and an expected output, we are going to develop a program that trains a logistic regression classifier and then plot the results showing the expected output and the actual predictions by this model. The input and output for this model look like the following:

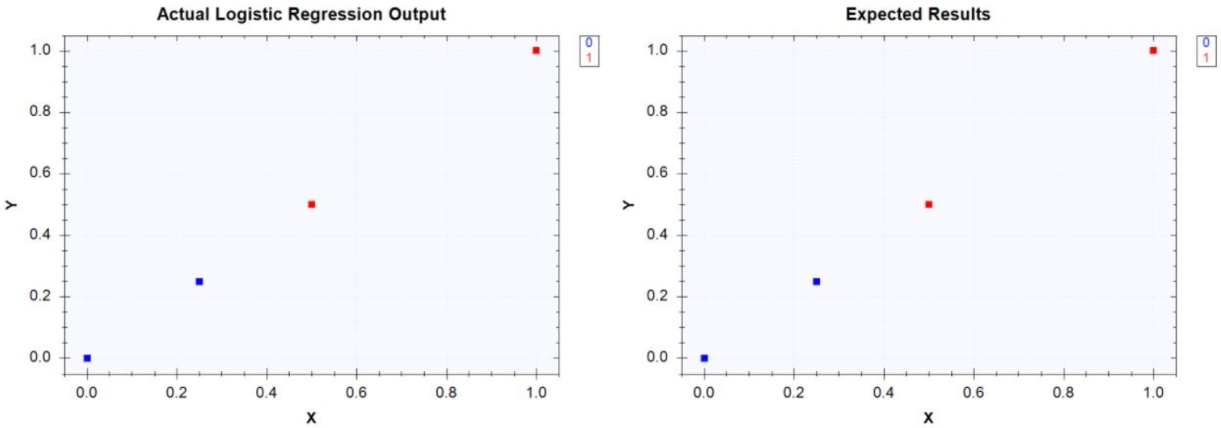


x1	x2	output
0	0	0
0.25	0.25	0
0.5	0.5	1
1	1	1

The code for this sample logistic regression classifier is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Accord.Controls;
using Accord.Statistics;
using Accord.Statistics.Models.Regression;
using Accord.Statistics.Models.Regression.Fitting;

namespace SampleAccordNETApp
{
    class Program
    {
        static void Main(string[] args)
        {
            double[][] inputs =
            {
                new double[] { 0, 0 },
                new double[] { 0.25, 0.25 },
                new double[] { 0.5, 0.5 },
                new double[] { 1, 1 },
            };

            int[] outputs =
            {
                0,
                0,
                1,
                1,
            };

            // Train a Logistic Regression model
            var learner = new IterativeReweightedLeastSquares<LogisticRegression>()
            {
                MaxIterations = 100
            };
            var logit = learner.Learn(inputs, outputs);

            // Predict output
            bool[] predictions = logit.Decide(inputs);

            // Plot the results
            ScatterplotBox.Show("Expected Results", inputs, outputs);
            ScatterplotBox.Show("Actual Logistic Regression Output", inputs,
predictions.ToZeroOne());

            Console.ReadKey();
        }
    }
}
```

Once you are done writing this code, you can run it by hitting *F5* or clicking on the Start button on top. If everything runs smoothly, it should produce the two plots shown in the following figure. If it fails, check for references or typos. You can always right-click on the class name or the light bulb icon to make Visual Studio help you find which packages are missing from the namespace references:



Plots produced by the sample program. Left: actual prediction results, right: expected output

This sample code can be found at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.1/SampleAccordNETApp.cs`.

Installing Deedle

Deedle is an open source .NET library for data frame programming. Deedle lets you do data manipulation in a way that is similar to R data frames and pandas data frames in Python. We will be using this package to load and manipulate the data for our ML projects in the following chapters.

Similar to how we installed Accord.NET, we can install the Deedle package from NuGet. Open the package manager (Tools | NuGet Package Manager | Package Manager Console) and install `Deedle` using the following command:

```
PM> Install-Package Deedle
```

Let's briefly look at how we can use this package to load data from a CSV file and do simple data manipulations. For more information, you can visit http://bluemountaincapital.github.io/Deedle/ for API documentation and sample code. We are going to use daily AAPL stock price data from 2010 to 2013 for this exercise. You can download this data from the following link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.1/table_aapl.csv.

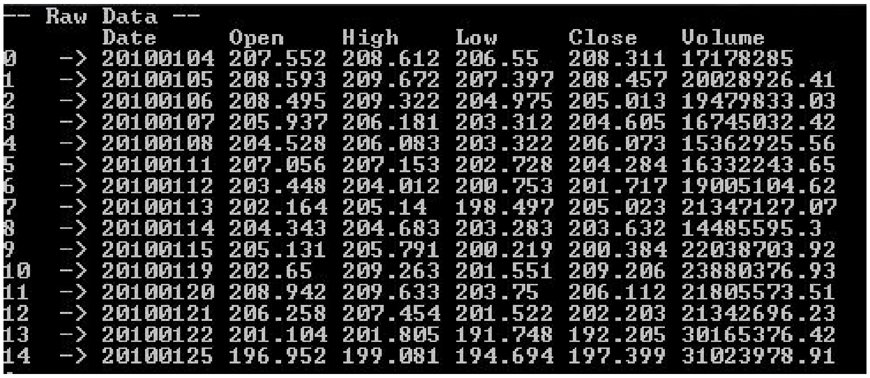
Open your Visual Studio and create a new `Console Application` under the Visual C# category. Use the preceding command to install the `Deedle` library through `NuGet` and add references to your project. You should see the `Deedle` package added to your references in your **Solutions Explorer**.

Now, we are going to load the CSV data into a `Deedle` data frame and then do some data manipulations. First, we are going to update the index of the data frame with the `Date` field. Then, we are going to apply some arithmetic operations on the `Open` and `Close` columns to calculate the percentage changes from open to close prices. Lastly, we will calculate daily returns by taking the differences between the close and the previous close prices, dividing them by the previous close prices, and then multiplying it by `100`. The code for this sample `Deedle` program is shown as follows:

```
using Deedle;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DeedleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Read AAPL stock prices from a CSV file
            var root =
Directory.GetParent(Directory.GetCurrentDirectory()).Parent.FullName;
            var aaplData = Frame.ReadCsv(Path.Combine(root, "table_aapl.csv"));
            // Print the data
            Console.WriteLine("-- Raw Data --");
            aaplData.Print();

            // Set Date field as index
            var aapl = aaplData.IndexRows<String>("Date").SortRowsByKey();
            Console.WriteLine("-- After Indexing --");
            aapl.Print();

            // Calculate percent change from open to close
            var openCloseChange =
                ((
                    aapl.GetColumn<double>("Close") - aapl.GetColumn<double>("Open")
                ) / aapl.GetColumn<double>("Open")) * 100.0;
            aapl.AddColumn("openCloseChange", openCloseChange);
            Console.WriteLine("-- Simple Arithmetic Operations --");
            aapl.Print();

            // Shift close prices by one row and calculate daily returns
            var dailyReturn = aapl.Diff(1).GetColumn<double>("Close") /
aapl.GetColumn<double>("Close") * 100.0;
            aapl.AddColumn("dailyReturn", dailyReturn);
            Console.WriteLine("-- Shift --");
            aapl.Print();

            Console.ReadKey();
        }
    }
}
```

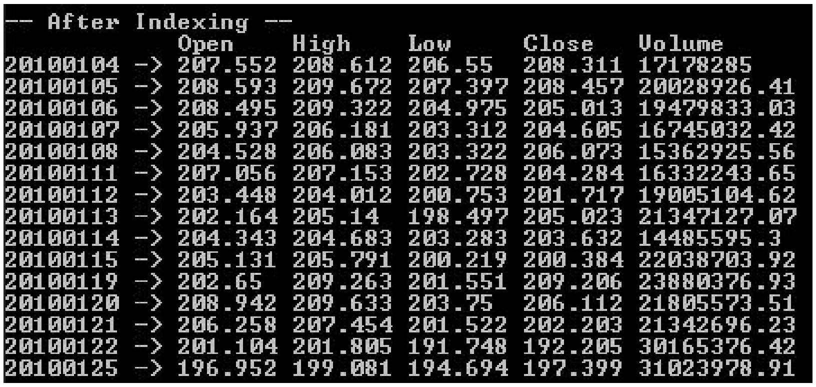
When you run this code, you will see the following outputs.

The raw dataset looks like the following:



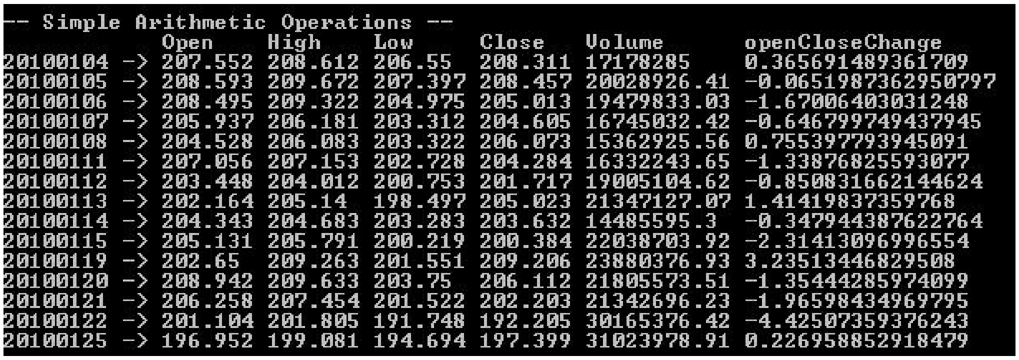
```
-- Raw Data --
      Date     Open    High    Low     Close   Volume
0   -> 20100104 207.552 208.612 206.55  208.311 17178285
1   -> 20100105 208.593 209.672 207.397 208.457 20028926.41
2   -> 20100106 208.495 209.322 204.975 205.013 19479833.03
3   -> 20100107 205.937 206.181 203.312 204.605 16745032.42
4   -> 20100108 204.528 206.083 203.322 206.073 15362925.56
5   -> 20100111 207.056 207.153 202.728 204.284 16332243.65
6   -> 20100112 203.448 204.012 200.753 201.717 19005104.62
7   -> 20100113 202.164 205.14  198.497 205.023 21347127.07
8   -> 20100114 204.343 204.683 203.283 203.632 14485595.3
9   -> 20100115 205.131 205.791 200.219 200.384 22038703.92
10  -> 20100119 202.65  209.263 201.551 209.206 23880376.93
11  -> 20100120 208.942 209.633 203.75  206.112 21805573.51
12  -> 20100121 206.258 207.454 201.522 202.203 21342696.23
13  -> 20100122 201.104 201.805 191.748 192.205 30165376.42
14  -> 20100125 196.952 199.081 194.694 197.399 31023978.91
```

After indexing this dataset with the date field, you will see the following:



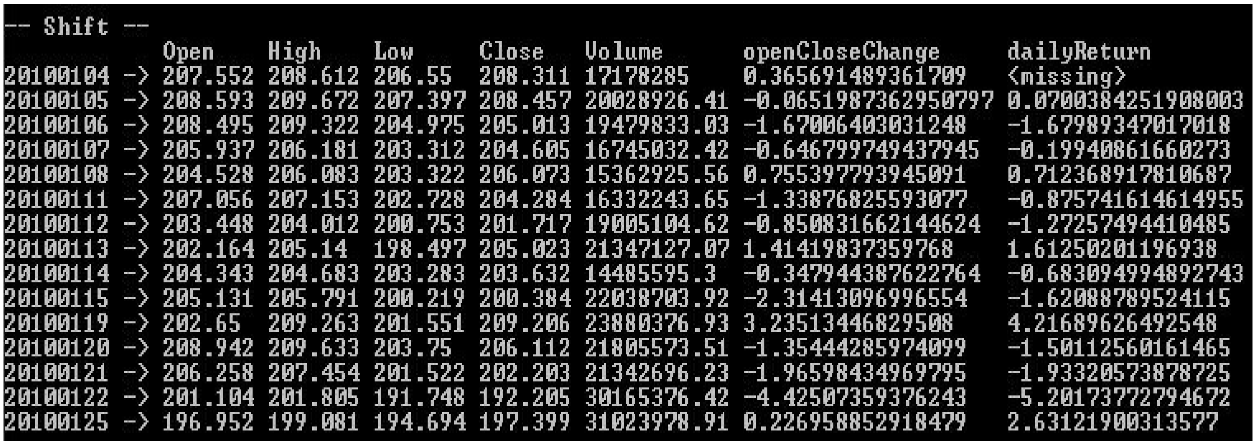
```
-- After Indexing --
            Open    High    Low     Close   Volume
20100104 -> 207.552 208.612 206.55  208.311 17178285
20100105 -> 208.593 209.672 207.397 208.457 20028926.41
20100106 -> 208.495 209.322 204.975 205.013 19479833.03
20100107 -> 205.937 206.181 203.312 204.605 16745032.42
20100108 -> 204.528 206.083 203.322 206.073 15362925.56
20100111 -> 207.056 207.153 202.728 204.284 16332243.65
20100112 -> 203.448 204.012 200.753 201.717 19005104.62
20100113 -> 202.164 205.14  198.497 205.023 21347127.07
20100114 -> 204.343 204.683 203.283 203.632 14485595.3
20100115 -> 205.131 205.791 200.219 200.384 22038703.92
20100119 -> 202.65  209.263 201.551 209.206 23880376.93
20100120 -> 208.942 209.633 203.75  206.112 21805573.51
20100121 -> 206.258 207.454 201.522 202.203 21342696.23
20100122 -> 201.104 201.805 191.748 192.205 30165376.42
20100125 -> 196.952 199.081 194.694 197.399 31023978.91
```

After applying simple arithmetic operations to compute the change rate from open to close, you will see the following:



```
-- Simple Arithmetic Operations --
            Open    High    Low     Close   Volume      openCloseChange
20100104 -> 207.552 208.612 206.55  208.311 17178285    0.365691489361709
20100105 -> 208.593 209.672 207.397 208.457 20028926.41 -0.0651987362950797
20100106 -> 208.495 209.322 204.975 205.013 19479833.03 -1.67006403031248
20100107 -> 205.937 206.181 203.312 204.605 16745032.42 -0.646799749437945
20100108 -> 204.528 206.083 203.322 206.073 15362925.56 0.755397793945091
20100111 -> 207.056 207.153 202.728 204.284 16332243.65 -1.33876825593077
20100112 -> 203.448 204.012 200.753 201.717 19005104.62 -0.850831662144624
20100113 -> 202.164 205.14  198.497 205.023 21347127.07 1.41419837359768
20100114 -> 204.343 204.683 203.283 203.632 14485595.3  -0.347944387622764
20100115 -> 205.131 205.791 200.219 200.384 22038703.92 -2.31413096996554
20100119 -> 202.65  209.263 201.551 209.206 23880376.93 3.23513446829508
20100120 -> 208.942 209.633 203.75  206.112 21805573.51 -1.35444285974099
20100121 -> 206.258 207.454 201.522 202.203 21342696.23 -1.96598434969795
20100122 -> 201.104 201.805 191.748 192.205 30165376.42 -4.42507359376243
20100125 -> 196.952 199.081 194.694 197.399 31023978.91 0.226958852918479
```

Finally, after shifting close prices by one row and computing daily returns, you will see the following:



```
-- Shift --
            Open    High    Low     Close   Volume      openCloseChange     dailyReturn
20100104 -> 207.552 208.612 206.55  208.311 17178285    0.365691489361709   <missing>
20100105 -> 208.593 209.672 207.397 208.457 20028926.41 -0.0651987362950797 0.0700384251908003
20100106 -> 208.495 209.322 204.975 205.013 19479833.03 -1.67006403031248   -1.67989347017018
20100107 -> 205.937 206.181 203.312 204.605 16745032.42 -0.646799749437945  -0.19940861660273
20100108 -> 204.528 206.083 203.322 206.073 15362925.56 0.755397793945091   0.712368917810687
20100111 -> 207.056 207.153 202.728 204.284 16332243.65 -1.33876825593077   -0.875741614614955
20100112 -> 203.448 204.012 200.753 201.717 19005104.62 -0.850831662144624  -1.27257494410485
20100113 -> 202.164 205.14  198.497 205.023 21347127.07 1.41419837359768    1.61250201196938
20100114 -> 204.343 204.683 203.283 203.632 14485595.3  -0.347944387622764  -0.683094994892743
20100115 -> 205.131 205.791 200.219 200.384 22038703.92 -2.31413096996554   -1.62088789524115
20100119 -> 202.65  209.263 201.551 209.206 23880376.93 3.23513446829508    4.21689626492548
20100120 -> 208.942 209.633 203.75  206.112 21805573.51 -1.35444285974099   -1.50112560161465
20100121 -> 206.258 207.454 201.522 202.203 21342696.23 -1.96598434969795   -1.93320573878725
20100122 -> 201.104 201.805 191.748 192.205 30165376.42 -4.42507359376243   -5.20173772794672
20100125 -> 196.952 199.081 194.694 197.399 31023978.91 0.226958852918479   2.63121900313577
```

As you can see from this sample `Deedle` project, we can run various data manipulation operations with one or two lines of code, where it would have required more lines of code to apply the same operations using native C#. We will use the `Deedle` library frequently throughout this book for data manipulation and feature engineering.

This sample Deedle code can be found at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.1/DeedleApp.cs`.

Summary

In this chapter, we briefly discussed some key ML tasks and real-life examples of ML applications. We also learned the steps for developing ML models and the common challenges and tasks in each step. We are going to follow these steps as we work through our projects in the following chapters and we will explore certain steps in more detail, especially for feature engineering, model selection, and model performance evaluations. We will discuss the various techniques we can apply in each step depending on the types of problems we are solving. Lastly, in this chapter, we walked you through how to set up a C# environment for our future ML projects. We built a simple logistic regression classifier using the Accord.NET framework and used the `Deedle` library to load and manipulate the data.

In the next chapter, we are going to dive straight into applying the fundamentals of ML, which we covered in this chapter, to build a ML model for spam email filtering. We will follow the steps for building ML models that we discussed in this chapter to transform raw email data into a structured dataset, analyze the email text data to draw some insights, and then finally build classification models that predict whether an email is a spam or not. We will also discuss some commonly used model evaluation metrics for classification models in the next chapter.

Spam Email Filtering

In this chapter, we are going to start building real-world **machine learning** (**ML**) models in C# using the two packages we installed in `Chapter 1`, *Basics of Machine Learning Modeling*; Accord.NET for ML and Deedle for data manipulation. In this chapter, we are going to build a classification model for spam email filtering. We will be working with a raw email dataset that contains both spam and ham (non-spam) emails and use it to train our ML model. We are going to start following the steps for developing ML models that we discussed in the previous chapter. This will help us understand the workflow and approaches in ML modeling better and make them second nature. While we work to build a spam email classification model, we will also discuss feature engineering techniques for text datasets and basic model validation methods for classification models, and compare the logistic regression classifier and Naive Bayes classifier for spam email filtering. Familiarizing ourselves with these model-building steps, basic text feature engineering techniques, and basic classification model validation methods will lay the groundwork for more advanced feature engineering techniques using **natural language processing** (**NLP**) and for building multi-class classification models in `Chapter 3`, *Twitter Sentiment Analysis*.

In this chapter, we will cover the following topics:

- Problem definition for the spam email filtering project
- Data preparation
- Email data analysis
- Feature engineering for email data
- Logistic regression versus Naive Bayes for spam email filtering
- Classification model validations

Problem definition for the spam email filtering project

Let's start by defining the problem that we are going to solve in this chapter. You are probably familiar with what spam emails are already; spam email filtering is an essential feature for email services such as Gmail, Yahoo Mail, and Outlook. Spam emails can be annoying for users, but they bring more issues and risks with them. For example, a spam email can be designed to solicit credit card numbers or bank account information, which can be used for credit card fraud or money laundering. A spam email can also be used to obtain personal data, such as a social security number or user IDs and passwords, which then can be used for identity theft and various other crimes. Having spam email filtering technology in place is an essential step for email services to save users from being exposed to such crimes. However, having the right spam email filtering solution is difficult. You want to filter out suspicious emails, but at the same time, you do not want to filter too much so that non-spam emails go into the spam folder and never get looked at by users. To solve this problem, we are going to have our ML models learn from the raw email dataset and classify suspicious emails as spam using the subject line. We are going to look at two performance metrics to measure our success: precision and recall. We will discuss these metrics in detail in the following sections.

To summarize our problem definition:

- What is the problem? We need a spam email filtering solution to prevent our users from being victims of fraudulent activities and to improve user experience at the same time.
- Why is it a problem? Having the right balance between filtering suspicious emails and not filtering too much, so that non-spam emails still get the Inbox, is difficult. We are going to rely on ML models to learn how to classify such suspicious emails statistically.
- What are some of the approaches to solving this problem? We will build a classification model that flags potential spam emails based on the subject

lines of emails. We will use precision and recall rates as a way to balance the amount of emails being filtered.

- What are the success criteria? We want high recall rates (the percentage of actual spam emails retrieved over the total number of spam emails) without sacrificing too much for precision rates (the percentage of correctly classified spam emails among those predicted as spam).

Data preparation

Now that we have clearly stated and defined the problem that we are going to solve with ML, we need the data. No data, no ML. Typically, you need to take an extra step prior to the data preparation step to collect and gather the data that you need, but in this book we are going to use a pre-compiled and labeled dataset that is publicly available. In this chapter, we are going to use the CSDMC2010 SPAM corpus dataset (`http://csmining.org/index.php/spam-email-datasets-.html`) to train and test our models. You can follow the link and download the compressed data at the bottom of the web page. When you have downloaded and decompressed the data, you will see two folders named `TESTING` and `TRAINING`, and a text file named `SPAMTrain.label`. The `SPAMTrain.label` file has encoded labels for each email in the `TRAINING` folder—`0` stands for spam and `1` stands for ham (non-spam). We will use this text file with the email data in the `TRAINING` folder to build spam email classification models.

Once you have downloaded the data and put it in a place where you can load it from, you need to prepare it for future feature engineering and model building steps. What we have now is a raw dataset that contains a number of EML files that contain information about individual emails and a text file that contains labeling information. To make this raw dataset usable for building spam email classification models using the email subject lines, we need to do the following tasks:

1. **Extract subject lines from EML files**: The first step to prepare our data for future tasks is to extract the subject and body from individual EML files. We are going to use a package called `EAGetMail` to load and extract information from EML files. You can install this package using the Package Manager in Visual Studio. Take a look at lines 4 to 6 of the code to install the package. Using the `EAGetMail` package, you can easily load and extract the subject and body contents from the EML files (lines 24–30). Once you have extracted the subject and body from an email, you need to append each line of data as a row to a Deedle data frame. Take a look at the `ParseEmails` function from line 18 in the following code to see how to create a Deedle data frame, where each row contains each email's index number, subject line, and body content.

2. **Combine the extracted data with the labels**: After extracting the subject and body contents from individual EML files, there is one more thing we need to do. We need to map the encoded labels (0 for spam versus 1 for ham) to each row of the data frame that we created in the previous step. If you open the `SPAMTrain.label` file with any text editor, you can see that the encoded label is in the first column and the corresponding email file name is in the second column, separated by a space. Using Deedle frame's `ReadCsv` function, you can easily load this label data into a data frame by specifying a space as a separator (see line 50 in the code). Once you have loaded this labeled data into a data frame, you can simply add the first column of this data frame to the other data frame we created in the previous step using the `AddColumn` function of Deedle's frame. Take a look at lines 49-52 of the following code to see how we can combine the labeling information with the extracted email data.
3. **Export this merged data as a CSV file**: Now that we have one data frame that contains both email and labeling data, it is time to export this data frame into a CSV file for future usage. As shown in line 54 in the following code, it takes one line to export the data frame into a CSV file. Using Deedle frame's `SaveCsv` function, you can easily save the data frame as a CSV file.

The code for this data preparation step is as follows:

```
// Install-Package Deedle
// Install-Package FSharp.Core
using Deedle;
// if you don't have EAGetMail package already, install it
// via the Package Manager Console by typing in "Install-Package EAGetMail"
using EAGetMail;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EmailParser
{
    class Program
    {
        private static Frame<int, string> ParseEmails(string[] files)
        {
            // we will parse the subject and body from each email
            // and store each record into key-value pairs
            var rows = files.AsEnumerable().Select((x, i) =>
            {
                // load each email file into a Mail object
                Mail email = new Mail("TryIt");
                email.Load(x, false);
                // extract the subject and body
                string emailSubject = email.Subject;
                string textBody = email.TextBody;

                // create key-value pairs with email id (emailNum), subject, and body
                return new { emailNum = i, subject = emailSubject, body = textBody };
            });

            // make a data frame from the rows that we just created above
            return Frame.FromRecords(rows);
        }

        static void Main(string[] args)
        {
            // Get all raw EML-format files
            // TODO: change the path to point to your data directory
            string rawDataDirPath = "<path-to-data-directory>";
            string[] emailFiles = Directory.GetFiles(rawDataDirPath, "*.eml");

            // Parse out the subject and body from the email files
            var emailDF = ParseEmails(emailFiles);
            // Get the labels (spam vs. ham) for each email
            var labelDF = Frame.ReadCsv(rawDataDirPath + "\\SPAMTrain.label",
hasHeaders: false, separators: " ", schema: "int,string");
            // Add these labels to the email data frame
            emailDF.AddColumn("is_ham", labelDF.GetColumnAt<String>(0));
            // Save the parsed emails and labels as a CSV file
            emailDF.SaveCsv("transformed.csv");

            Console.WriteLine("Data Preparation Step Done!");
            Console.ReadKey();
        }
    }
}
```

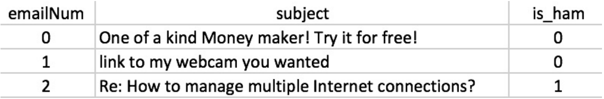
You will need to replace `<path-to-data-directory>` in line 44 with the actual path where you have your data stored before you run this code. Once you run this code, a file named `transformed.csv` should be created and it will contain four columns (`emailNum`, `subject`, `body`, and `is_ham`). We will use this output data as an input to the following steps for building ML models for the spam email filtering project. However, feel free to be creative and play around with the Deedle framework and `EAGetMail` package to tweak and prepare this data in a different way. The code we presented here is one way to prepare this raw email data for future usage and some of the information you can extract from the raw email data. Using the `EAGetMail` package, you can extract other features, such as the sender's email addresses and attachments in the emails, and these extra features can potentially help improve your spam email classification models.

The code for this data preparation step can also be found in the following repository: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.2/EmailParser.cs`.

Email data analysis

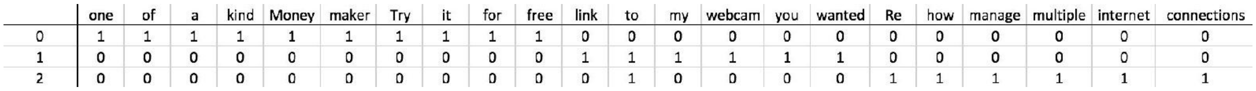
In the data preparation step, we transformed the raw dataset into a more readable and usable dataset. We now have one file to look at to figure out which emails are spam and which emails are not. Also, we can easily find out the subject lines for spam emails and non-spam emails. With this transformed data, let's start looking at what the data actually looks like and see if we can find any patterns or issues within the data.

Since we are dealing with text data, the first thing we want to look at is how the word distributions differ between spam and non-spam emails. In order to do this, we need to transform the data output from the previous step into a matrix representation of word occurrences. Let's work through this step by step, taking the first three subject lines from our data as an example. The first three subject lines we have are as follows:



emailNum	subject	is_ham
0	One of a kind Money maker! Try it for free!	0
1	link to my webcam you wanted	0
2	Re: How to manage multiple Internet connections?	1

If we transform this data such that each column corresponds to each word in each subject line and encode the value of each cell as `1`, if the given subject line has the word, and `0` if not, then the resulting matrix looks something like the following:



	one	of	a	kind	Money	maker	Try	it	for	free	link	to	my	webcam	you	wanted	Re	how	manage	multiple	internet	connections
0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	1

This specific way of encoding is called **one-hot encoding**, where we only care about whether the specific word occurred in the subject line or not and we do not care about the actual number of occurrences of each word in the subject line. In the aforementioned case, we also took out all the punctuation marks, such as colons, question marks, and exclamation points. To do this programmatically, we can use a `regex` to split each subject line into words that only contain alpha-numeric characters and then build a data frame with one-hot encoding. The code to do this encoding step looks like the following:

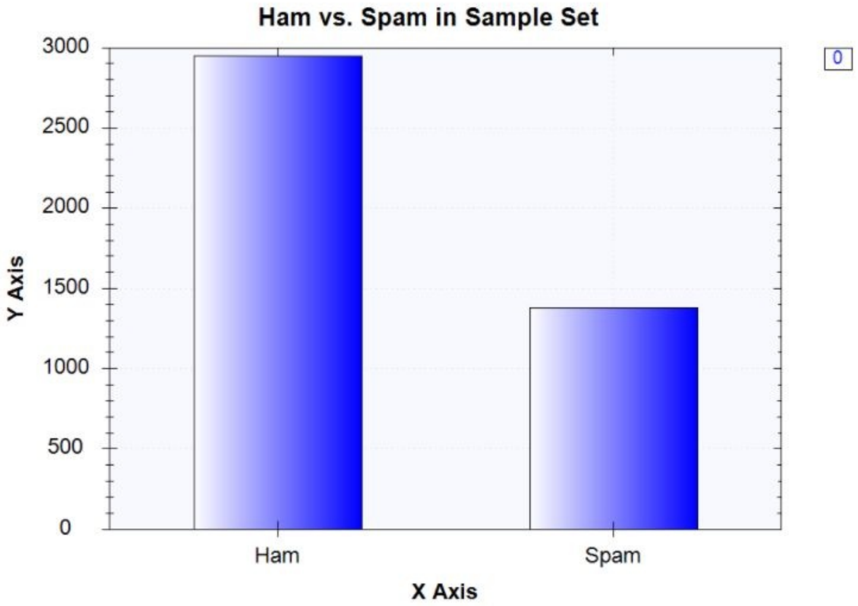
```
private static Frame<int, string> CreateWordVec(Series<int, string> rows)
{
    var wordsByRows = rows.GetAllValues().Select((x, i) =>
    {
        var sb = new SeriesBuilder<string, int>();

        ISet<string> words = new HashSet<string>(
            Regex.Matches(
                // Alphanumeric characters only
                x.Value, "\\w+('(s|d|t|ve|m))?"
            ).Cast<Match>().Select(
                // Then, convert each word to lowercase
                y => y.Value.ToLower()
            ).ToArray()
        );

        // Encode words appeared in each row with 1
        foreach (string w in words)
        {
            sb.Add(w, 1);
        }

        return KeyValue.Create(i, sb.Series);
    });

    // Create a data frame from the rows we just created
    // And encode missing values with 0
    var wordVecDF = Frame.FromRows(wordsByRows).FillMissing(0);

    return wordVecDF;
}
```

Having this one-hot encoded matrix representation of words makes our data analysis process much easier. For example, if we want to take a look at the top ten frequently occurring words in spam emails, we can simply sum the values in each column of the one-hot encoded word matrix for spam emails and take the ten words with the highest summed values. This is exactly what we do in the following code:

```
var hamTermFrequencies = subjectWordVecDF.Where(
    x => x.Value.GetAs<int>("is_ham") == 1
).Sum().Sort().Reversed.Where(x => x.Key != "is_ham");

var spamTermFrequencies = subjectWordVecDF.Where(
    x => x.Value.GetAs<int>("is_ham") == 0
).Sum().Sort().Reversed;

// Look at Top 10 terms that appear in Ham vs. Spam emails
var topN = 10;

var hamTermProportions = hamTermFrequencies / hamEmailCount;
var topHamTerms = hamTermProportions.Keys.Take(topN);
var topHamTermsProportions = hamTermProportions.Values.Take(topN);

System.IO.File.WriteAllLines(
    dataDirPath + "\\ham-frequencies.csv",
    hamTermFrequencies.Keys.Zip(
        hamTermFrequencies.Values, (a, b) => string.Format("{0},{1}", a, b)
    )
);

var spamTermProportions = spamTermFrequencies / spamEmailCount;
var topSpamTerms = spamTermProportions.Keys.Take(topN);
var topSpamTermsProportions = spamTermProportions.Values.Take(topN);

System.IO.File.WriteAllLines(
    dataDirPath + "\\spam-frequencies.csv",
    spamTermFrequencies.Keys.Zip(
        spamTermFrequencies.Values, (a, b) => string.Format("{0},{1}", a, b)
    )
);
```

As you can see from this code, we use the `Sum` method of Deedle's data frame to sum the values in each column and sort in reverse order. We do this once for spam emails and again for ham emails. Then, we use the `Take` method to get the top ten words that appear the most frequently in spam and ham emails. When you run this code, it will generate two CSV files: `ham-frequencies.csv` and `spam-frequencies.csv`. These two files contain information about the number of word occurrences in spam and ham emails, which we are going to use later for the feature engineering and model building steps.

Let's now visualize some of the data for further analysis. First, take a look at the following plot for the top ten frequently appearing terms in ham emails in the dataset:



A bar plot for the top ten frequently appearing terms in ham emails

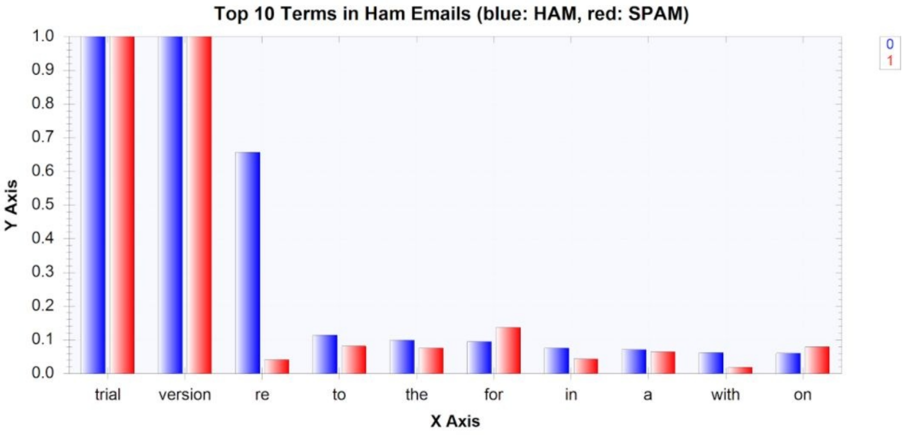
As you can see from this bar chart, there are more ham emails than spam emails in the dataset, as in the real world. We typically get more ham emails than spam emails in our inbox. We used the following code to generate this bar chart to visualize the distribution of ham and spam emails in the dataset:

```
var barChart = DataBarBox.Show(
    new string[] { "Ham", "Spam" },
    new double[] {
        hamEmailCount,
        spamEmailCount
    }
);
barChart.SetTitle("Ham vs. Spam in Sample Set");
```

Using the `DataBarBox` class in the Accord.NET framework, we can easily visualize data in bar charts. Let's now visualize the top ten frequently occurring terms in ham and spam emails. You can use the following code to generate bar charts for the top ten terms in ham and spam emails:

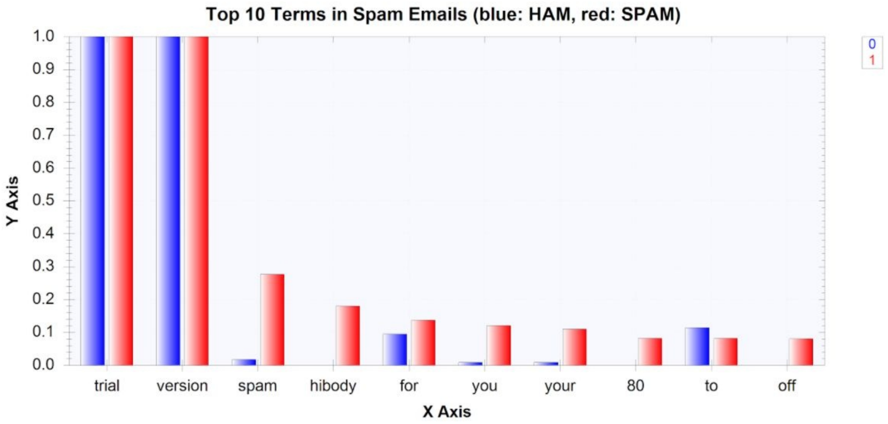
```
var hamBarChart = DataBarBox.Show(
    topHamTerms.ToArray(),
    new double[][] {
        topHamTermsProportions.ToArray(),
        spamTermProportions.GetItems(topHamTerms).Values.ToArray()
    }
);
hamBarChart.SetTitle("Top 10 Terms in Ham Emails (blue: HAM, red: SPAM)");

var spamBarChart = DataBarBox.Show(
    topSpamTerms.ToArray(),
    new double[][] {
        hamTermProportions.GetItems(topSpamTerms).Values.ToArray(),
        topSpamTermsProportions.ToArray()
    }
);
spamBarChart.SetTitle("Top 10 Terms in Spam Emails (blue: HAM, red: SPAM)");
```

Similarly, we used `DataBarBox` class to display bar charts. When you run this code, you will see the following plot for the top ten frequently appearing terms in ham emails:



A plot for the top ten frequently appearing terms in ham emails

The bar plot for the top ten frequently occurring terms in spam emails looks like the following:



A bar plot for the top ten frequently occurring terms in spam emails

As expected, the word distribution in spam emails is quite different from non-spam emails. For example, if you look at the chart on the right, the words **spam** and **hibody** appear frequently in spam emails, but not so much in non-spam emails. However, there is something that does not make much sense. If you look closely, the two words **trial** and **version** appear in all of the spam and ham emails, which is very unlikely to be true. If you open some of the raw EML files in a text editor, you can easily find out that not all of the emails contain those two words in their subject lines. So, what is happening? Did our data get polluted by our previous data preparation or data analysis steps?

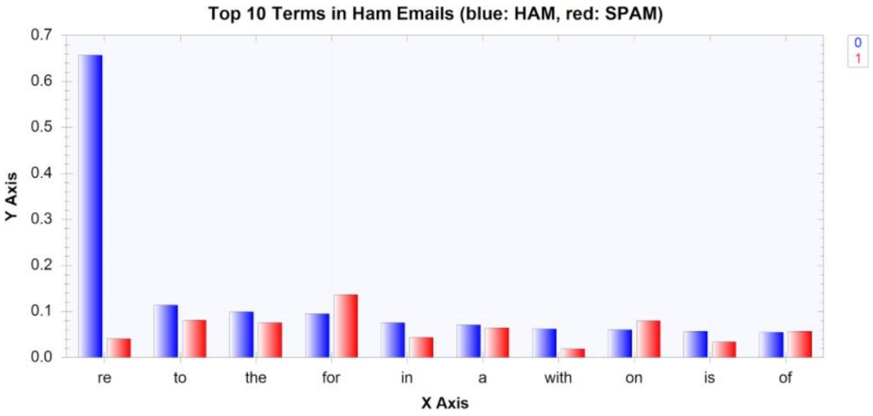
Further research suggests that one of the packages that we used caused this issue.

The `EAGetMail` package, which we used to load and extract email contents, automatically appends `(Trial Version)` to the end of the subject lines when we use their trial version. Now that we know the root cause of this data issue, we need to go back and fix it. One solution is to go back to the data preparation step and update our `ParseEmails` function with the following code, which simply drops the appended `(Trial Version)` flag from the subject lines:

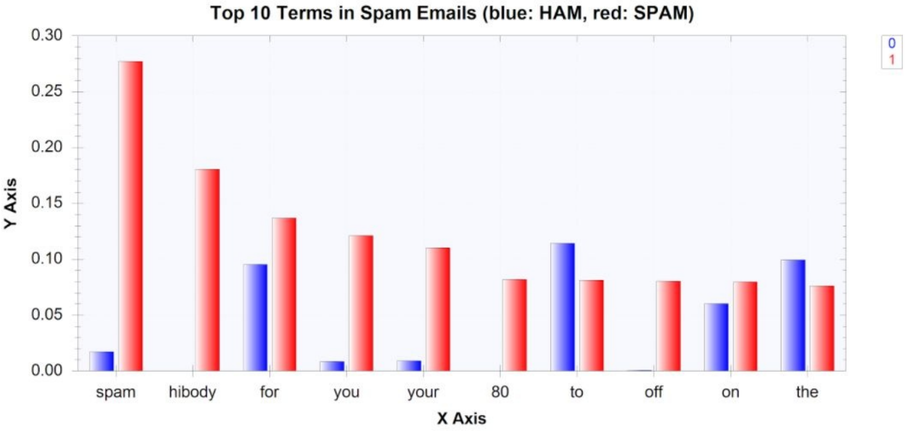
```
private static Frame<int, string> ParseEmails(string[] files)
{
    // we will parse the subject and body from each email
    // and store each record into key-value pairs
    var rows = files.AsEnumerable().Select((x, i) =>
    {
        // load each email file into a Mail object
        Mail email = new Mail("TryIt");
        email.Load(x, false);

        // REMOVE "(Trial Version)" flags
        string EATrialVersionRemark = "(Trial Version)"; // EAGetMail appends subjects
with "(Trial Version)" for trial version
        string emailSubject = email.Subject.EndsWith(EATrialVersionRemark) ?
            email.Subject.Substring(0, email.Subject.Length -
EATrialVersionRemark.Length) : email.Subject;
        string textBody = email.TextBody;

        // create key-value pairs with email id (emailNum), subject, and body
        return new { emailNum = i, subject = emailSubject, body = textBody };
    });

    // make a data frame from the rows that we just created above
    return Frame.FromRecords(rows);
}
```

After updating this code and running the previous data preparation and analysis code again, the bar charts for word distribution make much more sense.

The following bar plot shows the top ten frequently occurring terms in ham emails after fixing and removing `(Trial Version)` flags:



The following bar plot shows the top ten frequently occurring terms in spam emails after fixing and removing `(Trial Version)` flags:



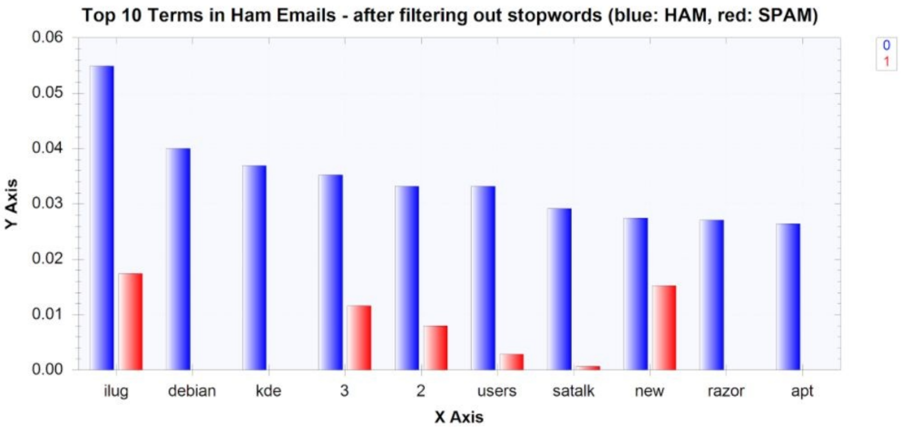
This is a good example of the importance of a data analysis step when building ML models. Iterating between the data preparation and data analysis steps is very common, as we typically find issues with the data in the analysis step and often we can improve the data quality by updating some of the code used in the data preparation step. Now that we have clean data with a matrix representation of words used in subject lines, it is time to start working on the actual features that we will use for building ML models.

Feature engineering for email data

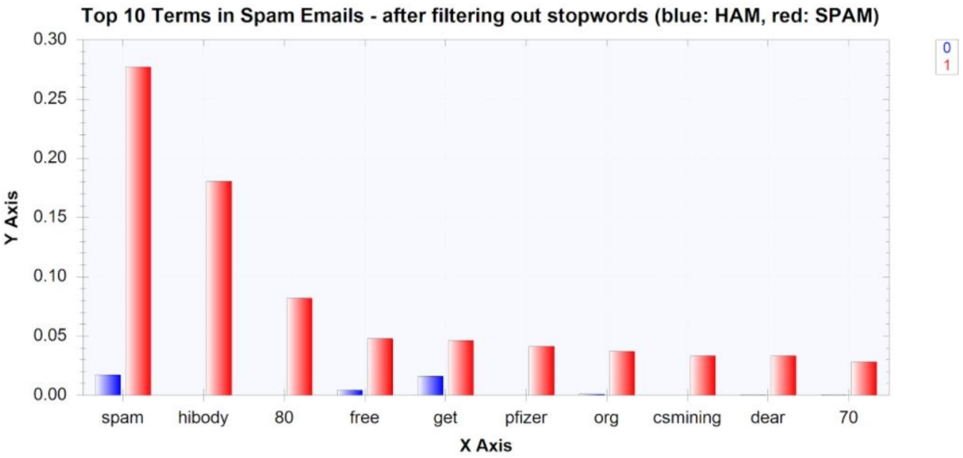
We briefly looked at word distributions for spam and ham emails in the previous step and there are a couple things that we noticed. First, a large number of the most frequently occurring words are commonly used words with out much meaning. For example, words like *to*, *the*, *for*, and *a* are commonly used words and our ML algorithms would not learn much from these words. These type of words are called **stop words** and are often ignored or dropped from the feature set. We will use NLTK's list of stop words to filter out commonly used words from our feature set. You can download the NLTK list of stop words from here: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.2/stopwords.txt. One way to filter out these stop words is shown in the following code:

```
// Read in stopwords list
ISet<string> stopWords = new HashSet<string>(
    File.ReadLines("<path-to-your-stopwords.txt>")
);
// Filter out stopwords from the term frequency series
var spamTermFrequenciesAfterStopWords = spamTermFrequencies.Where(
    x => !stopWords.Contains(x.Key)
);
```

After filtering out these stop words, the new top ten frequently occurring terms for non-spam emails are as follows:



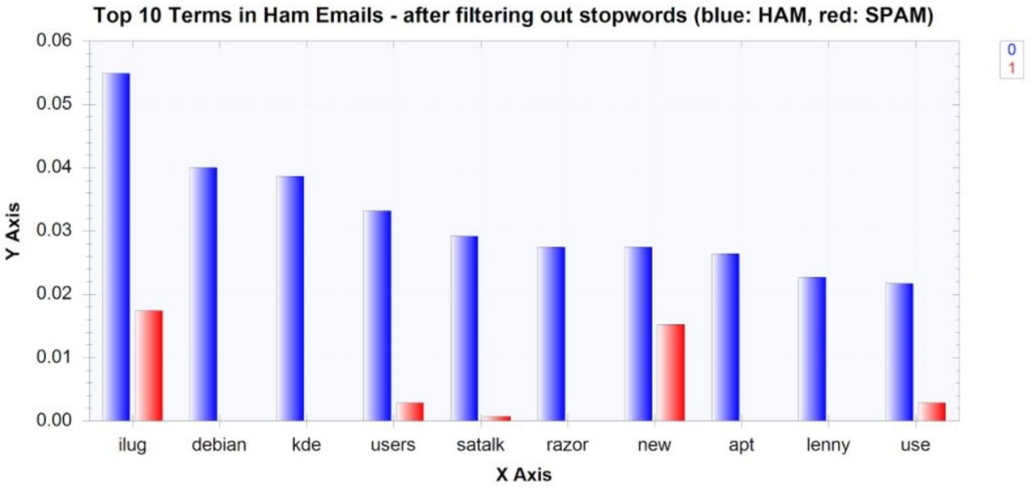
The top ten frequently occurring terms for spam emails, after filtering out stop words, look as the following:



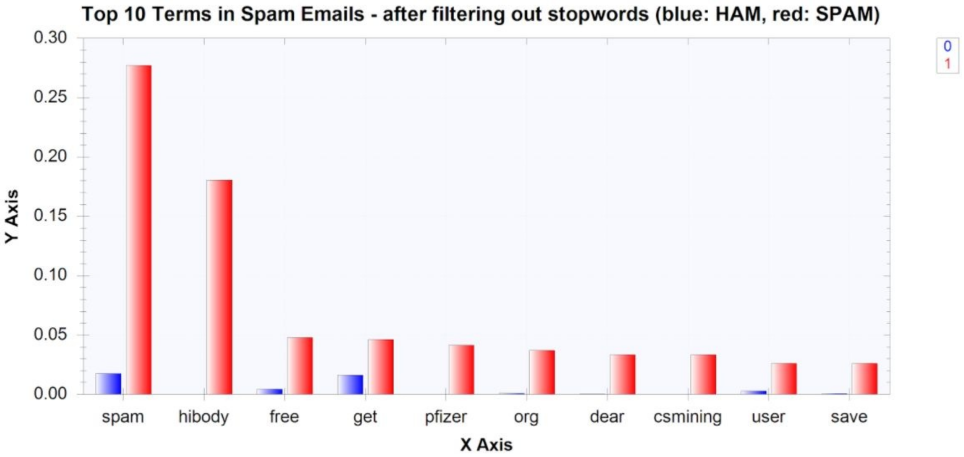
As you can see from these bar charts, filtering out these stop words from the feature set made more meaningful words come to the top of the frequently appearing word lists. However, there is one more thing we can notice here. Numbers seem to come up as some of the top frequently occurring words. For example, the numbers **3** and **2** made it to the top ten frequently appearing words in ham emails. Numbers **80** and **70** made it to the top ten frequently appearing words in spam emails. However, it is hard to establish whether or not those numbers would contribute much in training ML models to classify an email as a spam or ham. There are multiple ways to filter out these numbers from the feature set, but we will show you one way to do it here. We updated the `regex` we used in the previous step to match words that contain alphabetical characters only, not alphanumeric characters. The following code shows how we updated the `CreateWordVec` function to filter out the numbers from the feature set:

```
private static Frame<int, string> CreateWordVec(Series<int, string> rows)
{
    var wordsByRows = rows.GetAllValues().Select((x, i) =>
    {
        var sb = new SeriesBuilder<string, int>();

        ISet<string> words = new HashSet<string>(
            Regex.Matches(
                // Alphabetical characters only
                x.Value, "[a-zA-Z]+('(s|d|t|ve|m))?"
            ).Cast<Match>().Select(
                // Then, convert each word to lowercase
                y => y.Value.ToLower()
            ).ToArray()
        );

        // Encode words appeared in each row with 1
        foreach (string w in words)
        {
            sb.Add(w, 1);
        }

        return KeyValue.Create(i, sb.Series);
    });

    // Create a data frame from the rows we just created
    // And encode missing values with 0
    var wordVecDF = Frame.FromRows(wordsByRows).FillMissing(0);

    return wordVecDF;
}
```

Once we filter out those numbers from the feature set, the word distributions for ham emails looks like the following:



And the word distributions for spam emails, after filtering out the numbers from the feature set, looks like the following:



As you can see from these bar charts, we have more meaningful words on the top lists and there seems to be a greater distinction between the word distributions for spam and ham emails. Those words that frequently appear in spam emails do not seem to appear much in ham emails and vice versa.

The full code for the data analysis and feature engineering step can be found in the following repo: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.2/DataAnalyzer.cs`. Once you run this code, it will generate bar charts that show word distributions in spam and ham emails and two CSV files—one for the list of words in ham emails with the corresponding counts of occurrences and another for the list of words in spam emails with the corresponding counts of occurrences. We are going to use this term frequency output for feature selection processes when we build classification models for spam email filtering in the following model building section.

Logistic regression versus Naive Bayes for email spam filtering

We have come a long way to finally build our very first ML models in C#. In this section, we are going to train logistic regression and Naive Bayes classifiers to classify emails into spam and ham. We are going to run cross-validations with those two learning algorithms to estimate and get a better understanding of how our classification models will perform in practice. As discussed briefly in the previous chapter, in k-fold cross-validation, the training set is divided into *k* equally sized subsets and one of those *k* subsets is held out as a validation set, and the rest of the *k-1* subsets are used to train a model. It then repeats this process *k* times, where different subsets or folds are used in each iteration as a validation set for testing, and the corresponding *k* validation results are then averaged to report a single estimation.

Let's first look at how we can instantiate a cross-validation algorithm with logistic regression in C# using the Accord.NET framework. The code is as follows:

```
var cvLogisticRegressionClassifier = CrossValidation.Create<LogisticRegression,
IterativeReweightedLeastSquares<LogisticRegression>, double[], int>(
    // number of folds
    k: numFolds,
    // Learning Algorithm
    learner: (p) => new IterativeReweightedLeastSquares<LogisticRegression>()
    {
        MaxIterations = 100,
        Regularization = 1e-6
    },
    // Using Zero-One Loss Function as a Cost Function
    loss: (actual, expected, p) => new ZeroOneLoss(expected).Loss(actual),
    // Fitting a classifier
    fit: (teacher, x, y, w) => teacher.Learn(x, y, w),
    // Input with Features
    x: input,
    // Output
    y: output
);

// Run Cross-Validation
var result = cvLogisticRegressionClassifier.Learn(input, output);
```

Let's take a deeper look at this code. We can create a new `CrossValidation` algorithm using the static `Create` function by supplying the type of model to train, the type of learning algorithm to fit the model, the type of input data, and the type of output data. For this example, we created a new `CrossValidation` algorithm with `LogisticRegression` as the model, `IterativeReweightedLeastSquares` as the learning algorithm, a double array as the type of input, and an integer as the type of output (each label). You can experiment with different learning algorithms to train a logistic regression model. In Accord.NET, you have the option to choose the stochastic gradient descent algorithm (`LogisticGradientDescent`) as a learning algorithm to fit a logistic regression model.

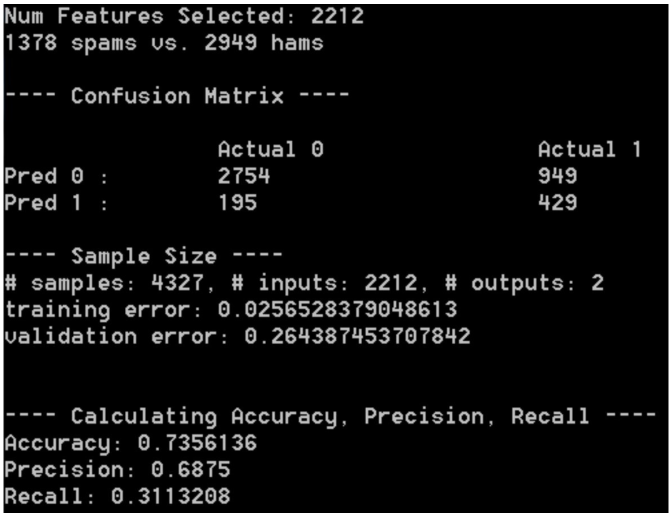
For the parameters, you can specify the number of folds for the k-fold cross-validation (*k*), the learning method with custom parameters (`learner`), the loss/cost function of your choice (`loss`), and a function that knows how to fit a model using the learning algorithm (`fit`), the input (`x`), and the output (`y`). For illustration purposes in this section, we set a relatively small number, `3`, for the k-fold cross-validation. Also, we chose a relatively small number, `100`, for the max iterations and a relatively large number, 1e-6 or 1/1,000,000, for regularization of the `IterativeReweightedLeastSquares` learning algorithm. For the loss function, we used a simple zero-one loss function, where it assigns 0s for the correct predictions and 1s for the incorrect predictions. This is the cost function that our learning algorithm tries to minimize. All of these parameters can be tuned differently. You can choose a different loss/cost function, the number of folds to use in k-fold cross-validation, and the maximum number of iterations and the regularization number for the learning algorithm. You can even use a different learning algorithm to fit a logistic regression model, such as `LogisticGradientDescent`, which iteratively tries to find the local minimum of a loss function.

We can apply this same approach to train the Naive Bayes classifier with a k-fold cross-validation. The code to run k-fold cross-validation with the Naive Bayes learning algorithm is as follows:

```
var cvNaiveBayesClassifier = CrossValidation.Create<NaiveBayes<BernoulliDistribution>,
NaiveBayesLearning<BernoulliDistribution>, double[], int>(
    // number of folds
    k: numFolds,
    // Naive Bayes Classifier with Binomial Distribution
    learner: (p) => new NaiveBayesLearning<BernoulliDistribution>(),
    // Using Zero-One Loss Function as a Cost Function
    loss: (actual, expected, p) => new ZeroOneLoss(expected).Loss(actual),
    // Fitting a classifier
    fit: (teacher, x, y, w) => teacher.Learn(x, y, w),
    // Input with Features
    x: input,
    // Output
    y: output
);

// Run Cross-Validation
var result = cvNaiveBayesClassifier.Learn(input, output);
```

The only difference between the previous code for the logistic regression model and this code is the model and the learning algorithm we chose. Instead of `LogisticRegression` and `IterativeReweightedLeastSquares`, we used `NaiveBayes` as a model and `NaiveBayesLearning` as a learning algorithm to train our Naive Bayes classifier. Since all of our input values are binary (either 0 or 1), we used `BernoulliDistribution` for our Naive Bayes classifier model.

The full code to train and validate a classification model with k-fold cross validation can be found in the following repository: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.2/Modeling.cs. When you run this code, you should see an output that looks like the following:



```
Num Features Selected: 2212
1378 spams vs. 2949 hams

---- Confusion Matrix ----

                Actual 0                Actual 1
Pred 0 :        2754                    949
Pred 1 :        195                     429

---- Sample Size ----
# samples: 4327, # inputs: 2212, # outputs: 2
training error: 0.0256528379048613
validation error: 0.264387453707842


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.7356136
Precision: 0.6875
Recall: 0.3113208
```

We will take a closer look at what these numbers represent in the following section where we discuss model validation methods. In order to try different ML models, simply modify lines 68–88 in the code. You can replace these with the logistic regression model code that we discussed previously or you can also try fitting a different learning algorithm of your choice.

Classification model validations

We built our very first ML models in C# using the Accord.NET framework in the previous section. However, we are not quite done yet. If we look at the previous console output more closely, there is one thing that is quite concerning. The training error is around 0.03, but the validation error is about 0.26. This means that our classification model predicted correctly 87 out of 100 times in the training set, but the model predictions in the validation or test set were correct only 74 times out of 100. This is a typical example of overfitting, where the model fits so closely to the train set that its predictions for the unforeseen dataset are unreliable and unpredictable. If we were to take this model and put it in the production spam filtering system, the model performance in practice for filtering spam emails would be unreliable and would be different from what we saw in the training set.

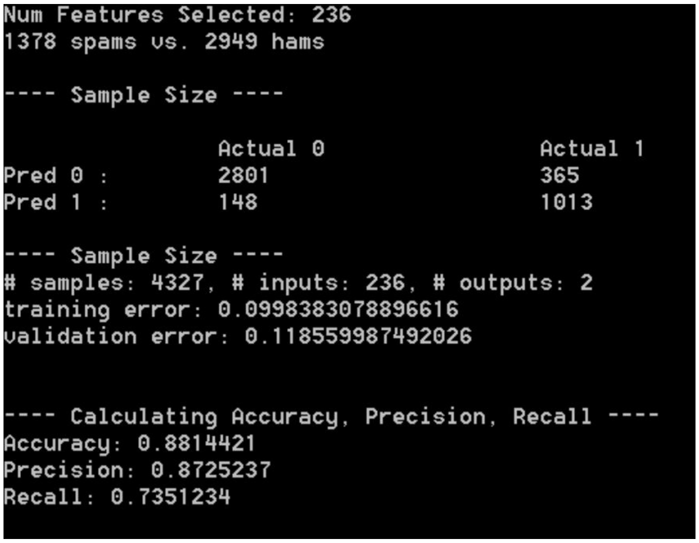
Overfitting typically happens because the model is too complex for the given dataset or too many parameters were used to fit the model. The overfitting problem with the Naive Bayes classifier model we built in the last section is most likely due to the complexity and the number of features we used to train the model. If you look at the console output at the end of the last section again, you can see that the number of features used to train our Naive Bayes model was 2,212. This is way too many features, considering that we only have about 4,200 email records in our sample set and only about two thirds of them (or about 3,000 records) were used to train our model (this is because we used 3-fold cross-validation and only two of those three folds were used as a training set in each iteration). To fix this overfitting issue, we will have to reduce the number of features we use to train a model. In order to do this, we can filter out those terms that occur not so often. The code to do this is in lines 48–53 of the full code in the previous section, which looks like the following:

```
// Change number of features to reduce overfitting
int minNumOccurrences = 1;
string[] wordFeatures = indexedSpamTermFrequencyDF.Where(
x => x.Value.GetAs<int>("num_occurences") >= minNumOccurrences
).RowKeys.ToArray();
Console.WriteLine("Num Features Selected: {0}", wordFeatures.Count());
```

As you can see from this code, the Naive Bayes classifier model that we built in the previous section used all the words that appeared in the spam emails at least once. If you look at the word frequencies in spam emails, there are about 1,400 words that only occur once (take a look at the `spam-frequencies.csv` file that was created in the data analysis step). Intuitively, those words with a low number of occurrences would only create noise, not much information for our models to learn. This immediately tells us how much noise our model would have been exposed to when we initially built our classification model in the previous section.

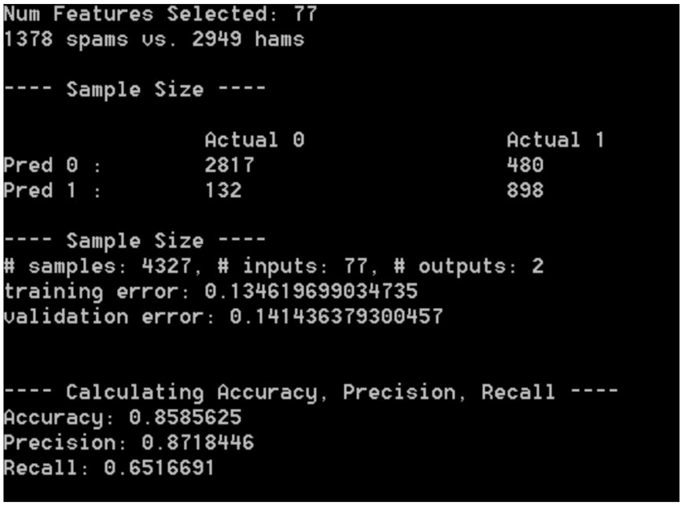
Now that we know the cause of this overfitting issue, let's fix it. Let's experiment with different thresholds for selecting features. We have tried 5, 10, 15, 20, and 25 for the minimum number of occurrences in spam emails (that is, we set `minNumOccurrences` to 5, 10, 15, and so on) and trained Naive Bayes classifiers with these thresholds.

First, the Naive Bayes classifier results with a minimum of five occurrences looks like the following:



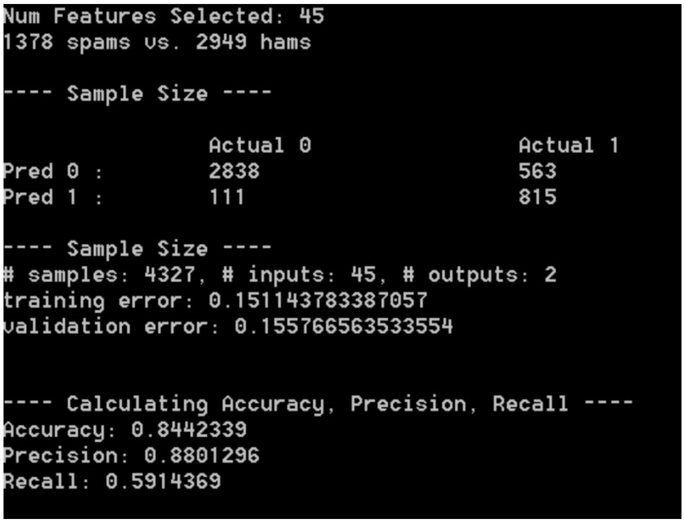
```
Num Features Selected: 236
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2801                    365
Pred 1 :        148                     1013

---- Sample Size ----
# samples: 4327, # inputs: 236, # outputs: 2
training error: 0.0998383078896616
validation error: 0.118559987492026


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8814421
Precision: 0.8725237
Recall: 0.7351234
```

The Naive Bayes classifier results with a minimum of 10 occurrences looks like the following:



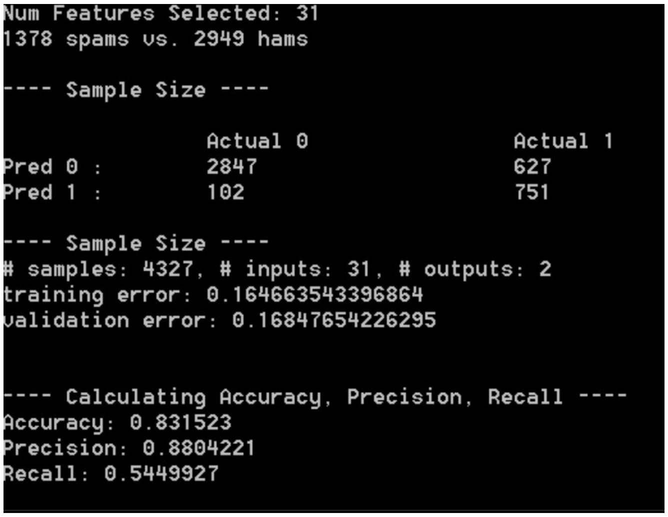
```
Num Features Selected: 77
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2817                    480
Pred 1 :        132                     898

---- Sample Size ----
# samples: 4327, # inputs: 77, # outputs: 2
training error: 0.134619699034735
validation error: 0.141436379300457


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8585625
Precision: 0.8718446
Recall: 0.6516691
```

The Naive Bayes classifier results with a minimum of 15 occurrences looks like the following:



```
Num Features Selected: 45
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2838                    563
Pred 1 :        111                     815

---- Sample Size ----
# samples: 4327, # inputs: 45, # outputs: 2
training error: 0.151143783387057
validation error: 0.155766563533554


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8442339
Precision: 0.8801296
Recall: 0.5914369
```

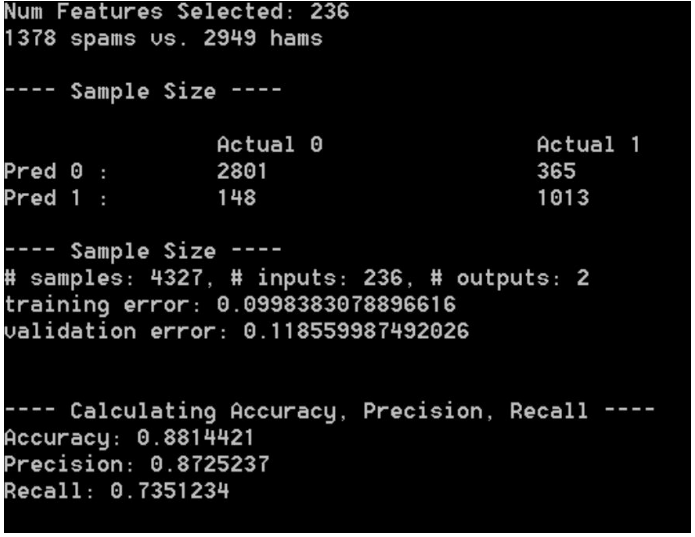
Lastly, the Naive Bayes classifier results with a minimum of 20 occurrences looks like the following:



```
Num Features Selected: 31
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2847                    627
Pred 1 :        102                     751

---- Sample Size ----
# samples: 4327, # inputs: 31, # outputs: 2
training error: 0.164663543396864
validation error: 0.16847654226295


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.831523
Precision: 0.8804221
Recall: 0.5449927
```

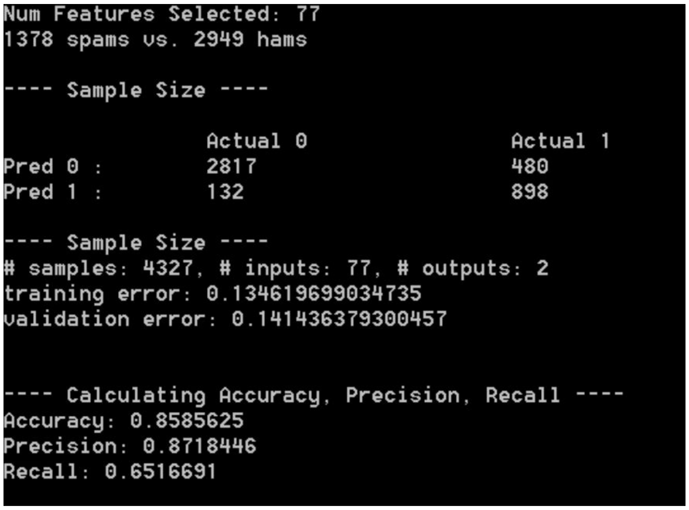
As you can see from these experiment results, as we increase the minimum number of word occurrences and reduce the number of features being used to train the model accordingly, the gap between `training error` and `validation error` decreases and the training errors start to look more similar to the validation errors. As we resolve the overfitting issues, we can be more confident in how the model will behave for the unforeseen data and in production systems. We ran the same experiment with the logistic regression classification model and the results are similar to what we have found with the Naive Bayes classifiers. The experiment results for the logistic regression model are shown in the following outputs.

First, the logistic regression classifier results with a minimum of five occurrences looks like the following:



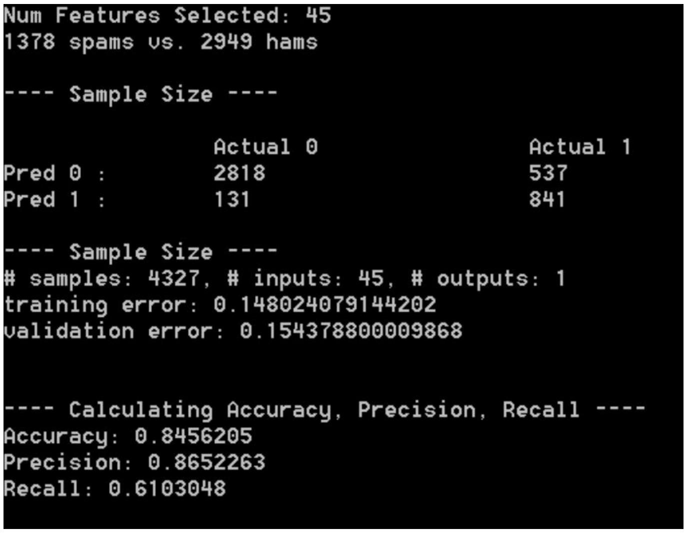
```
Num Features Selected: 236
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2801                    365
Pred 1 :        148                     1013

---- Sample Size ----
# samples: 4327, # inputs: 236, # outputs: 2
training error: 0.0998383078896616
validation error: 0.118559987492026


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8814421
Precision: 0.8725237
Recall: 0.7351234
```

The logistic regression classifier results with a minimum of ten occurrences looks like the following:



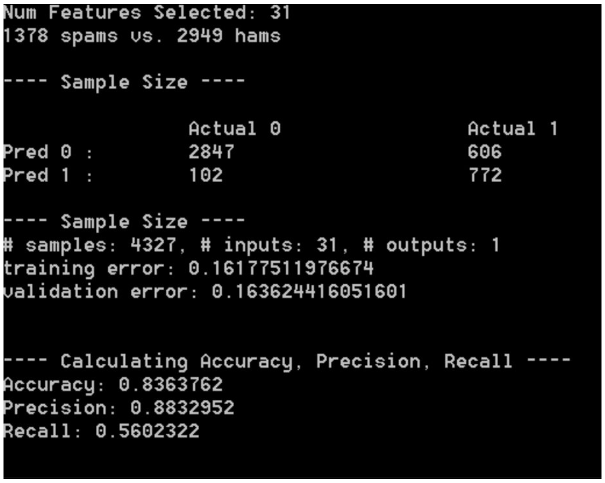
```
Num Features Selected: 77
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2817                    480
Pred 1 :        132                     898

---- Sample Size ----
# samples: 4327, # inputs: 77, # outputs: 2
training error: 0.134619699034735
validation error: 0.141436379300457


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8585625
Precision: 0.8718446
Recall: 0.6516691
```

The logistic regression classifier results with a minimum of 15 occurrences looks like the following:



```
Num Features Selected: 45
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2818                    537
Pred 1 :        131                     841

---- Sample Size ----
# samples: 4327, # inputs: 45, # outputs: 1
training error: 0.148024079144202
validation error: 0.154378800009868


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8456205
Precision: 0.8652263
Recall: 0.6103048
```

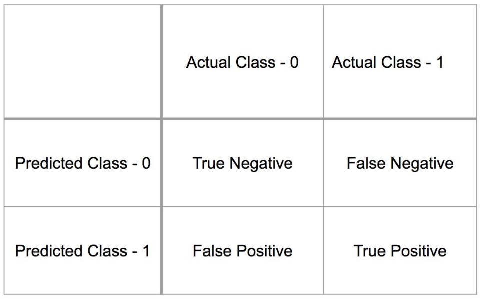
The logistic regression classifier results with a minimum of 20 occurrences looks like the following:



```
Num Features Selected: 31
1378 spams vs. 2949 hams

---- Sample Size ----

                Actual 0                Actual 1
Pred 0 :        2847                    606
Pred 1 :        102                     772

---- Sample Size ----
# samples: 4327, # inputs: 31, # outputs: 1
training error: 0.16177511976674
validation error: 0.163624416051601


---- Calculating Accuracy, Precision, Recall ----
Accuracy: 0.8363762
Precision: 0.8832952
Recall: 0.5602322
```

Now that we have covered how we can handle overfitting issues, there are a few more model performance metrics we want to look at:

- **Confusion matrix**: Confusion matrix is a table that tells us the overall performance of a prediction model. Each column represents each of the actual classes and each row represents each of the predicted classes. In the case of a binary classification problem, the confusion matrix will be a 2 x 2 matrix, where the first row represents negative predictions and the second row represents positive predictions. The first column represents actual negatives and the second column represents actual positives. The following table illustrates what each of the cells in the confusion matrix for a binary classification problem represents:



	Actual Class - 0	Actual Class - 1
Predicted Class - 0	True Negative	False Negative
Predicted Class - 1	False Positive	True Positive

- **True Negative** (**TN**) is when the model predicted class 0 correctly; **False Negative** (**FN**) is when the model prediction is **0**, but the actual class is **1**; **False Positive** (**FP**) is when the model prediction is class **1**, but the actual class is **0**; and **True Positive** (**TP**) is when the model predicted class **1** correctly. As you can see from the table, a confusion matrix describes the overall model performance. In our example, if we look at the last console output in the previous screenshots where it shows the console output of our logistic regression classification model, we can see that the number of TNs is `2847`, the number of FNs is `606`, the number of FPs is `102`, and the number of TPs is `772`. With this information, we can further calculate the **true positive rates** (**TPR**), **true negative rates** (**TNR**), **false positive rates** (**FPR**), and **false negative rates** (**FNR**) as follows:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{FNR} = \frac{\text{FN}}{\text{FN} + \text{TP}}$$

Using the preceding example, the true positive rate in our example is 0.56, the TNR is 0.97, the FPR is 0.03, and the FNR is 0.44.

- **Accuracy**: Accuracy is the proportion of correct predictions. Using the same notations from the previous example confusion matrix, the accuracy can be calculated as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

Accuracy is a frequently used model performance metric, but sometimes it is not a good representation of the overall model performance. For instance, if the sample set is largely unbalanced, and if, say, there are five spam emails and 95 hams in our sample set, then a simple classifier that classifies every email as ham will have to be 95% accurate. However, it will never catch spam emails. This is the reason why we need to look at confusion matrixes and other performance metrics, such as precision and recall rates:

- **Precision rate**: Precision rate is the proportion of the number of correct positive predictions over the total number of positive predictions. Using the same notation as before, we can calculate the precision rate as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

If you look at the last console output in the previous screenshots of our logistic regression classification model results, the precision rate was calculated by dividing the number of TPs in the confusion matrix, 772, by the sum of TPs, 772, and FPs, 102, from the confusion matrix, and the result was 0.88.

- **Recall rate**: Recall rate is the proportion of the number of correct positive predictions over the total number of actual positive cases. This is a way of telling us how many of the actual positive cases are retrieved by this model. Using the same notation as before, we can compute the recall rate as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP + FN}}$$

  If you look at the last console output in the previous screenshots for our logistic regression classification mode results, the recall rate was calculated by dividing the number of TPs in the confusion matrix, 772, by the sum of TPs, 772, and FNs, 606, from the confusion matrix, and the result was 0.56.

With these performance metrics, it is the data scientist's duty to choose the optimal model. There will always be a trade-off between precision and recall rates. A model with a higher precision rate than others will have a lower recall rate. In the case of our spam filtering problem, if you believe correctly filtering out spam emails is more important and that you can sacrifice some of the spam emails going through your users' Inboxes, then you might want to optimize for precision. On the other hand, if you believe filtering out as many spam emails as possible is more important, even though you might end up filtering out some non-spam emails as well, then you might want to optimize for recall. Choosing the right model is not an easy decision and thinking through the requirements and success criteria will be essential in making the right choice.

In summary, the following are the code we can use to compute performance metrics from the cross-validation result and confusion matrix:

- **Training versus validation (test) errors**: Used to identify overfitting issues (lines 48–52):

```
// Run Cross-Validation
var result = cvNaiveBayesClassifier.Learn(input, output);

// Training Error vs. Test Error
double trainingError = result.Training.Mean;
double validationError = result.Validation.Mean;
```

- **Confusion matrix**: True Positives versus False Positives and True Negatives versus False Negatives (lines 95–108):

```
// Confusion Matrix
GeneralConfusionMatrix gcm = result.ToConfusionMatrix(input, output);

float truePositive = (float)gcm.Matrix[1, 1];
float trueNegative = (float)gcm.Matrix[0, 0];
float falsePositive = (float)gcm.Matrix[1, 0];
float falseNegative = (float)gcm.Matrix[0, 1];
```

- **Accuracy versus precision versus recall**: Used to measure the correctness of ML models (lines 122–130):

```
// Accuracy vs. Precision vs. Recall
float accuracy = (truePositive + trueNegative) / numberOfSamples;
float precision = truePositive / (truePositive + falsePositive);
float recall = truePositive / (truePositive + falseNegative);
```

Summary

In this chapter, we built our very first ML models in C# that can be used for spam email filtering. We first defined and clearly stated what we were trying to solve and what the success criteria would be. Then, we extracted the relevant information from the raw email data and transformed it into a format that we could use for the data analysis, feature engineering, and ML model building steps. In the data analysis step, we learned how to apply one-hot encoding and built a matrix representation of words used in subject lines. We also identified a data issue from our data analysis process and learned how we often iterate back and forth between the data preparation and analysis steps. Then, we further improved our feature set by filtering out stop words and using a `regex` to split by non-alphanumeric or non-alphabetical words. With this feature set, we built our very first classification models using the logistic regression and Naive Bayes classifier algorithms, briefly covered the danger of overfitting, and learned how to evaluate and compare model performance by looking at accuracy, precision, and recall rates. Lastly, we also learned the trade-off between precision and recall and how to choose a model based on these metrics and business requirements.

In the next chapter, we are going to further expand our knowledge and skills in building classification models using a text dataset. We will start looking at a dataset where we have more than two classes by using Twitter sentiment data. We are going to learn the difference between the binary classification model and the multi-class classification model. We will also discuss some other NLP techniques for feature engineering and how to build a multi-class classification model using the random forest algorithm.

Twitter Sentiment Analysis

In this chapter, we are going to expand our knowledge of building classification models in C#. Along with the two packages, Accord.NET and Deedle, which we used in the previous chapter, we are going to start using the Stanford CoreNLP package to apply more advanced **natural language processing** (**NLP**) techniques, such as tokenization, **part of speech** (**POS**) tagging, and lemmatization. Using these packages, our goal for this chapter is to build a multi-class classification model that predicts the sentiments of tweets. We will be working with a raw Twitter dataset that contains not only words, but also emoticons, and will use it to train a **machine learning** (**ML**) model for sentiment prediction. We will be following the same steps that we follow when building ML models. We are going to start with the problem definition and then data preparation and analysis, feature engineering, and model development and validation. During our feature engineering step, we will expand our knowledge of NLP techniques and explore how we can apply tokenization, POS tagging, and lemmatization to build more advanced text features. In the model building step, we are going to explore a new classification algorithm, a random forest classifier, and compare its performance to the Naive Bayes classifier. Lastly, in our model validation step, we are going to expand our knowledge of confusion matrixes, precision, and recall, which we covered in the previous chapter, and discuss what the **Receiver Operating Characteristic** (**ROC**) curve and **area under the curve** (**AUC**) are and how these concepts can be used to evaluate our ML models.

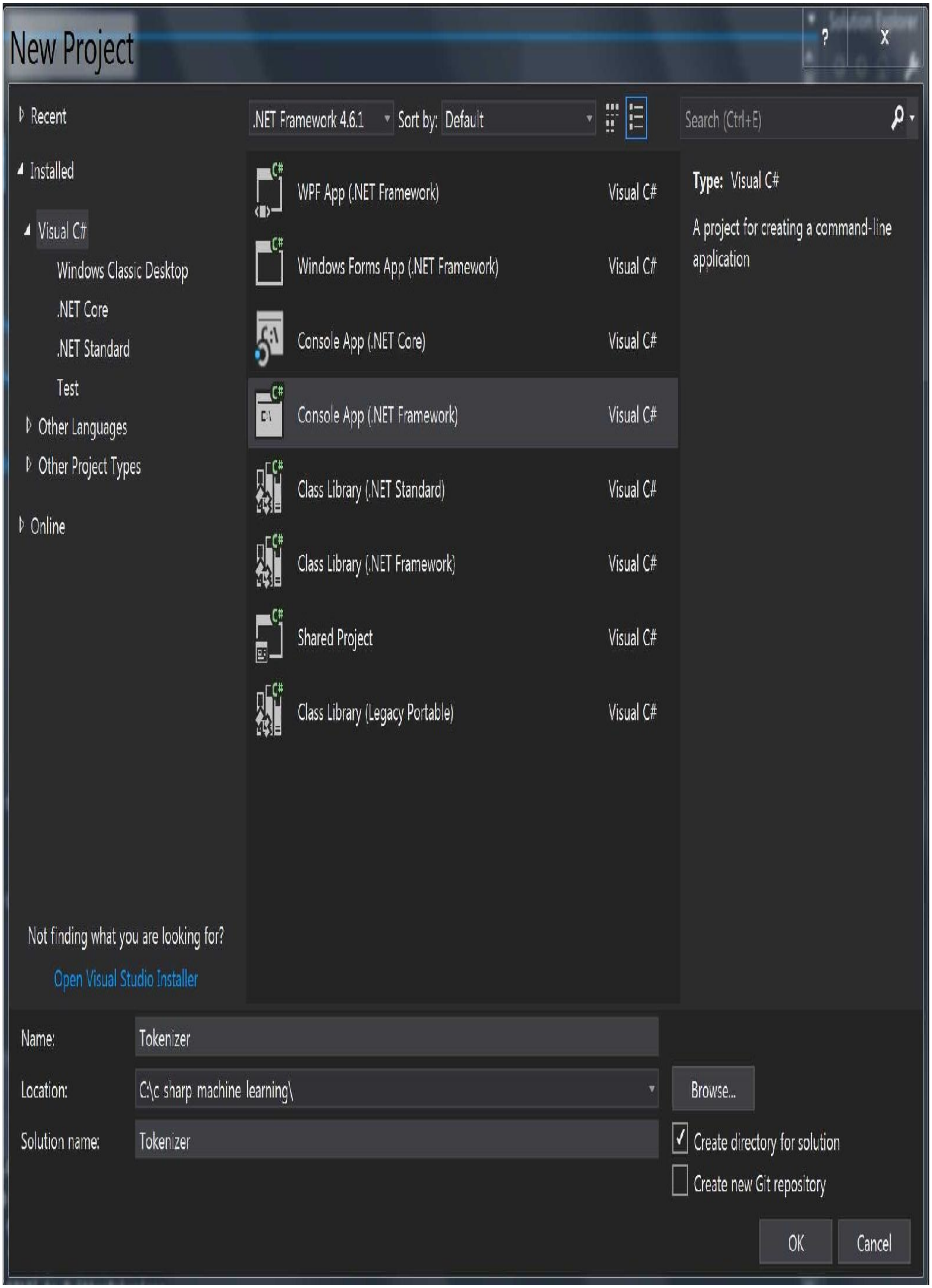
In this chapter, we will cover the following:

- Setting up the environment with the Stanford CoreNLP package
- Problem definition for the Twitter sentiment analysis project
- Data preparation using Stanford CoreNLP
- Data analysis using lemmas as tokens
- Feature engineering using lemmatization and emoticons
- Naive Bayes versus random forest
- Model validations using the ROC curve and AUC metrics

Setting up the environment

Before we dive into our Twitter sentiment analysis project, let's set up our development environment with the Stanford CoreNLP package that we are going to use throughout this chapter. Multiple steps are required to get your environment ready with the Standford CoreNLP package, so it is a good idea to work through this:

1. The first step is to create a new Console App (.NET Framework) project in Visual Studio. Make sure you use a .NET Framework version higher than or equal to 4.6.1. If you have an older version installed, go to `https://docs.microsoft.com/en-us/dotnet/framework/install/guide-for-developers` and follow the installation guide. Following is a screenshot of the project setup page (note that you can select your .NET Framework version in the top bar):



New Project
?
X
Recent
Installed
Visual C#
Windows Classic Desktop
.NET Core
.NET Standard
Test
Other Languages
Other Project Types
Online
Search (Ctrl+E)
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
Not finding what you are looking for?
Open Visual Studio Installer
Browse...
OK
Cancel

.NET Framework 4.6.1

Sort by: Default

Type: Visual C#

A project for creating a command-line application

Name: Tokenizer

Location: C:\c sharp machine learning\

Solution name: Tokenizer

☑ Create directory for solution

□ Create new Git repository

2. Now, let's install the Stanford CoreNLP package. You can type in the following command in your Package Manager Console:

```
Install-Package Stanford.NLP.CoreNLP
```

The version we are going to use in this chapter is `Stanford.NLP.CoreNLP` 3.9.1. Over time, the versions might change and you might have to update your installations.

3. We just have to do one more thing and our environment will be ready to start using the package. We need to install the CoreNLP models JAR, which contains various models for parsing, POS tagging, **Named Entity Recognition** (**NER**), and some other tools. Follow this link to download and unzip Stanford CoreNLP: `https://stanfordnlp.github.io/CoreNLP/`. Once you have downloaded and unzipped it, you will see multiple files in there. The particular file of interest is `stanford-corenlp-<version-number>-models.jar`. We need to extract the contents from that jar file into a directory so that we can load all the model files within our C# project. You can use the following command to extract the contents from `stanford-corenlp-<version-number>-models.jar`:

```
jar xf stanford-corenlp-<version-number>-models.jar
```

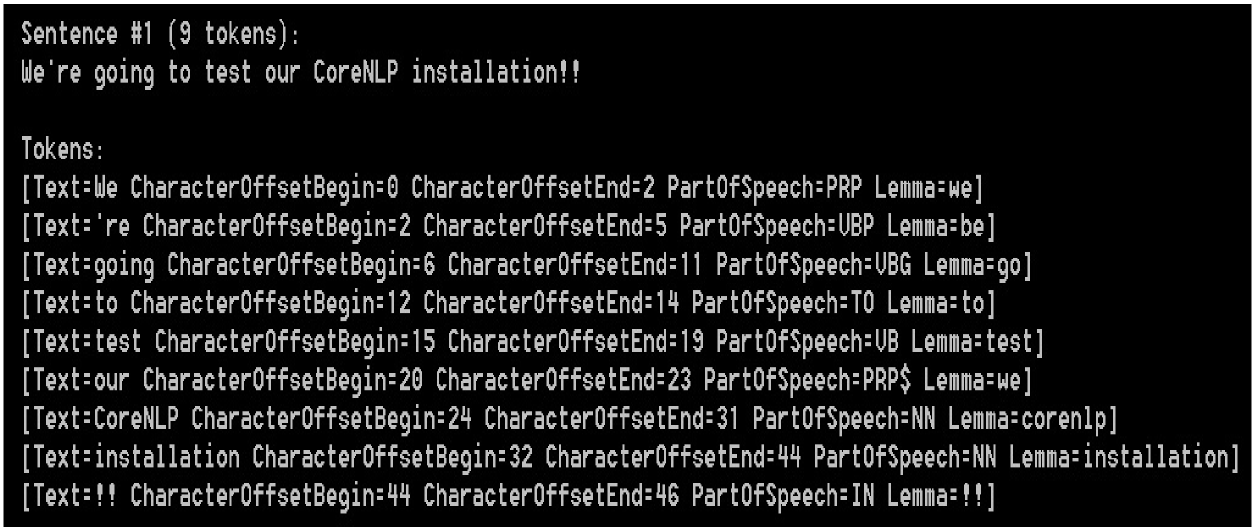
When you are done extracting all the model files from the models jar file, you are now ready to start using the Stanford CoreNLP package in your C# project.

Now, let's check whether our installation was successful. The following code is a slight modification for this example (`https://sergey-tihon.github.io/Stanford.NLP.NET/StanfordCoreNLP.html`) :

```
using System;
using System.IO;
using java.util;
using java.io;
using edu.stanford.nlp.pipeline;
using Console = System.Console;

namespace Tokenizer
{
class Program
{
static void Main()
{
```

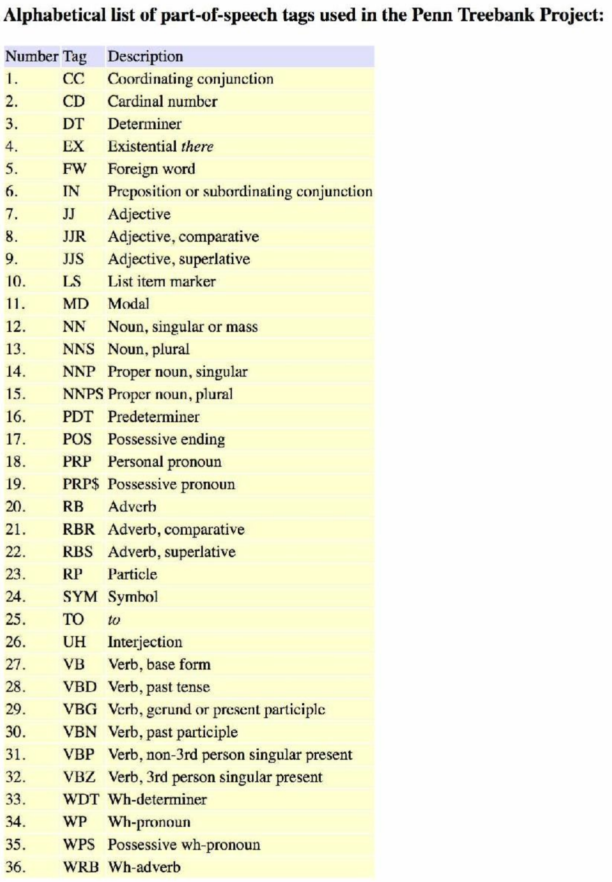
```
// Path to the folder with models extracted from Step #3
var jarRoot = @"<path-to-your-model-files-dir>";

// Text for processing
var text = "We're going to test our CoreNLP installation!!";

// Annotation pipeline configuration
var props = new Properties();
props.setProperty("annotators", "tokenize, ssplit, pos, lemma");
props.setProperty("ner.useSUTime", "0");

// We should change current directory, so StanfordCoreNLP could find all the
model files automatically
var curDir = Environment.CurrentDirectory;
Directory.SetCurrentDirectory(jarRoot);
var pipeline = new StanfordCoreNLP(props);
Directory.SetCurrentDirectory(curDir);

// Annotation
var annotation = new Annotation(text);
pipeline.annotate(annotation);

// Result - Pretty Print
using (var stream = new ByteArrayOutputStream())
{
pipeline.prettyPrint(annotation, new PrintWriter(stream));
Console.WriteLine(stream.toString());
stream.close();
}

Console.ReadKey();
}
}
}
```

If your installation was successful, you should see output similar to the following:



```
Sentence #1 (9 tokens):
We're going to test our CoreNLP installation!!

Tokens:
[Text=We CharacterOffsetBegin=0 CharacterOffsetEnd=2 PartOfSpeech=PRP Lemma=we]
[Text='re CharacterOffsetBegin=2 CharacterOffsetEnd=5 PartOfSpeech=VBP Lemma=be]
[Text=going CharacterOffsetBegin=6 CharacterOffsetEnd=11 PartOfSpeech=VBG Lemma=go]
[Text=to CharacterOffsetBegin=12 CharacterOffsetEnd=14 PartOfSpeech=TO Lemma=to]
[Text=test CharacterOffsetBegin=15 CharacterOffsetEnd=19 PartOfSpeech=VB Lemma=test]
[Text=our CharacterOffsetBegin=20 CharacterOffsetEnd=23 PartOfSpeech=PRP$ Lemma=we]
[Text=CoreNLP CharacterOffsetBegin=24 CharacterOffsetEnd=31 PartOfSpeech=NN Lemma=corenlp]
[Text=installation CharacterOffsetBegin=32 CharacterOffsetEnd=44 PartOfSpeech=NN Lemma=installation]
[Text=!! CharacterOffsetBegin=44 CharacterOffsetEnd=46 PartOfSpeech=IN Lemma=!!]
```

Let's take a closer look at this output. Tokens are character sequences that are grouped as individual semantic units. Often, tokens are *words* or *terms*. In each token output, we can see the original text, such as `We`, `'re`, and `going`. The `PartOfSpeech` tag refers to the category of each word, such as noun, verb, and adjective. For example, the `PartOfSpeech` tag of the first token in our example, `We`, is `PRP` and it stands for *personal pronoun*. The `PartOfSpeech` tag of the second token in our example, `'re`, is `VBP` and it stands for *verb, non-third-person singular present*. The complete list of POS tags can be found here (`http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`) or in the following screenshot:



Alphabetical list of part-of-speech tags used in the Penn Treebank Project:

Number	Tag	Description
1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential *there*
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass
13.	NNS	Noun, plural
14.	NNP	Proper noun, singular
15.	NNPS	Proper noun, plural
16.	PDT	Predeterminer
17.	POS	Possessive ending
18.	PRP	Personal pronoun
19.	PRP$	Possessive pronoun
20.	RB	Adverb
21.	RBR	Adverb, comparative
22.	RBS	Adverb, superlative
23.	RP	Particle
24.	SYM	Symbol
25.	TO	*to*
26.	UH	Interjection
27.	VB	Verb, base form
28.	VBD	Verb, past tense
29.	VBG	Verb, gerund or present participle
30.	VBN	Verb, past participle
31.	VBP	Verb, non-3rd person singular present
32.	VBZ	Verb, 3rd person singular present
33.	WDT	Wh-determiner
34.	WP	Wh-pronoun
35.	WPS	Possessive wh-pronoun
36.	WRB	Wh-adverb

A list of POS tags Lastly, the `Lemma` tag in our tokenization example refers to the standard form of the given word. For example, the lemma of `am` and `are` is `be`. In our example, the word `going` in our third token has `go` as its lemma. We will discuss how we can use word lemmatization for feature engineering in the following sections.

Problem definition for Twitter sentiment analysis

Let's start our Twitter sentiment analysis project by clearly defining what models we will be building and what they are going to predict. You might have heard the term **sentiment analysis** in the past already. Sentiment analysis is essentially a process of computationally determining whether a given text expresses a positive, neutral, or negative emotion. Sentiment analysis for social media content can be used in various ways. For example, it can be used by marketers to identify how effective a marketing campaign was and how it affected consumers' opinions and attitudes towards a certain product or company. Sentiment analysis can also be used to predict stock market changes. Positive news and aggregate positive emotions towards a certain company often move its stock price in a positive direction, and sentiment analysis in the news and social media for a given company can be used to predict how stock prices will move in the near future. To experiment with how we can build a sentiment analysis model, we are going to use a precompiled and labeled airline sentiment Twitter dataset that originally came from CrowdFlower's Data for Everyone library (`https://www.figure-eight.com/data-for-everyone/`). Then, we are going to apply some NLP techniques, especially word tokenization, POS tagging, and lemmatization, to build meaningful text and emoticon features from raw tweet data. Since we want to predict three different emotions (positive, neutral, and negative) for each tweet, we are going to build a multi-class classification model and experiment with different learning algorithms—Naive Bayes and random forest. Once we build the sentiment analysis models, we are going to evaluate the performance mainly via these three metrics: precision, recall, and AUC.

Let's summarize our problem definition for the Twitter sentiment analysis project:

- What is the problem? We need a Twitter sentiment analysis model to computationally identify the emotions in tweets.

- Why is it a problem? Identifying and measuring the emotions of users or consumers about a certain topic, such as a product, company, advertisement, and so forth, are often an essential tool to measure the impact and success of certain tasks.
- What are some of the approaches to solving this problem? We are going to use the Stanford CoreNLP package to apply various NLP techniques, such as tokenization, POS tagging, and lemmatization, to build meaningful features from a raw Twitter dataset. With these features, we are going to experiment with different learning algorithms to build a sentiment analysis model. We will use precision, recall, and AUC measures to evaluate the performance of the models.
- What are the success criteria? We want high precision rates, without sacrificing too much for recall rates, as correctly classifying a tweet into one of three emotion buckets (positive, neutral, and negative) is more important than a higher retrieval rate. Also, we want a high AUC number, which we will discuss in more detail in later sections of this chapter.

Data preparation using Stanford CoreNLP

Now that we know what our goals are in this chapter, it is time to dive into the data. Similar to the last chapter, we are going to use precompiled and pre-labeled Twitter sentiment data. We are going to use a dataset from CrowdFlower's Data for Everyone library (https://www.figure-eight.com/data-for-everyone/) and you can download the data from this link: https://www.kaggle.com/crowdflower/twitter-airline-sentiment. The data we have here is about 15,000 tweets about US airlines. This Twitter data was scraped from February of 2015 and was then labeled into three buckets—positive, negative, and neutral. The link provides you with two types of data: a CSV file and an SQLite database. We are going to work with the CSV file for this project.

Once you have downloaded this data, we need to get it prepared for our future analysis and model building. The two columns of interest in the dataset are airline_sentiment and text. The airline_sentiment column contains information about the sentiment—whether a tweet has positive, negative, or neutral sentiments—and the text column contains the raw Twitter text. To make this raw data readily available for our future data analysis and model building steps, we need to do the following tasks:

- **Clean up unnecessary text**: It's hard to justify some parts of the text as providing many insights and much information for our models to learn from, such as URLs, user IDs, and raw numbers. So, the first step to prepare our raw data is to clean up unnecessary text that does not contain much information. In this example, we removed the URLs, Twitter user IDs, numbers, and hash signs in hashtags. We used Regex to replace such texts with empty strings. The following code illustrates the Regex expressions we used to filter out those texts:

```
// 1. Remove URL's
string urlPattern = @"https?:\/\/\S+\b|www\.(\w+\.)+\S*";
Regex rgx = new Regex(urlPattern);
tweet = rgx.Replace(tweet, "");

// 2. Remove Twitter ID's
string userIDPattern = @"@\w+";
```

```
rgx = new Regex(userIDPattern);
tweet = rgx.Replace(tweet, "");

// 3. Remove Numbers
string numberPattern = @"[-+]?[.\d]*[\d]+[:,.\d]*";
tweet = Regex.Replace(tweet, numberPattern, "");

// 4. Replace Hashtag
string hashtagPattern = @"#";
tweet = Regex.Replace(tweet, hashtagPattern, "");
```

As you can see from this code, there are two ways to replace a string that matches a `Regex` pattern. You can instantiate a `Regex` object and then replace matching strings with the other string, as shown in the first two cases. You can also directly call the static `Regex.Replace` method for the same purpose, as shown in the last two cases. The static method is going to create a `Regex` object each time you call the `Regex.Replace` method, so if you are using the same pattern in multiple places, it will be better to go with the first approach:

- **Group and encode similar emoticons together**: Emoticons, such as smiley faces and sad faces, are frequently used in tweets and provide useful insights about the emotion of each tweet. Intuitively, one user will use smiley face emoticons to tweet about positive events, while another will use sad face emoticons to tweet about negative events. However, different smiley faces show similar positive emotions and can be grouped together. For example, a smiley face with a parenthesis, `:)`, will have the same meaning as another smiley face with a capital letter `D`, `:D`. So, we want to group these similar emoticons together and encode them as one group rather than having them in separate groups. We will use the R code that Romain Paulus and Jeffrey Pennington shared (https://nlp.stanford.edu/projects/glove/preprocess-twitter.rb), translate it into C#, and then apply it to our raw Twitter dataset. The following is how we translated the emoticon `Regex` codes, written in R, into C#, so that we can group and encode similar emoticons together:

```
// 1. Replace Smiley Faces
string smileyFacePattern = String.Format(@"{0}{1}[)dD]+|[)dD]+{1}{0}",
eyesPattern, nosePattern);
tweet = Regex.Replace(tweet, smileyFacePattern, " emo_smiley ");

// 2. Replace LOL Faces
string lolFacePattern = String.Format(@"{0}{1}[pP]+", eyesPattern,
nosePattern);
tweet = Regex.Replace(tweet, lolFacePattern, " emo_lol ");

// 3. Replace Sad Faces
string sadFacePattern = String.Format(@"{0}{1}\(+|\)+{1}{0}", eyesPattern,
nosePattern);
```

```
        tweet = Regex.Replace(tweet, sadFacePattern, " emo_sad ");

        // 4. Replace Neutral Faces
        string neutralFacePattern = String.Format(@"{0}{1}[\/|l*]", eyesPattern,
        nosePattern);
        tweet = Regex.Replace(tweet, neutralFacePattern, " emo_neutral ");

        // 5. Replace Heart
        string heartPattern = "<3";
        tweet = Regex.Replace(tweet, heartPattern, " emo_heart ");
```

- **Group and encode additional helpful expressions together**: Lastly, there are some more expressions that can help our models detect the emotions of tweets. Repeated punctuation, such as `!!!` and `???`, and elongated words, such as `wayyyy` and `soooo`, can provide some extra information about the sentiments of tweets. We will group and encode them separately so that our models can learn from such expressions. The following code shows how we encoded such expressions:

```
        // 1. Replace Punctuation Repeat
        string repeatedPunctuationPattern = @"([!?.]){2,}";
        tweet = Regex.Replace(tweet, repeatedPunctuationPattern, " $1_repeat ");

        // 2. Replace Elongated Words (i.e. wayyyy -> way_emphasized)
        string elongatedWordsPattern = @"\b(\S*?)(.)\2{2,}\b";
        tweet = Regex.Replace(tweet, elongatedWordsPattern, " $1$2_emphasized ");
```

As shown in the code, for repeated punctuation we appended a string with a suffix, `_repeat`. For example, `!!!` will become `!_repeat` and `???` will become `?_repeat`. For elongated words, we appended a string with a suffix, `_emphasized`. For example, `wayyyy` will become `way_emphasized` and `soooo` will become `so_emphasized`.

The full code that takes the raw dataset, processes individual Twitter text as discussed previously, and exports the processed Twitter text into another data file can be found in this repository: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.3/DataProcessor.cs`. Let's briefly walk through the code. It first reads the raw `Tweets.csv` dataset into a Deedle data frame (lines 76–82). Then, it calls a method named `FormatTweets` with a column series that contains all the raw Twitter text. The `FormatTweets` method code in lines 56–65 looks like the following:

```
private static string[] FormatTweets(Series<int, string> rows)
{
    var cleanTweets = rows.GetAllValues().Select((x, i) =>
    {
        string tweet = x.Value;
        return CleanTweet(tweet);
    });

    return cleanTweets.ToArray();
}
```

This `FormatTweets` method iterates through each element in the series, which is the raw tweets, and calls the `CleanTweet` method. Within the `CleanTweet` method, each raw tweet is run against all the `Regex` patterns that we defined previously and is then processed as discussed earlier. The `CleanTweet` method in lines 11–54 looks as follows:

```
private static string CleanTweet(string rawTweet)
{
      string eyesPattern = @"[8:=;]";
      string nosePattern = @"['`\-]?";

      string tweet = rawTweet;
      // 1. Remove URL's
      string urlPattern = @"https?:\/\/\S+\b|www\.(\w+\.)+\S*";
      Regex rgx = new Regex(urlPattern);
      tweet = rgx.Replace(tweet, "");
      // 2. Remove Twitter ID's
      string userIDPattern = @"@\w+";
      rgx = new Regex(userIDPattern);
      tweet = rgx.Replace(tweet, "");
      // 3. Replace Smiley Faces
      string smileyFacePattern = String.Format(@"{0}{1}[)dD]+|[)dD]+{1}{0}",
eyesPattern, nosePattern);
      tweet = Regex.Replace(tweet, smileyFacePattern, " emo_smiley ");
      // 4. Replace LOL Faces
      string lolFacePattern = String.Format(@"{0}{1}[pP]+", eyesPattern, nosePattern);
      tweet = Regex.Replace(tweet, lolFacePattern, " emo_lol ");
      // 5. Replace Sad Faces
      string sadFacePattern = String.Format(@"{0}{1}\(+|\)+{1}{0}", eyesPattern,
nosePattern);
      tweet = Regex.Replace(tweet, sadFacePattern, " emo_sad ");
      // 6. Replace Neutral Faces
      string neutralFacePattern = String.Format(@"{0}{1}[\/|l*]", eyesPattern,
nosePattern);
      tweet = Regex.Replace(tweet, neutralFacePattern, " emo_neutral ");
      // 7. Replace Heart
      string heartPattern = "<3";
      tweet = Regex.Replace(tweet, heartPattern, " emo_heart ");
      // 8. Replace Punctuation Repeat
      string repeatedPunctuationPattern = @"([!?.]){2,}";
      tweet = Regex.Replace(tweet, repeatedPunctuationPattern, " $1_repeat ");
      // 9. Replace Elongated Words (i.e. wayyyy -> way_emphasized)
      string elongatedWordsPattern = @"\b(\S*?)(.)\2{2,}\b";
      tweet = Regex.Replace(tweet, elongatedWordsPattern, " $1$2_emphasized ");
      // 10. Replace Numbers
      string numberPattern = @"[-+]?[.\d]*[\d]+[:,.\d]*";
      tweet = Regex.Replace(tweet, numberPattern, "");
      // 11. Replace Hashtag
      string hashtagPattern = @"#";
      tweet = Regex.Replace(tweet, hashtagPattern, "");

      return tweet;
}
```

Once all the raw Twitter tweets are cleaned and processed, the result gets added to the original Deedle data frame as a separate column with `tweet` as its column name. The following code (line 89) shows how you can add an array of strings to a data frame:

```
rawDF.AddColumn("tweet", processedTweets);
```

Once you have come this far, the only additional step we need to do is export the processed data. Using Deedle data frame's `SaveCsv` method, you can easily export a data frame into a CSV file. The following code shows how we exported the processed data into a CSV file:

```
rawDF.SaveCsv(Path.Combine(dataDirPath, "processed-training.csv"));
```

Now that we have clean Twitter text, let's tokenize and create a matrix representation of tweets. Similar to what we did in Chapter 2, *Spam Email Filtering*, we are going to break a string into words. However, we are going to use the Stanford CoreNLP package that we installed in the previous section of this chapter and utilize the sample code that we wrote in the previous section. The code to tokenize tweets and build a matrix representation of them is as follows:

```
private static Frame<int, string> CreateWordVec(Series<int, string> rows, ISet<string>
stopWords, bool useLemma=false)
        {
            // Path to the folder with models extracted from `stanford-corenlp-
<version>-models.jar`
            var jarRoot = @"<path-to-model-files-dir>";

            // Annotation pipeline configuration
            var props = new Properties();
            props.setProperty("annotators", "tokenize, ssplit, pos, lemma");
            props.setProperty("ner.useSUTime", "0");

            // We should change current directory, so StanfordCoreNLP could find all
the model files automatically
            var curDir = Environment.CurrentDirectory;
            Directory.SetCurrentDirectory(jarRoot);
            var pipeline = new StanfordCoreNLP(props);
            Directory.SetCurrentDirectory(curDir);

            var wordsByRows = rows.GetAllValues().Select((x, i) =>
            {
                var sb = new SeriesBuilder<string, int>();

                // Annotation
                var annotation = new Annotation(x.Value);
                pipeline.annotate(annotation);

                var tokens = annotation.get(typeof(CoreAnnotations.TokensAnnotation));
                ISet<string> terms = new HashSet<string>();

                foreach (CoreLabel token in tokens as ArrayList)
                {
                    string lemma = token.lemma().ToLower();
                    string word = token.word().ToLower();
                    string tag = token.tag();
                    //Console.WriteLine("lemma: {0}, word: {1}, tag: {2}", lemma, word,
tag);

                    // Filter out stop words and single-character words
                    if (!stopWords.Contains(lemma) && word.Length > 1)
                    {
                        if (!useLemma)
                        {
                            terms.Add(word);
                        }
                        else
                        {
                            terms.Add(lemma);
                        }
                    }
                }

                foreach (string term in terms)
                {
                    sb.Add(term, 1);
                }

                return KeyValue.Create(i, sb.Series);
            });

            // Create a data frame from the rows we just created
            // And encode missing values with 0
            var wordVecDF = Frame.FromRows(wordsByRows).FillMissing(0);

            return wordVecDF;
        }
```

As you can see from the code, the main difference between this code and the sample code in the previous section is that this code iterates over each tweet and stores the tokens into a Deedle's data frame. As in `Chapter 2`, *Spam Email Filtering*, we are using one-hot encoding to assign each term's value (0 versus 1) within the matrix. One thing to note here is how we have an option to create the matrix with lemmas or words. Words are the original untouched terms that are broken down from each tweet. For example, a string, `I am a data scientist`, will be broken down into `I`, `am`, `a`, `data`, and `scientist`, if you use words as tokens. Lemmas are standard forms of words in each token. For example, the same string, `I am a data scientist`, will be broken down into `I`, `be`, `a`, `data`, and `scientist`, if you use lemmas as tokens. Note that `be` is a lemma for `am`. We will discuss what lemmas are and what lemmatization is in the *Feature engineering using lemmatization and emoticons* section.

The full code to tokenize and create a matrix representation of tweets can be found here: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.3/TwitterTokenizer.cs`. There are a few things to note in this code. First, let's look at how it counts how many samples we have for each sentiment. The following code snippet (lines 122–127) shows how we computed the number of samples per sentiment:

```
// Look at the sentiment distributions in our sample set
var sampleSetDistribution = rawDF.GetColumn<string>(
    "airline_sentiment"
).GroupBy<string>(x => x.Value).Select(x => x.Value.KeyCount);
sampleSetDistribution.Print();
```

As you can see from this code, we first get the sentiment column, `airline_sentiment`, and group it by the values, where the values can be `neutral`, `negative`, or `positive`. Then, it counts the number of occurrences and returns the count.

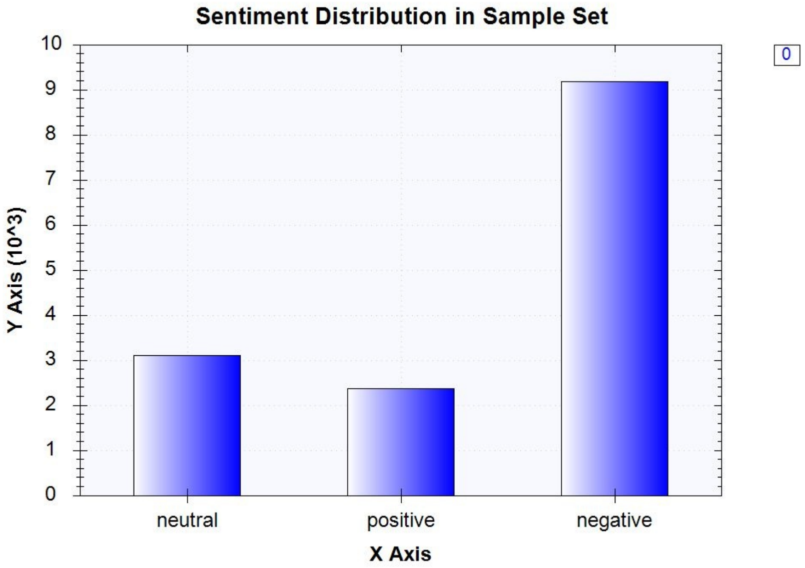
The second thing to note in the TwitterTokenizer code is how we encoded sentiments with integer values. The following is what you see in lines 149–154 of the full code:

```
tweetLemmaVecDF.AddColumn(
    "tweet_polarity",
    rawDF.GetColumn<string>("airline_sentiment").Select(
        x => x.Value == "neutral" ? 0 : x.Value == "positive" ? 1 : 2
    )
);
```

As you can see from this code snippet, we are creating and adding a new column, `tweet_polarity`, to the term matrix data frame. We are taking the values of the `airline_sentiment` column and encoding `0` for neutral, `1` for positive, and `2` for negative. We are going to use this newly added column in our future model building steps.

Lastly, note how we are calling the `CreateWordVec` method twice—once without lemmatization (lines 135-144) and once with lemmatization (lines 147-156). If we create a term matrix with one-hot encodings without lemmatization, we are essentially taking all the words as individual tokens in our term matrix. As you can imagine, this will create a much larger and more sparse matrix than one with lemmatization. We left both codes there for you to explore both options. You can try building ML models with a matrix with words as columns and compare them against those with lemmas as columns. In this chapter, we are going to use the matrix with lemmas instead of words.

When you run this code, it will output a bar chart that shows the sentiment distribution in the sample set. As you can see in the following chart, we have about 3,000 neutral tweets, 2,000 positive tweets, and 9,000 negative tweets in our sample set. The chart looks as follows:



Data analysis using lemmas as tokens

It is now time to look at the actual data and seek any patterns or differences in the distributions of term frequencies along with the different sentiments of tweets. We are going to take the output from the previous step and get the distributions of the top seven most frequently occurring tokens for each sentiment. In this example, we use a term matrix with lemmas. Feel free to run the same analysis for a term matrix with words. The code to analyze the top N most frequently used tokens in each sentiment of tweets can be found here: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.3/DataAnalyzer.cs`.

There is one thing to note in this code. Unlike in the previous chapter, we need to compute term frequencies for three sentiment classes—neutral, negative, and positive. The following is the code snippet from the full code (lines 54-73):

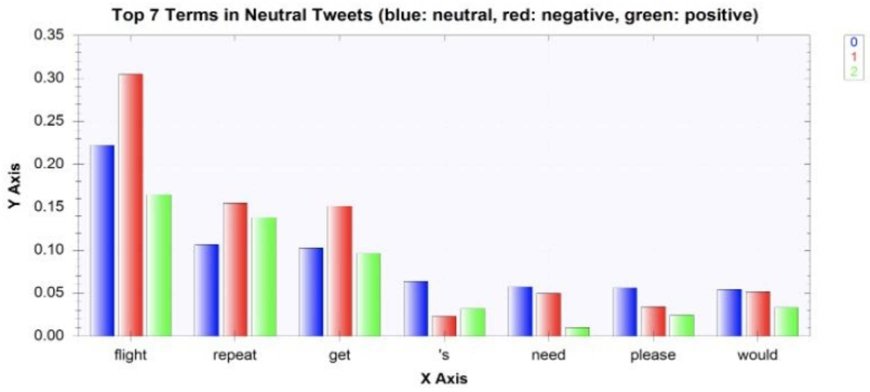
```
var neutralTermFrequencies = ColumnWiseSum(
    tweetLemmaDF.Where(
        x => x.Value.GetAs<int>("tweet_polarity") == 0
    ),
    "tweet_polarity"
).Sort().Reversed;

var positiveTermFrequencies = ColumnWiseSum(
    tweetLemmaDF.Where(
        x => x.Value.GetAs<int>("tweet_polarity") == 1
    ),
    "tweet_polarity"
).Sort().Reversed;

var negativeTermFrequencies = ColumnWiseSum(
    tweetLemmaDF.Where(
        x => x.Value.GetAs<int>("tweet_polarity") == 2
    ),
    "tweet_polarity"
).Sort().Reversed;
```

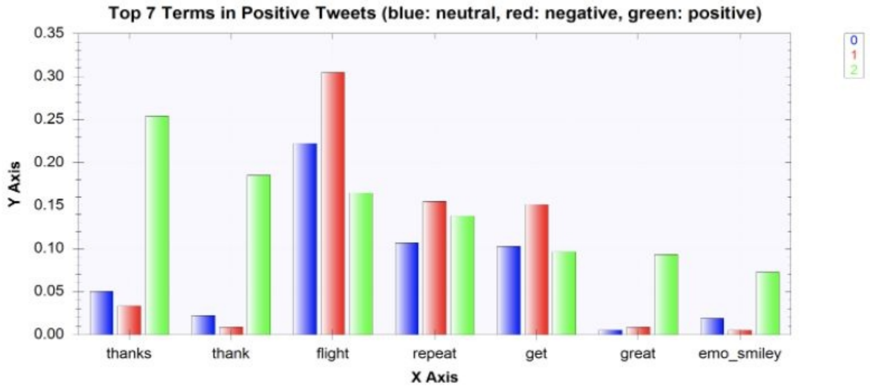
As you can see from the code, we call the `ColumnWiseSum` method for each sentiment class, and the code for this method is as follows:

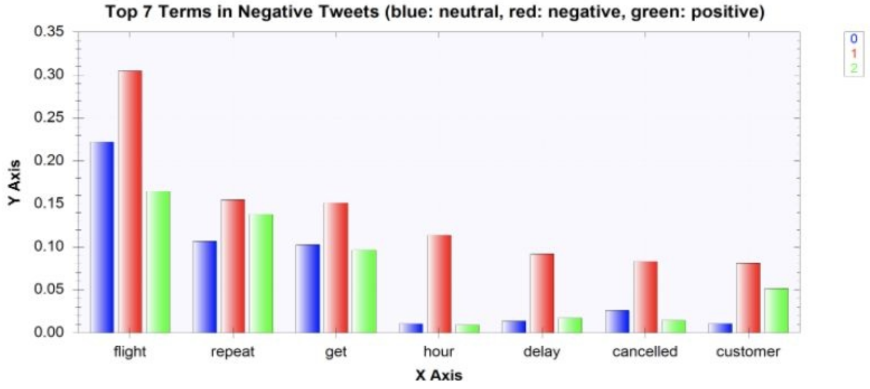
```
private static Series<string, double> ColumnWiseSum(Frame<int, string> frame, string
exclude)
{
    var sb = new SeriesBuilder<string, double>();
    foreach(string colname in frame.ColumnKeys)
    {
        double frequency = frame[colname].Sum();
        if (!colname.Equals(exclude))
        {
            sb.Add(colname, frequency);
        }
    }

    return sb.ToSeries();
}
```

As you can see from this code, it iterates through each column or term and sums all the values within that column. Since we used one-hot encodings, a simple column-wise sum will give us the number of occurrences for each term in our Twitter dataset. Once we have computed all the column-wise summations, we return them as a Deedle series object. With these results, we rank-order the terms by their frequencies and store this information into three separate files, `neutral-frequencies.csv`, `negative-frequencies.csv`, and `positive-frequencies.csv`. We are going to use the term frequency output in later sections for feature engineering and model building.

When you run the code, it will generate the following charts:





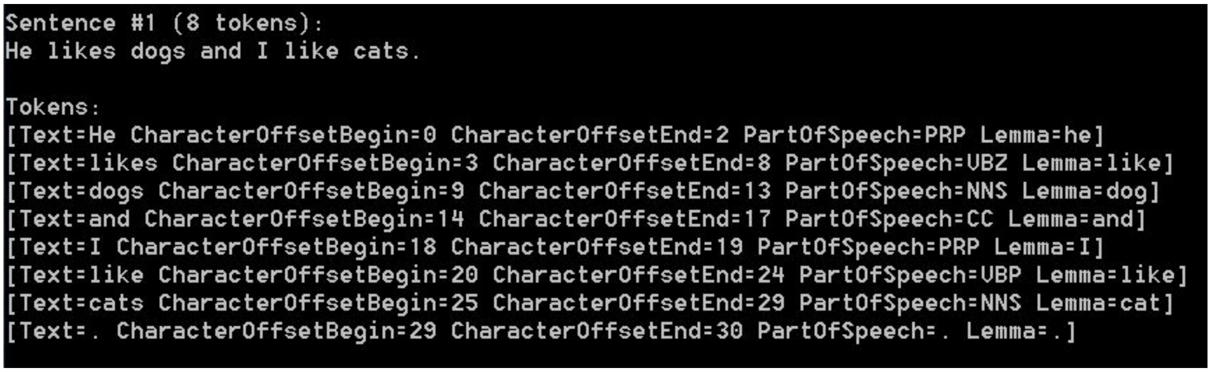


As you can see from the charts, there are some obvious differences in distributions among different sentiments. Words such as **thanks** and **great** were two of the top seven frequently occurring terms in positive tweets, while words like **delay** and **cancelled** were two of the top seven frequently occurring terms in negative tweets. Intuitively, these make sense. You typically use words **thanks** and **great** when you express positive feelings towards someone or something. On the other hand, **delay** and **cancelled** are related to negative events in the context of flights or airlines. Maybe some of the users' flights were delayed or cancelled and they tweeted about their frustrations. Another interesting thing to note is how the term `emo_smiley` was ranked seventh of the most frequently occurring terms in positive tweets. If you remember, in the previous step we grouped and encoded all smiley face emoticons (such as `:)`, `:D`, and so on) as `emo_smiley`. This tells us that emoticons may play an important role for our models to learn how to classify the sentiment of each tweet. Now that we have a rough idea of what our data looks like and what kinds of terminology appear for each sentiment, let's talk about the feature engineering techniques we will employ in this chapter.

Feature engineering using lemmatization and emoticons

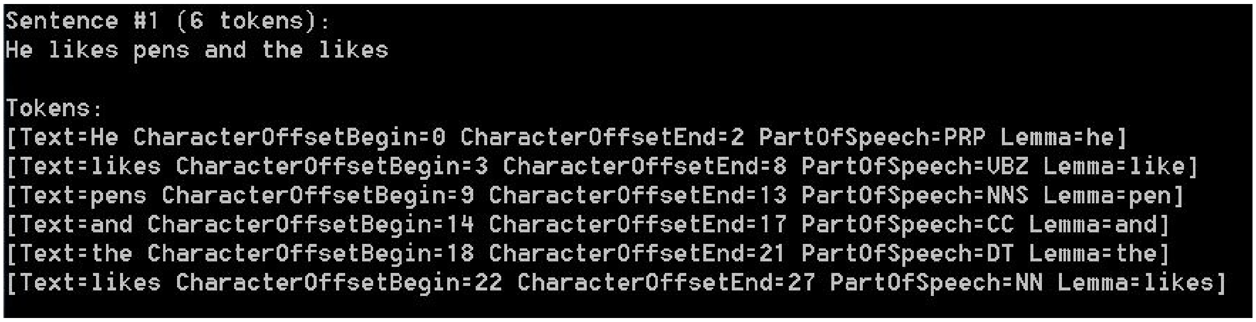
We briefly talked about lemmas in the previous section. Let's take a deeper look at what lemmas are and what lemmatization is. Depending on how and where a word is being used in a sentence, the word is going to be in different forms. For example, the word `like` can take the form of `likes` or `liked` depending on what came before. If we simply tokenize sentences into words, then our program is going to see the words `like`, `likes`, and `liked` as three different tokens. However, that might not be something we want. Those three words share the same meaning and when we are building models, it would be useful to group those words as one token in our feature set. This is what lemmatization does. A lemma is the base form of a word and lemmatization is transforming each word into a lemma based on the part of the sentence each word was used in. In the preceding example, `like` is the lemma for `likes` and `liked`, and systematically transforming `likes` and `liked` into `like` is a lemmatization.

Following is an example of a lemmatization using Stanford CoreNLP:



```
Sentence #1 (8 tokens):
He likes dogs and I like cats.

Tokens:
[Text=He CharacterOffsetBegin=0 CharacterOffsetEnd=2 PartOfSpeech=PRP Lemma=he]
[Text=likes CharacterOffsetBegin=3 CharacterOffsetEnd=8 PartOfSpeech=VBZ Lemma=like]
[Text=dogs CharacterOffsetBegin=9 CharacterOffsetEnd=13 PartOfSpeech=NNS Lemma=dog]
[Text=and CharacterOffsetBegin=14 CharacterOffsetEnd=17 PartOfSpeech=CC Lemma=and]
[Text=I CharacterOffsetBegin=18 CharacterOffsetEnd=19 PartOfSpeech=PRP Lemma=I]
[Text=like CharacterOffsetBegin=20 CharacterOffsetEnd=24 PartOfSpeech=VBP Lemma=like]
[Text=cats CharacterOffsetBegin=25 CharacterOffsetEnd=29 PartOfSpeech=NNS Lemma=cat]
[Text=. CharacterOffsetBegin=29 CharacterOffsetEnd=30 PartOfSpeech=. Lemma=.]
```

Here, you can see that both `likes` and `like` were lemmatized into `like`. This is because both of those words were used as verbs in a sentence and the lemma for the verbal form is `like`. Let's look at another example:



```
Sentence #1 (6 tokens):
He likes pens and the likes

Tokens:
[Text=He CharacterOffsetBegin=0 CharacterOffsetEnd=2 PartOfSpeech=PRP Lemma=he]
[Text=likes CharacterOffsetBegin=3 CharacterOffsetEnd=8 PartOfSpeech=VBZ Lemma=like]
[Text=pens CharacterOffsetBegin=9 CharacterOffsetEnd=13 PartOfSpeech=NNS Lemma=pen]
[Text=and CharacterOffsetBegin=14 CharacterOffsetEnd=17 PartOfSpeech=CC Lemma=and]
[Text=the CharacterOffsetBegin=18 CharacterOffsetEnd=21 PartOfSpeech=DT Lemma=the]
[Text=likes CharacterOffsetBegin=22 CharacterOffsetEnd=27 PartOfSpeech=NN Lemma=likes]
```

Here, the first `likes` and the second `likes` have different lemmas. The first one has `like` as its lemma, while the second one has `likes` as its lemma. This is because the first one is used as a verb, while the second one as a noun. As you can see from these examples, depending on the parts of the sentence, the lemmas for the same words can be different. Using lemmatization for your text dataset can greatly reduce the sparsity and dimensions of your feature space and can help your models learn better without being exposed to too much noise.

Similar to lemmatization, we also grouped similar emoticons into the same group. This is based on the assumption that similar emoticons have similar meanings. For example, `:)` and `:D` have almost the same meanings, if not exactly the same. In another case, depending on the users, the positions of the colon and parenthesis can differ. Some users might type `:)`, but some others might type `(:`. However, the only different between these two is the positioning of the colon and parenthesis and the meanings are the same. In all of these cases, we want our models to learn the same emotion and not create any noise. Grouping similar emoticons into the same group, as we did in the previous step, helps reduce unnecessary noise for our models and help them learn the most from these emoticons.

Naive Bayes versus random forest

It is finally time to train our ML models to predict the sentiments of tweets. In this section, we are going to experiment with Naive Bayes and random forest classifiers. There are two things that we are going to do differently from the previous chapter. First, we are going to split our sample set into a train set and a validation set, instead of running k-fold cross-validation. This is also a frequently used technique, where the models learn only from a subset of the sample set and then they are tested and validated with the rest, which they did not observe at training time. This way, we can test how the models will perform in the unforeseen dataset and simulate how they are going to behave in a real-world case. We are going to use the `SplitSetValidation` class in the Accord.NET package to split our sample set into train and validation sets with pre-defined proportions for each set and fit a learning algorithm to the train set.

Secondly, our target variable is no longer binary (0 or 1), unlike in the previous chapter 2, *Spam Email Filtering*. Instead, it can take any values from 0, 1, or 2, where 0 stands for neutral sentiment tweets, 1 for positive sentiment tweets, and 2 for negative sentiment tweets. So, we are now dealing with a multi-class classification problem, rather than a binary classification problem. We will have to approach things differently when evaluating our models. We will have to modify our accuracy, precision, and recall calculation codes from the previous chapter to compute those numbers for each of the three target sentiment classes in this project. Also, we will have to use a one-versus-rest approach when we look at certain metrics, such as a ROC curve and AUC, which we will be discussing in the following section.

Let's first look at how to instantiate our learning algorithms with the `SplitSetValidation` class in the Accord.NET Framework. The following is how you can instantiate a `SplitSetValidation` object with the Naive Bayes classifier algorithm:

```
var nbSplitSet = new SplitSetValidation<NaiveBayes<BernoulliDistribution>, double[]>()
{
    Learner = (s) => new NaiveBayesLearning<BernoulliDistribution>(),

    Loss = (expected, actual, p) => new ZeroOneLoss(expected).Loss(actual),

    Stratify = false,

    TrainingSetProportion = 0.8,

    ValidationSetProportion = 0.2
};
var nbResult = nbSplitSet.Learn(input, output);
```

As you can see from the preceding code snippet, the major difference between the code you used in the previous chapter and the code shown here is the two parameters that we pass onto a `SplitSetValidation` object—`TrainingSetProportion` and `ValidationSetProportion`. As the name suggests, you can define what percentage of your sample set is should be used for training with the `TrainingSetProportion` parameter and what percentage of your sample set to be used for validation with the `ValidationSetProportion` parameter. Here in our code snippet, we are telling our program to use 80% of our sample for training and 20% for validation. In the last line of the code snippet, we fit a Naive Bayes classification model to the train set that was split from the sample set. Also, note here that we used `BernoulliDistribution` for our Naive Bayes classifier, as we used one-hot encoding to encode our features and all of our features have binary values, similar to what we did in the previous chapter.

Similar to how we instantiated a `SplitSetValidation` object with the Naive Bayes classifier, you can instantiate another one with the random forest classifier as in the following:

```
var rfSplitSet = new SplitSetValidation<RandomForest, double[]>()
{
    Learner = (s) => new RandomForestLearning()
    {
        NumberOfTrees = 100, // Change this hyperparameter for further tuning

        CoverageRatio = 0.5, // the proportion of variables that can be used at maximum
by each tree

        SampleRatio = 0.7 // the proportion of samples used to train each of the trees

    },

    Loss = (expected, actual, p) => new ZeroOneLoss(expected).Loss(actual),

    Stratify = false,

    TrainingSetProportion = 0.7,

    ValidationSetProportion = 0.3
};
var rfResult = rfSplitSet.Learn(input, output);
```

We replaced the previous code with random forest as a model and

`RandomForestLearning` as a learning algorithm. If you look closely, there are some hyperparameters that we can tune for `RandomForestLearning`. The first one is `NumberOfTrees`. This hyperparameter lets you choose the number of decision trees that go into your random forest. In general, having more trees in a random forest results in better performance, as you are essentially building more decision trees in the forest. However, the performance lift comes at the cost of training and prediction time. It will take more time to train and make predictions as you increase the number of trees in your random forest. The other two parameters to note here are `CoverageRatio` and `SampleRatio`. `CoverageRatio` sets the proportion of the feature set to be used in each tree, while `SampleRatio` sets the proportion of the train set to be used in each tree. Having a higher `CoverageRatio` and `SampleRatio` increases the performance of individual trees in the forest, but it also increases the correlation among the trees. Lower correlation among the trees helps reduce the generalization error; thus, finding a good balance between the prediction powers of individual trees and correlation among the trees will be essential in building a good random forest model. Tuning and experimenting with various combinations of these hyperparameters can help you avoid overfitting issues, as well as improving your model performance when training a random forest model. We recommend you build a number of random forest classifiers with various combinations of those hyperparameters and experiment with their effects on your model performances.

The full code that we used to train Naive Bayes and random forest classification models and output validation results can be found here: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.3/TwitterSentimentModeling.cs. Let's take a closer look at this code. In lines 36-41, it first reads in the token matrix file, `tweet-lemma.csv`, which we built in the data preparation step. Then in lines 43-51, we read in the term frequency files, `positive-frequencies.csv` and `negative-frequencies.csv`, which we built in the data analysis step. Similar to what we did in the previous chapter, we do feature selection based on the number of term occurrences in line 64. In this example, we experimented with 5, 10, 50, 100, and 150 as the thresholds for the minimum number of term occurrences in our sample tweets. From line 65, we iterate through those thresholds and start training and evaluating Naive Bayes and random forest classifiers. Each time a model is trained on a train set, it is then run against the validation set that was not observed during the training time.

Following is part of the full code (lines 113-135) that runs the trained Naive Bayes model on the train and validation sets to measure in-sample and out-of-sample performance:

```
// Get in-sample & out-sample prediction results for NaiveBayes Classifier
var nbTrainedModel = nbResult.Model;

int[] nbTrainSetIDX = nbSplitSet.IndicesTrainingSet;
int[] nbTestSetIDX = nbSplitSet.IndicesValidationSet;

Console.WriteLine("* Train Set Size: {0}, Test Set Size: {1}", nbTrainSetIDX.Length,
nbTestSetIDX.Length);

int[] nbTrainPreds = new int[nbTrainSetIDX.Length];
int[] nbTrainActual = new int[nbTrainSetIDX.Length];
for (int i = 0; i < nbTrainPreds.Length; i++)
{
   nbTrainActual[i] = output[nbTrainSetIDX[i]];
   nbTrainPreds[i] = nbTrainedModel.Decide(input[nbTrainSetIDX[i]]);
}

int[] nbTestPreds = new int[nbTestSetIDX.Length];
int[] nbTestActual = new int[nbTestSetIDX.Length];
for (int i = 0; i < nbTestPreds.Length; i++)
{
   nbTestActual[i] = output[nbTestSetIDX[i]];
   nbTestPreds[i] = nbTrainedModel.Decide(input[nbTestSetIDX[i]]);
}
```

Following is the part of the full code (lines 167-189) that runs the trained random forest model on the train and validation sets to measure in-sample and out-of-sample performance:

```
// Get in-sample & out-sample prediction results for RandomForest Classifier
var rfTrainedModel = rfResult.Model;

int[] rfTrainSetIDX = rfSplitSet.IndicesTrainingSet;
int[] rfTestSetIDX = rfSplitSet.IndicesValidationSet;

Console.WriteLine("* Train Set Size: {0}, Test Set Size: {1}", rfTrainSetIDX.Length,
rfTestSetIDX.Length);

int[] rfTrainPreds = new int[rfTrainSetIDX.Length];
int[] rfTrainActual = new int[rfTrainSetIDX.Length];
for (int i = 0; i < rfTrainPreds.Length; i++)
{
    rfTrainActual[i] = output[rfTrainSetIDX[i]];
    rfTrainPreds[i] = rfTrainedModel.Decide(input[rfTrainSetIDX[i]]);
}

int[] rfTestPreds = new int[rfTestSetIDX.Length];
int[] rfTestActual = new int[rfTestSetIDX.Length];
for (int i = 0; i < rfTestPreds.Length; i++)
{
    rfTestActual[i] = output[rfTestSetIDX[i]];
    rfTestPreds[i] = rfTrainedModel.Decide(input[rfTestSetIDX[i]]);
}
```

Let's take a closer look at these. For brevity, we will only take a look at the random forest model case, as it will be the same for the Naive Bayes classifier. In line 168, we first get the trained model from the learned results. Then, we get the indexes of in-sample (train set) and out-of-sample (test/validation set) sets from the `SplitSetValidation` object in lines 170-171, so that we can iterate through each row or record and make predictions. We iterate this process twice—once for the in-sample training set in lines 175-181 and again for the out-of-sample validation set in lines 183-189.

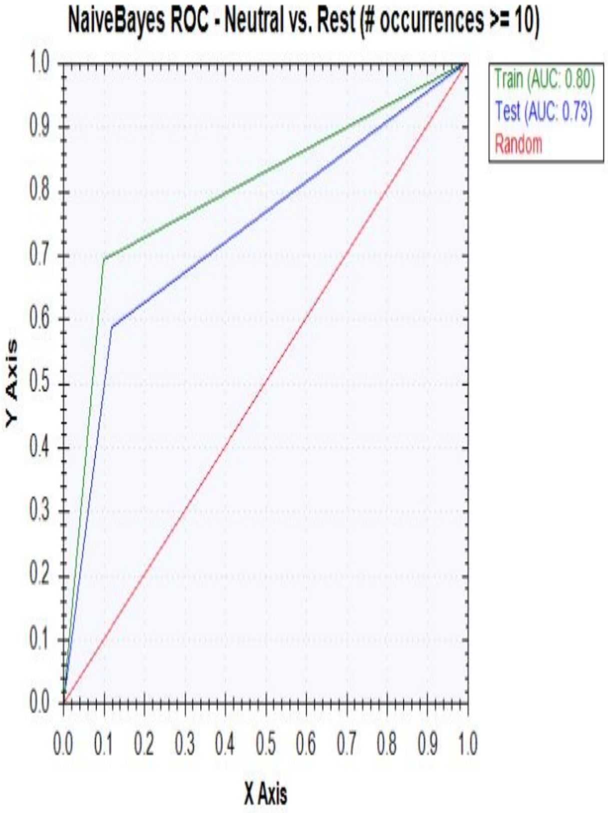
Once we have the prediction results on the train and test sets, we run those results through some validation methods (lines 138-141 for the Naive Bayes classifier and lines 192-196 for the random forest classifier). There are two methods that we wrote specifically for the model validation for this project —`PrintConfusionMatrix` and `DrawROCCurve`. `PrintConfusionMatrix` is an updated version of what we had in `Chapter 2`, *Spam Email Filtering*, where it now prints a 3 x 3 confusion matrix, instead of a 2 x 2 confusion matrix. On the other hand, the `DrawROCCurve` method brings in some new concepts and new model validation methods for this project. Let's discuss those new evaluation metrics, which we are using for this project, in greater detail in the following section.

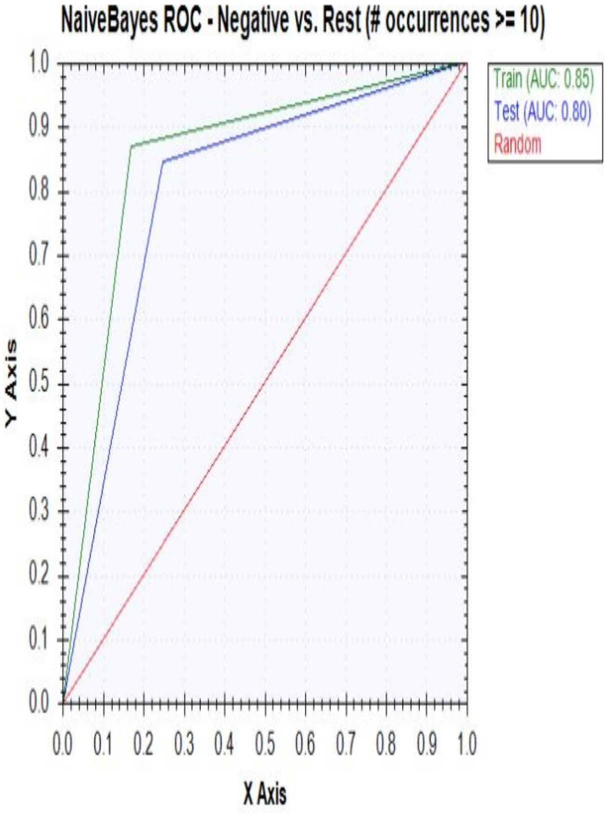
Model validations – ROC curve and AUC

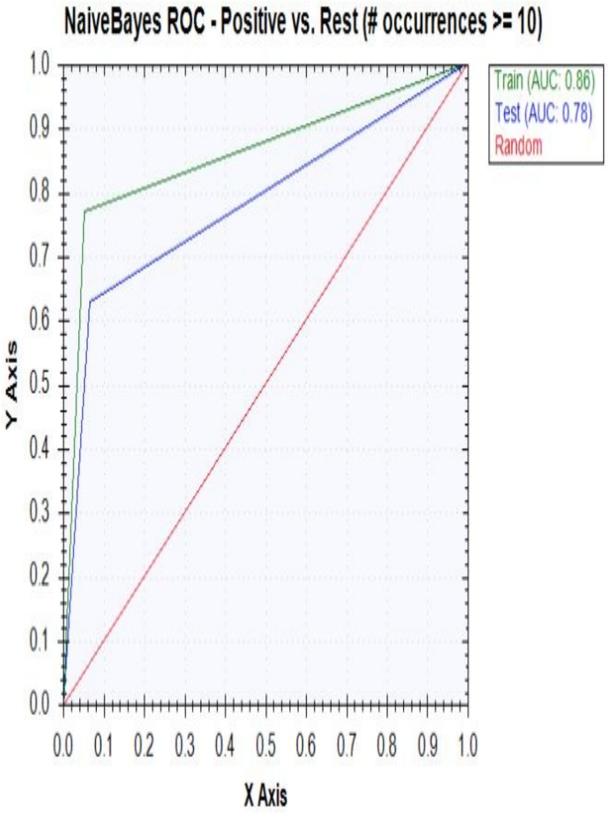
As mentioned before, we are using different model validation metrics in this chapter: the ROC curve and AUC. The ROC curve is a plot of a true positive rate against a false positive rate at various thresholds. Each point in the curve represents the true positive and false positive rate pair corresponding at a certain probability threshold. It is commonly used to select the best and the most optimal models among different model candidates.

The area under the ROC curve (AUC) measures how well the model can distinguish the two classes. In the case of a binary classification, AUC measures how well a model distinguishes the positive outcomes from the negative outcomes. Since we are dealing with a multi-class classification problem in this project, we are using a one-versus-rest approach to build the ROC curve and compute the AUC. For example, one ROC curve can take positive tweets as positive outcomes and neutral and negative tweets as negative outcomes, while another ROC curve can take neutral tweets as positive outcomes and positive and negative tweets as negative outcomes. As shown in the following charts, we drew three ROC charts for each model we built—one for Neutral verses Rest (Positive and Negative), one for Positive versus Rest (Neutral and Negative), and one for Negative versus Rest (Neutral and Positive). The higher the AUC number is, the better the model is as it suggests that the model can distinguish positive classes from negative classes with much better chance.

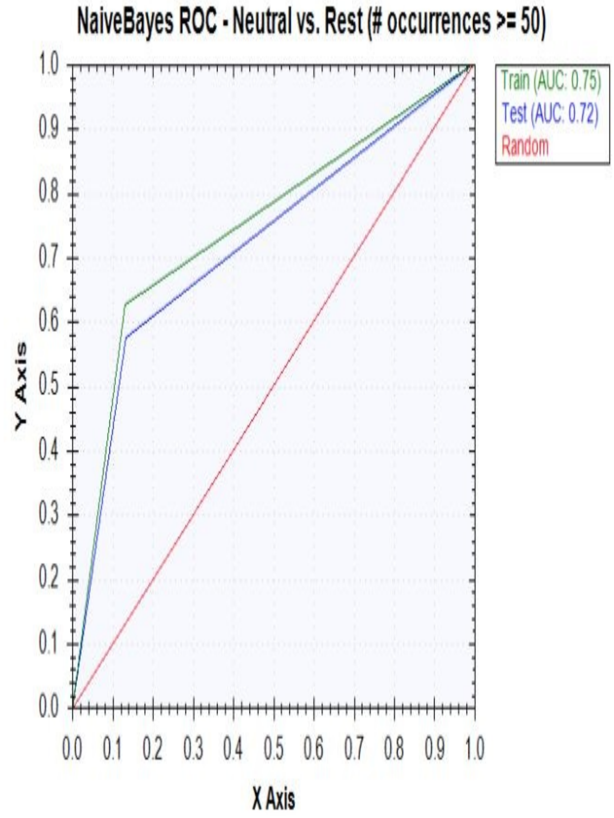
The following charts show ROC curves for Naive Bayes classifiers with the minimum number of term occurrences at **10**:

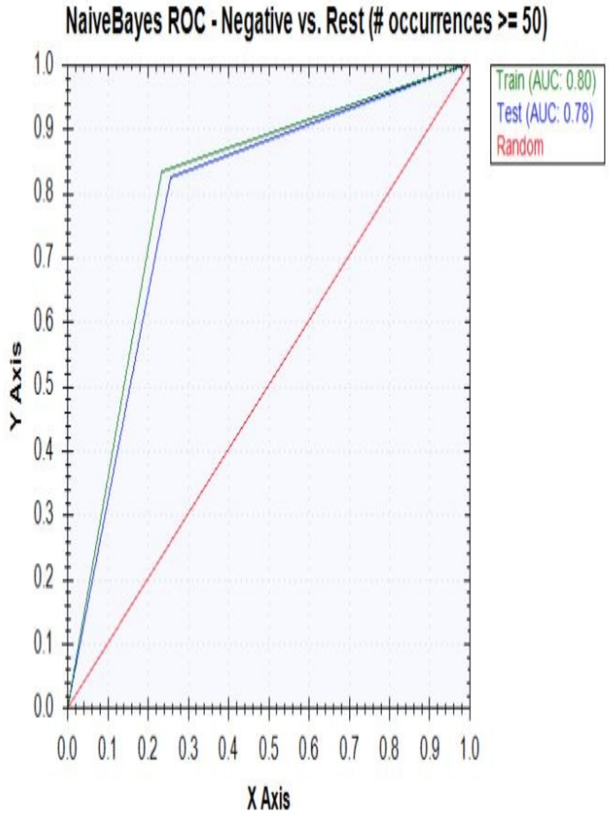

NaiveBayes ROC - Neutral vs. Rest (# occurrences >= 10)
Train (AUC: 0.80)
Test (AUC: 0.73)
Random
Y Axis
X Axis
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

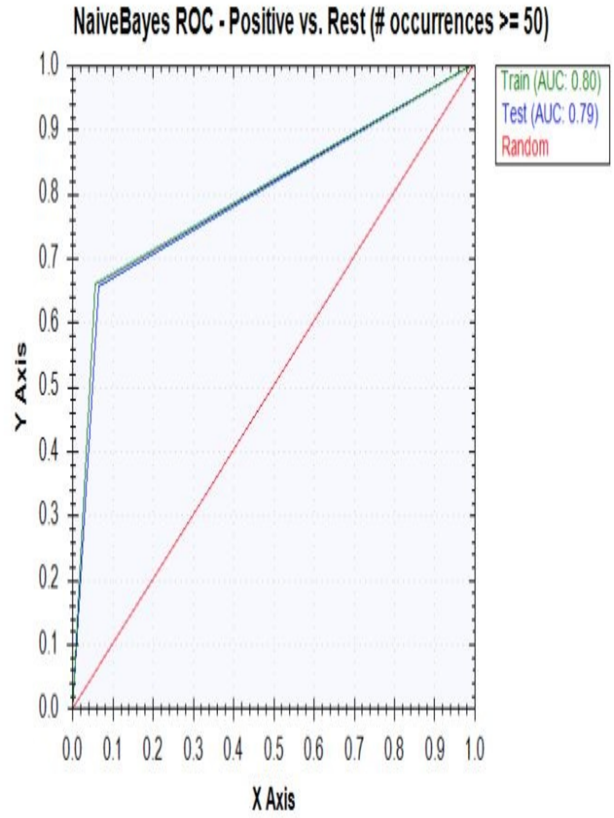

NaiveBayes ROC - Negative vs. Rest (# occurrences >= 10)
Train (AUC: 0.85)
Test (AUC: 0.80)
Random
Y Axis
X Axis
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


NaiveBayes ROC - Positive vs. Rest (# occurrences >= 10)
Train (AUC: 0.86)
Test (AUC: 0.78)
Random
Y Axis
X Axis
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

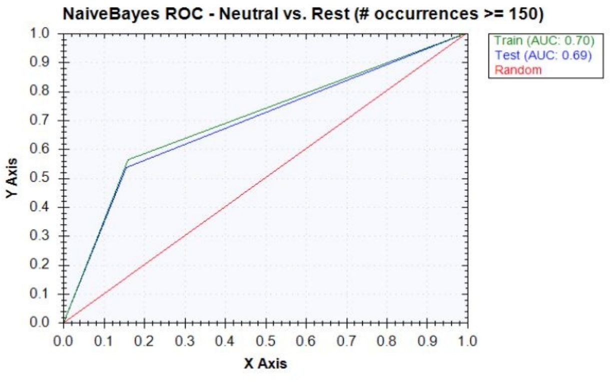
The following charts show ROC curves for Naive Bayes classifiers with the minimum number of term occurrences at **50**:

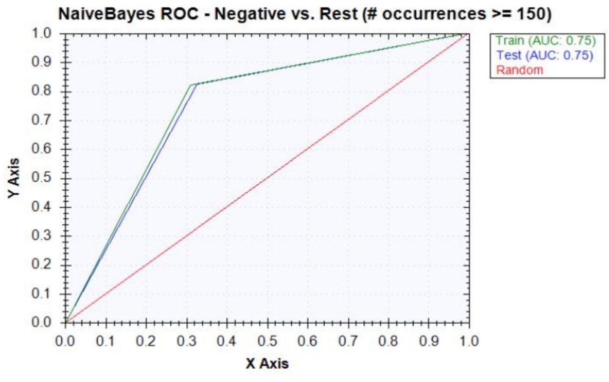

NaiveBayes ROC - Neutral vs. Rest (# occurrences >= 50)

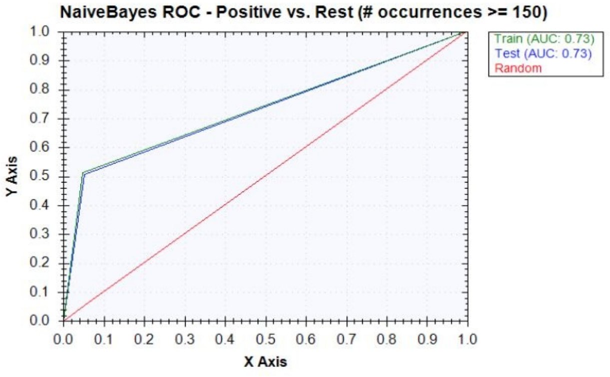

NaiveBayes ROC - Negative vs. Rest (# occurrences >= 50)


NaiveBayes ROC - Positive vs. Rest (# occurrences >= 50)

The following charts show ROC curves for Naive Bayes classifiers with the minimum number of term occurrences at **150**:

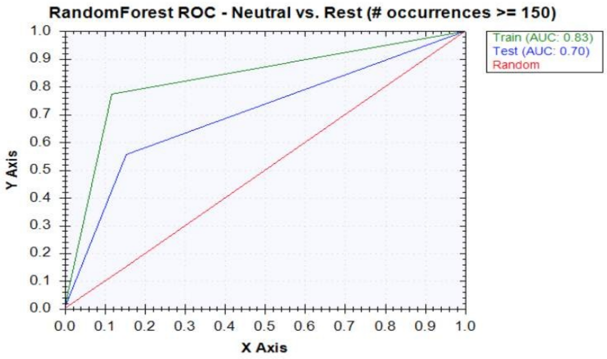


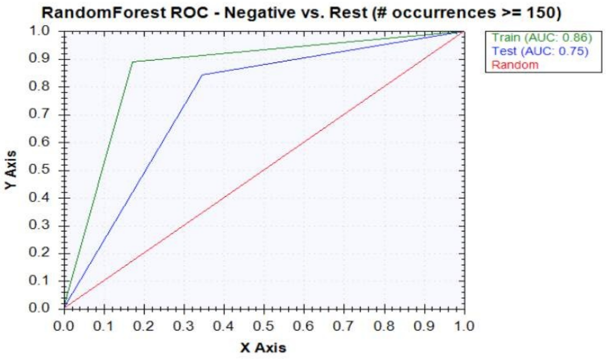


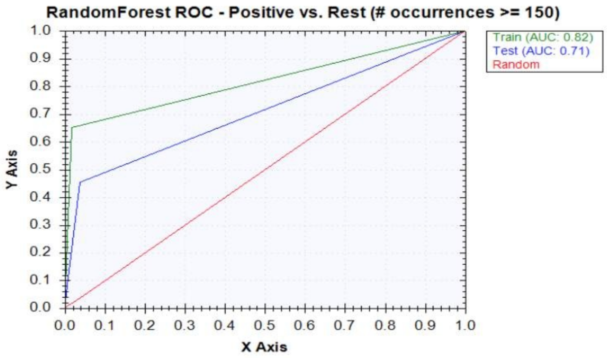


As you can see from the charts, we can also detect overfitting issues from ROC charts by looking at the gaps between the curves from training and testing results. The larger the gap is, the more the model is overfitting. If you look at the first case, where we only filter out those terms that appear in tweets fewer than ten times, the gap between the two curves is large. As we increase the threshold, we can see that the gap decreases. When we are choosing the final model, we want the train ROC curve and the test/validation ROC curve to be as small as possible. As this resolution comes at the expense of the model performance, we need to find the right cutoff line for this trade-off.

Let's now look at a sample of how one of our random forest classifiers did. The following is a sample result from fitting a random forest classifier:







Ensemble methods, such as random forest, generally work well for classification problems and accuracy can be improved by ensembling with more trees. However, they come with some limitations, one of which is shown in the previous sample results for the random forest classifier. As is true for all decision tree-based models, the random forest model tends to overfit, especially when it tries to learn from many categorical variables. As you can see from the ROC curves for the random forest classifier, the gap between the train and test ROC curves is large, especially when compared to those for the Naive Bayes classifier. The Naive Bayes classifier with a minimum number of term occurrences threshold at 150 has almost no gap between the train and test ROC curves, whereas a random forest classifier at the same threshold shows a large gap between the two ROC curves. When dealing with such a dataset, where there are lots of categorical variables, we need to be careful about which model to choose and pay special attention to tuning the hyperparameters, such as `NumberOfTrees`, `CoverageRatio`, and `SampleRatio`, for a random forest model.

Summary

In this chapter, we built and trained more advanced classification models for Twitter sentiment analysis. We applied what we have learned in the previous chapter to a multi-class classification problem with more complex text data. We first started off by setting up our environment with the Stanford CoreNLP package that we used for tokenization, POS tagging, and lemmatization in the data preparation and analysis steps. Then, we transformed the raw Twitter dataset into a one-hot encoded matrix by tokenizing and lemmatizing the tweets. During this data preparation step, we also discussed how we could use Regex to group similar emoticons together and remove unnecessary text, such as URLs, Twitter IDs, and raw numbers, from tweets. We further analyzed the distribution of frequently used terms and emoticons in our data analysis step and we saw how lemmatization and grouping similar emoticons together help in reducing unnecessary noise in the dataset. With data and insights from previous steps, we experimented with building multi-class classification models using Naive Bayes and random forest classifiers. As we built these models, we covered a frequently used model validation technique, where we split a sample set into two subsets, the train set and validation set, and used the train set to fit a model and the validation set to evaluate the model performance. We also covered new model validation metrics, the ROC curve and AUC, which we can use to select the best and most optimal model among model candidates.

In the next chapter, we are going to switch gears and start building regression models where the target variables are continuous variables. We will use a foreign exchange rate dataset to build time series features and explore some other ML models for regression problems. We will also discuss how evaluating the performance of regression models is different from that of classification models.

Foreign Exchange Rate Forecast

In this chapter, we are going to start building regression models in C#. Up until now, we have built **machine learning** (**ML**) models with the goal of classifying data into binary or multiple buckets using logistic regression, Naive Bayes, and random forest learning algorithms. However, we are now going to switch gears and start building models that predict continuous outcomes. In this chapter, we will explore a financial dataset, more specifically a Foreign Exchange Rate market dataset. We will be using historical data of daily currency exchange rates between Euros (EUR) and U.S. Dollars (USD) to build a regression model that forecasts future exchange rates. We are going to start with the problem definition and then move on to data preparation and data analysis. During the data preparation and analysis steps, we are going to explore how we can manage the time series data and analyze the distributions of daily returns. Then, we are going to start building features that can forecast currency exchange rates in the feature engineering step. We are going to discuss a few commonly used technical indicators in the financial market, such as moving averages, Bollinger Bands, and lagging variables. Using those technical indicators, we will be building regression ML models using linear regression and **Support Vector Machine** (**SVM**) learning algorithms. While building such models, we will also explore some of the ways we can fine-tune hyperparameters for the SVM model. Lastly, we will discuss a few validation metrics and methods for evaluating our regression models. We will discuss how we can use **root mean square error** (**RMSE**), R^2, and an observed versus fitted values plot to evaluate the performances of our models. By the end of this chapter, you will have working regression models for forecasting daily EUR/USD exchange rates.

In this chapter, we will cover the following steps:

- Problem definition for the Foreign Exchange Rate (EUR versus USD) forecast project
- Data preparation using time-series functionalities in the Deedle framework
- Time series data analysis

- Feature engineering using various technical indicators in Forex
- Linear regression versus SVM
- Model validations using RMSE, R^2, and the actual versus predicted plot

Problem definition

Let's start this chapter by defining what we are trying to solve in this project. You might have heard the terms *Algorithmic Trading* or *Quantitative Finance/Trading*. This is one of the well-known fields in the finance industry where data science and ML meet finance. Algorithmic Trading or Quantitative Finance refers to a strategy where you use statistical learning models that were built from a large amount of historical data to forecast future financial market movements. Such strategies and techniques are used widely by various traders and investors forecast future prices for various financial assets. The foreign exchange market is one of the largest and most liquid financial markets, and a large pool of traders and investors take part in it. It is a unique market that is open 24 hours a day and five days a week and traders from all over the world come in to buy and sell certain currency pairs. Due to this advantage and uniqueness, the foreign exchange market is also an attractive financial market for algorithmic and quantitative traders to build ML models to forecast future exchange rates and automate their trades to take advantage of the fast decisions and executions that computers can make.

To get a feel for how we can apply our ML knowledge to financial markets and regression models, we are going to use historical data of daily EUR/USD rates from January 1, 1999 to December 31st, 2017. We are going to use a publicly available dataset, which can be downloaded from this link: `http://www.global-view.com/forex-trading-tools/forex-history/index.html`. With this data, we are going to build features by using commonly used technical indicators, such as moving averages, Bollinger Bands, and lagging variables. Then, we are going to build regression models using linear regression and SVM learning algorithms that forecast future daily exchange rates for EUR/USD currency pairs. Once we have built these models, we are going to use RMSE, R^2, and a plot of observed values against predicted values to evaluate our models.

To summarize our problem definition for the foreign exchange rate forecasting project:

- What is the problem? We need a regression model that forecasts future foreign exchange rates between Euros and U.S. Dollars; more specifically, we want to build a ML model that forecasts daily changes in EUR/USD exchange rates.
- Why is it a problem? Due to the fast-paced and volatile environments in the foreign exchange market, it is advantageous to have a ML model that can forecast and make autonomous decisions on when to buy and when to sell certain currency pairs.
- What are some of the approaches to solving this problem? We are going to use historical data of daily exchange rates between EUR and USD. With this dataset, we are going to build financial features using often used technical indicators, such as moving averages, Bollinger Bands, and lagging variables. We will explore linear regression and SVM learning algorithms as our candidates for the regression model. Then, we will look at RMSE, R^2, and use an observed versus predicted plot to evaluate the performances of the models that we built.
- What are the success criteria? We want low RMSE, as we want our predictions to be as close to the actual values as possible. We want high R^2, as it indicates the goodness of fit for our models. Lastly, we would like to see data points lined up closely to a diagonal line in the observed versus predicted plot.

Data preparation

Now that we know what kind of problem we are trying to solve in this chapter, let's start looking into the data. Unlike the two previous chapters, where we precompiled and prelabeled data, we are going to start with raw EUR/USD exchange rate data. Follow this link: `http://www.global-view.com/forex-trading-tools/forex-history/index.html` and select **EUR/USD Close**, **EUR/USD High**, and **EUR/USD Low**. You can also select different currency pairs, if you'd like to explore different datasets. Once you have selected the data points you want, you can then select the start and end dates and you can also choose whether you want to download daily, weekly, or monthly data. For this chapter, we choose **01/01/1999** as the **Start Date** and **12/31/2017** as the **Stop Date** and we download the daily dataset that contains close, high, and low prices for the EUR/USD currency pair.

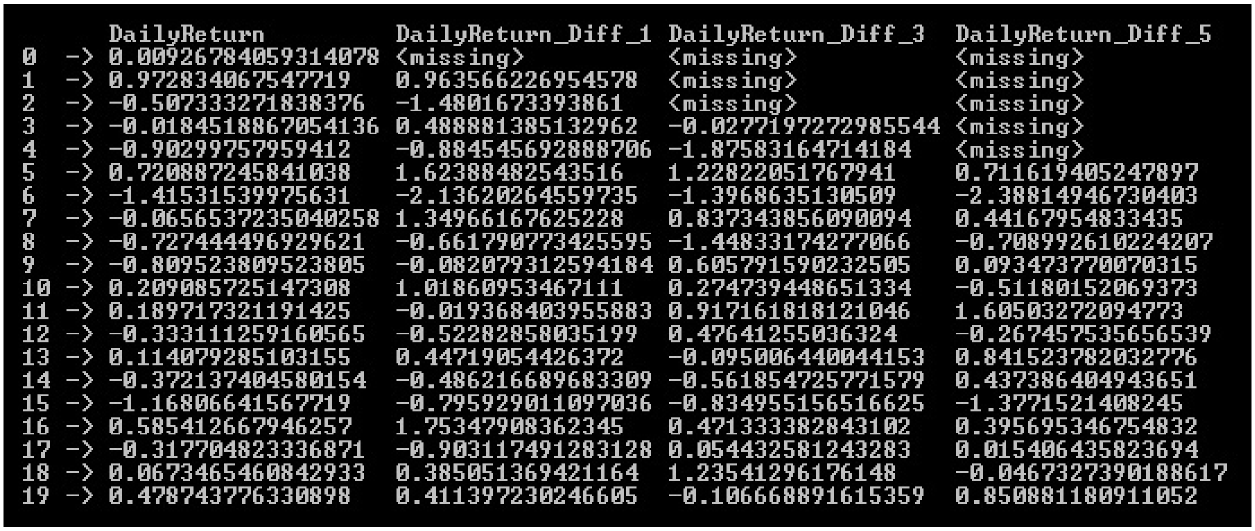
Once you have downloaded the data, there are a few tasks we need to do to get it ready for our future data analysis, feature engineering, and ML modeling. First, we need to define target variables. As discussed in our problem definition step, our target variable is going to be the daily change in EUR/USD exchange rates. To compute daily returns, we need to subtract the previous day's close price from today's close price and then divide it by previous day's close price. The formula for calculating daily returns is as follows:

$$Daily\ Return = \frac{ClosePrice^{T} - ClosePrice^{T-1}}{ClosePrice^{T-1}}$$

We can use the `Diff` method in Deedle's data frame to calculate the difference between the previous price and the current price. You can actually use the `Diff` method to calculate the difference between a data point at any arbitrary point in time and the current data point. For example, the following code shows how you can calculate the differences between the current data point and the data points at one step ahead, three steps ahead, and five steps ahead:

```
rawDF["DailyReturn"].Diff(1)
rawDF["DailyReturn"].Diff(3)
rawDF["DailyReturn"].Diff(5)
```

The output of the preceding code is as follows:



```
      DailyReturn         DailyReturn_Diff_1 DailyReturn_Diff_3  DailyReturn_Diff_5
0  -> 0.00926784059314078 <missing>          <missing>           <missing>
1  -> 0.972834067547719   0.963566226954578  <missing>           <missing>
2  -> -0.507333271838376  -1.4801673393861   <missing>           <missing>
3  -> -0.0184518867054136 0.488881385132962  -0.0277197272985544 <missing>
4  -> -0.90299757959412   -0.884545692888706 -1.87583164714184   <missing>
5  -> 0.720887245841038   1.62388482543516   1.22822051767941    0.711619405247897
6  -> -1.41531539975631   -2.13620264559735  -1.3968635130509    -2.38814946730403
7  -> -0.0656537235040258 1.34966167625228   0.837343856090094   0.44167954833435
8  -> -0.727444496929621  -0.661790773425595 -1.44833174277066   -0.708992610224207
9  -> -0.809523809523805  -0.082079312594184 0.605791590232505   0.093473770070315
10 -> 0.209085725147308   1.01860953467111   0.274739448651334   -0.51180152069373
11 -> 0.189717321191425   -0.019368403955883 0.917161818121046   1.60503272094773
12 -> -0.333111259160565  -0.52282858035199  0.47641255036324    -0.267457535656539
13 -> 0.114079285103155   0.44719054426372   -0.095006440044153  0.841523782032776
14 -> -0.372137404580154  -0.486216689683309 -0.561854725771579  0.437386404943651
15 -> -1.16806641567719   -0.795929011097036 -0.834955156516625  -1.3771521408245
16 -> 0.585412667946257   1.75347908362345   0.471333382843102   0.395695346754832
17 -> -0.317704823336871  -0.903117491283128 0.054432581243283   0.015406435823694
18 -> 0.0673465460842933  0.385051369421164  1.23541296176148    -0.0467327390188617
19 -> 0.478743776330898   0.411397230246605  -0.106668891615359  0.850881180911052
```

Using this `Diff` method, the following code is how we can calculate the daily returns of EUR/USD exchange rates:

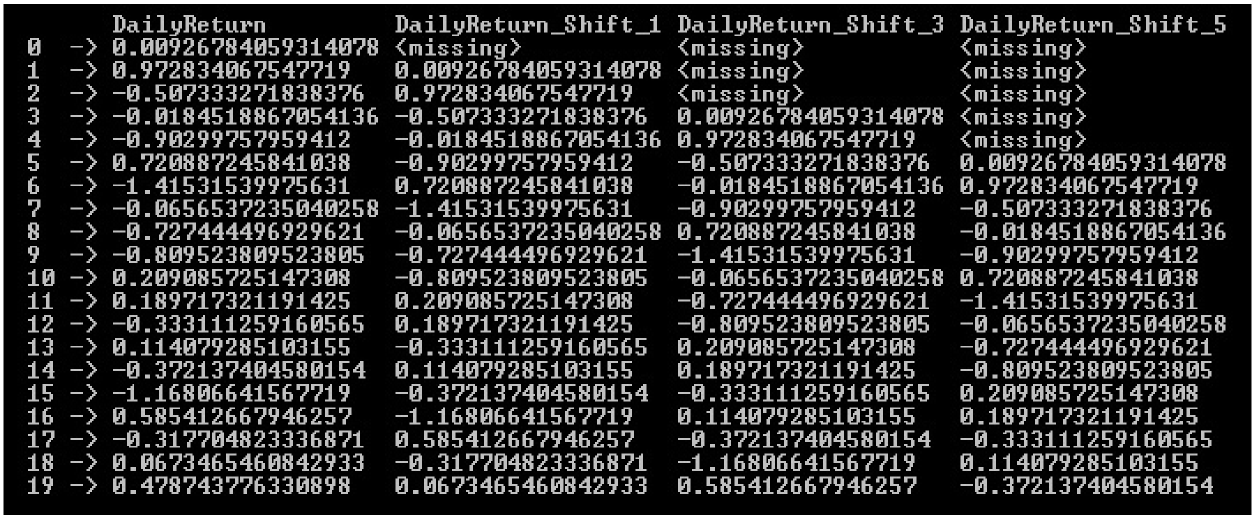
```
// Compute Daily Returns
rawDF.AddColumn(
    "DailyReturn",
    rawDF["Close"].Diff(1) / rawDF["Close"] * 100.0
);
```

In this code, we are taking the difference in close prices between the previous day and the current day and then dividing them by the previous close price. By multiplying them by `100`, we can get the daily returns in a percentage. Finally, we add this daily return series back to the original data frame with a column name, `DailyReturn`, by using the `AddColumn` method in Deedle's data frame.

However, we are not quite done yet with building the target variables. Since we are building a forecasting model, we need to take the next day return as the target variable. We can use the `Shift` method in Deedle's data frame to associate each record with the next day return. Similar to the `Diff` method, you can use the `Shift` method to move a series back and forth to any arbitrary point in time. The following code shows how you can move the `DailyReturn` column by `1`, `3`, and `5` steps:

```
rawDF["DailyReturn"].Shift(1)
rawDF["DailyReturn"].Shift(3)
rawDF["DailyReturn"].Shift(5)
```

The output of the preceding code is as follows:



```
      DailyReturn         DailyReturn_Shift_1 DailyReturn_Shift_3 DailyReturn_Shift_5
0  -> 0.00926784059314078 <missing>           <missing>           <missing>
1  -> 0.972834067547719   0.00926784059314078 <missing>           <missing>
2  -> -0.507333271838376  0.972834067547719   <missing>           <missing>
3  -> -0.0184518867054136 -0.507333271838376  0.00926784059314078 <missing>
4  -> -0.90299757959412   -0.0184518867054136 0.972834067547719   <missing>
5  -> 0.720887245841038   -0.90299757959412   -0.507333271838376  0.00926784059314078
6  -> -1.41531539975631   0.720887245841038   -0.0184518867054136 0.972834067547719
7  -> -0.0656537235040258 -1.41531539975631   -0.90299757959412   -0.507333271838376
8  -> -0.727444496929621  -0.0656537235040258 0.720887245841038   -0.0184518867054136
9  -> -0.809523809523805  -0.727444496929621  -1.41531539975631   -0.90299757959412
10 -> 0.209085725147308   -0.809523809523805  -0.0656537235040258 0.720887245841038
11 -> 0.189717321191425   0.209085725147308   -0.727444496929621  -1.41531539975631
12 -> -0.333111259160565  0.189717321191425   -0.809523809523805  -0.0656537235040258
13 -> 0.114079285103155   -0.333111259160565  0.209085725147308   -0.727444496929621
14 -> -0.372137404580154  0.114079285103155   0.189717321191425   -0.809523809523805
15 -> -1.16806641567719   -0.372137404580154  -0.333111259160565  0.209085725147308
16 -> 0.585412667946257   -1.16806641567719   0.114079285103155   0.189717321191425
17 -> -0.317704823336871  0.585412667946257   -0.372137404580154  -0.333111259160565
18 -> 0.0673465460842933  -0.317704823336871  -1.16806641567719   0.114079285103155
19 -> 0.478743776330898   0.0673465460842933  0.585412667946257   -0.372137404580154
```

As you can see from this example, the `DailyReturn` column or series has been moved forward by `1`, `3`, and `5` steps, depending on the parameters you fed into the `Shift` method. Using this `Shift` method, we are going to move daily returns back one step, so that each record has the next day's return as a target variable. The following code is how we created a target variable column, `Target`:

```
// Encode Target Variable - Predict Next Daily Return
rawDF.AddColumn(
    "Target",
    rawDF["DailyReturn"].Shift(-1)
);
```

Now that we have encoded target variables, there is one more step we need to take to get our data prepared for future tasks. When you are working with financial data, you will often hear the terms *OHLC chart* or *OHLC prices*. OHLC stands for Open, High, Low, and Close and it is often used to show price movements over time. If you look at the data that we downloaded, you will notice that open prices are missing in the dataset. However, we are going to need open prices for our future feature engineering step. We are going to assume that the open price for a given day is the close price of the previous day, given that the foreign exchange market is run 24 hours a day and is very liquid with high trading volume. In order to take previous close prices as open prices, we are going to use the `Shift` method. The following code shows how we created and added open prices into our data frame:

```
// Assume Open prices are previous Close prices
rawDF.AddColumn(
    "Open",
    rawDF["Close"].Shift(1)
);
```

The following code is the full code that we used for the data preparation step:

```
using Deedle;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DataPrep
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.SetWindowSize(100, 50);

            // Read in the raw dataset
            // TODO: change the path to point to your data directory
            string dataDirPath = @"\\Mac\Home\Documents\c-sharp-machine-
learning\ch.4\input-data";

            // Load the data into a data frame
            string rawDataPath = Path.Combine(dataDirPath, "eurusd-daily.csv");
            Console.WriteLine("Loading {0}\n", rawDataPath);
            var rawDF = Frame.ReadCsv(
                rawDataPath,
                hasHeaders: true,
                schema: "Date,float,float,float",
                inferTypes: false
            );

            // Rename & Simplify Column Names
            rawDF.RenameColumns(c => c.Contains("EUR/USD ") ? c.Replace("EUR/USD ", "")
: c);

            // Assume Open prices are previous Close prices
            rawDF.AddColumn(
                "Open",
                rawDF["Close"].Shift(1)
            );

            // Compute Daily Returns
            rawDF.AddColumn(
                "DailyReturn",
                rawDF["Close"].Diff(1) / rawDF["Close"] * 100.0
            );

            // Encode Target Variable - Predict Next Daily Return
            rawDF.AddColumn(
                "Target",
                rawDF["DailyReturn"].Shift(-1)
            );

            rawDF.Print();

            // Save OHLC data
            string ohlcDataPath = Path.Combine(dataDirPath, "eurusd-daily-ohlc.csv");
            Console.WriteLine("\nSaving OHLC data to {0}\n", rawDataPath);
            rawDF.SaveCsv(ohlcDataPath);

            Console.WriteLine("DONE!!");
            Console.ReadKey();
        }
    }
}
```

When you run this code, it is going to output the results into a file named `eurusd-daily-ohlc.csv`, which contains the OHLC prices, daily returns, and target variables. We are going to use this file for the future data analysis and feature engineering steps.

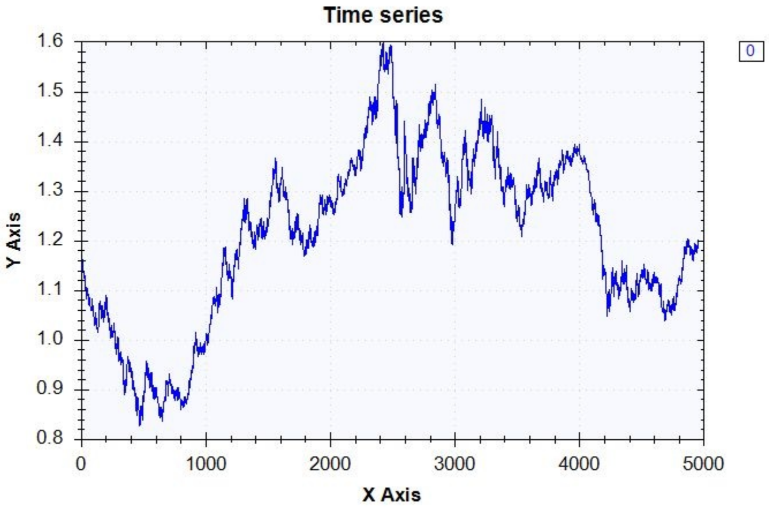
This code can also be found in the following repository: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.4/DataPrep.cs.

Time series data analysis

Let's start looking into the data. We are going to take the output from the previous data preparation step and start looking at the distributions of daily returns. Unlike previous chapters, where we primarily worked with categorical variables, we are dealing with continuous and time series variables. We will look at this data in a few different ways. First, let's look at the time series close prices chart. The following code shows how to build a line chart using the Accord.NET framework:

```
// Time-series line chart of close prices
DataSeriesBox.Show(
    ohlcDF.RowKeys.Select(x => (double)x),
    ohlcDF.GetColumn<double>("Close").ValuesAll
);
```

Refer to the Accord.NET documentation the `DataSeriesBox.Show` method for various other ways to display a line chart. In this example, we built a line chart with the integer indexes of our data frame as the *x* axis values and the close prices as the *y* axis values. The following is the time series line chart that you will see when you run the code:



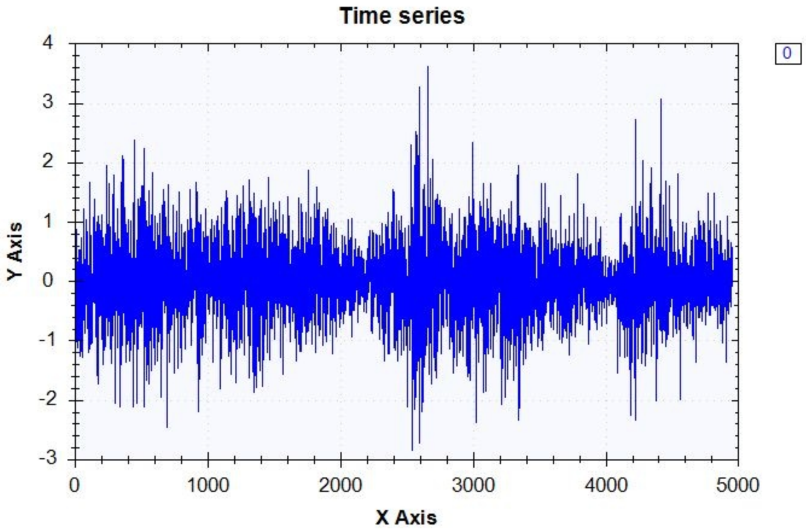
This chart shows us the overall movements of EUR/USD exchange rates over time from 1999 to 2017. It started from around 1.18 and went below 1.0 in 2000 and 2001. Then, it went as high as 1.6 in 2008 and then ended 2017 at around

1.20. Let's now look at the historical daily returns. The following code shows you how to build a line chart of historical daily returns:

```
// Time-series line chart of daily returns
DataSeriesBox.Show(
    ohlcDF.RowKeys.Select(x => (double)x),
    ohlcDF.FillMissing(0.0)["DailyReturn"].ValuesAll
);
```

One thing to note here is the usage of the `FillMissing` method. If you remember from the previous data preparation step, the `DailyReturn` series was built by taking the difference between the previous period and the current period. As a result, we have a missing value for the very first data point, since there is no previous period data point for the first record. The `FillMissing` method helps you encode missing values with custom values. Depending on your dataset and assumptions, you can encode missing values with different values, and the `FillMissing` method in Deedle's data frame will come in handy.

When you run the previous code, it will display a chart as follows:

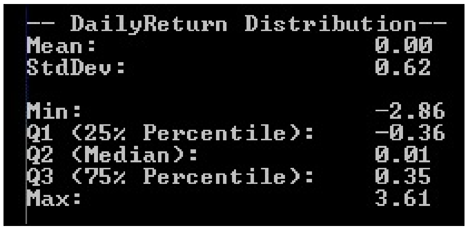


As you can see from this chart, daily returns oscillate around **0**, mostly between -2.0% and +2.0%. Let's look at the distribution of daily returns more closely. We are going to look at the minimum, maximum, mean, and standard deviation values. Then, we are going to look at the quartiles of daily returns, which we will discuss in more detail after looking at the code. The code to compute those numbers is as follows:

```
// Check the distribution of daily returns
double returnMax = ohlcDF["DailyReturn"].Max();
double returnMean = ohlcDF["DailyReturn"].Mean();
double returnMedian = ohlcDF["DailyReturn"].Median();
double returnMin = ohlcDF["DailyReturn"].Min();
double returnStdDev = ohlcDF["DailyReturn"].StdDev();

double[] quantiles = Accord.Statistics.Measures.Quantiles(
    ohlcDF.FillMissing(0.0)["DailyReturn"].ValuesAll.ToArray(),
    new double[] {0.25, 0.5, 0.75}
);

Console.WriteLine("-- DailyReturn Distribution-- ");

Console.WriteLine("Mean: \t\t\t{0:0.00}\nStdDev: \t\t{1:0.00}\n", returnMean,
returnStdDev);

Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    returnMin, quantiles[0], quantiles[1], quantiles[2], returnMax
);
```

As you can see from this code, the Deedle framework has numerous built-in methods for computing basic statistics. As shown in the first six lines of the code, you can use the `Max`, `Mean`, `Median`, `Min`, and `StdDev` methods in the Deedle framework in order to get corresponding statistics for daily returns.

In order to get quartiles, we need to use the `Quantiles` method in the `Accord.Statistics.Measures` module of the Accord.NET Framework. Quantiles are the points that divide an ordered distribution into equal-length intervals. For example, ten-quantiles break an ordered distribution into ten subsets of equal sizes, so that the first subset represents the bottom 10% of the distribution and the last subset represents the top 10% of the distribution. Similarly, four-quantiles break an ordered distribution into four subsets of equal sizes, where the first subset represents the bottom 25% of the distribution and the last subset represents the top 25% of the distribution. Four-quantiles are often called **quartiles**, ten-quantiles are called **deciles**, and 100-quantiles are called **percentiles**. As you can deduce from these definitions, the 1st quartile is the same as the 0.25th decile and the 25th percentile. Similarly, the 2nd and 3rd quartiles are the same as the 0.50th and 0.75th deciles and the 50th and 75th percentiles. As we are interested in the quartiles, we used 25%, 50%, and 75% as the inputs for the `percentiles` parameter in the `Quantiles` method. The following shows the output when you run this code:

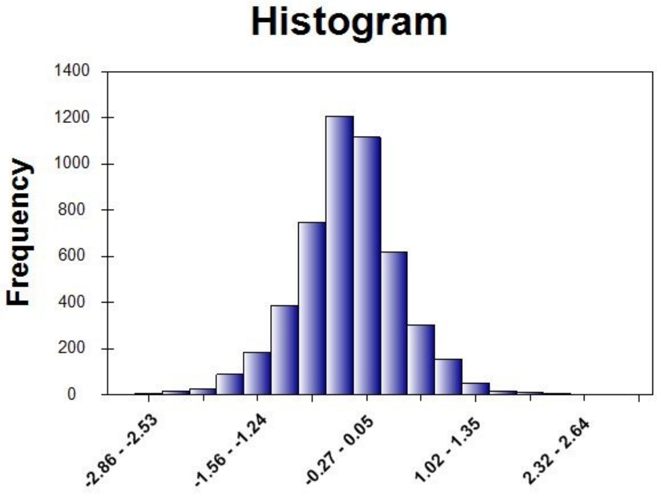


```
-- DailyReturn Distribution--
Mean:                        0.00
StdDev:                      0.62

Min:                        -2.86
Q1 (25% Percentile):        -0.36
Q2 (Median):                 0.01
Q3 (75% Percentile):         0.35
Max:                         3.61
```

Similar to what we have noticed from the daily return time series line chart, mean, and median are about 0, suggesting the daily returns oscillate around 0%. From 1999 to 2017, the largest negative daily return in history is -2.86% and the largest positive daily return is 3.61%. The first quartile, which is the middle number between the minimum and median, is at -0.36% and the third quartile, which is the middle number between the median and the maximum, is at 0.35%. From these summary statistics, we can see that the daily returns are spread almost symmetrically from 0%. To show this more visually, let's now look at the histogram of daily returns. The code to plot a histogram of daily returns is as follows:

```
var dailyReturnHistogram = HistogramBox
.Show(
    ohlcDF.FillMissing(0.0)["DailyReturn"].ValuesAll.ToArray()
)
.SetNumberOfBins(20);
```

We used `HistogramBox` in the Accord.NET framework to build a histogram chart of daily returns. Here, we set the number of bins as `20`. You can increase or decrease the number of bins to show more or less granular buckets. When you run this code, the following chart is what you will see:



Similar to what we have observed in the summary statistics, daily returns are spread almost symmetrically from 0%. This histogram of daily returns shows a clear bell curve, which suggests that daily returns follow a normal distribution.

The full code that we ran for this data analysis step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.4/DataAnalyzer.cs`.

Feature engineering

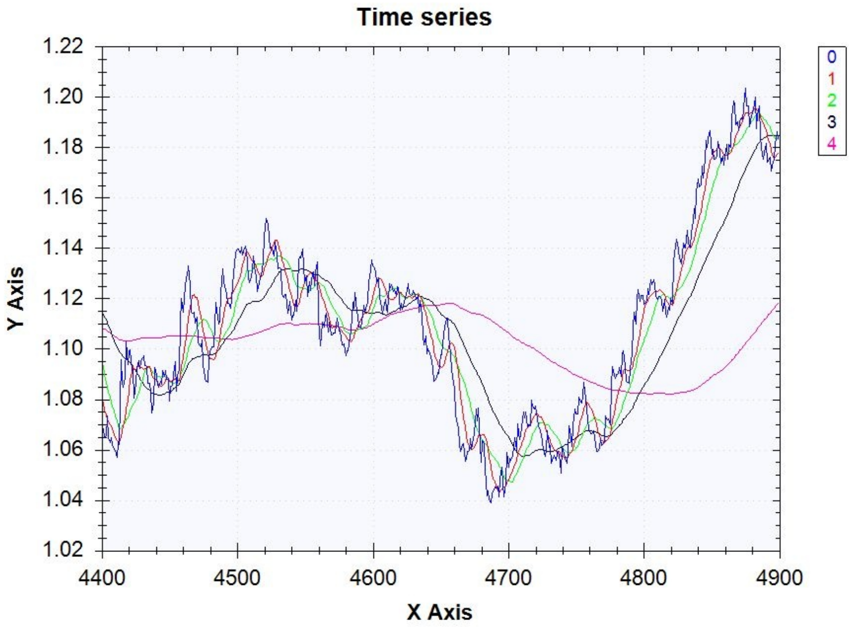
Now that we have a better understanding of the distribution of daily returns, let's start building features for our ML modeling. In this step, we are going to discuss a couple of frequently used technical indicators that traders in the foreign exchange market use and how we can build features for our ML models using those technical indicators.

Moving average

The first set of features we are going to build moving averages. A moving average is a rolling average for a pre-defined number of periods and is an often used technical indicator. A moving average helps smooth out volatile price movements and shows the overall trends of price actions. An in-depth discussion about how moving averages are used in trading financial assets is beyond the scope of this book, but in short, looking at multiple moving averages with different timeframes helps traders to identify trends and support and resistance levels for trading. In this chapter, we are going to use four moving averages, where the look-back periods are 10 days, 20 days, 50 days, and 200 days. The following code shows how we can compute moving averages using the `Window` method:

```
// 1. Moving Averages
ohlcDF.AddColumn("10_MA", ohlcDF.Window(10).Select(x => x.Value["Close"].Mean()));
ohlcDF.AddColumn("20_MA", ohlcDF.Window(20).Select(x => x.Value["Close"].Mean()));
ohlcDF.AddColumn("50_MA", ohlcDF.Window(50).Select(x => x.Value["Close"].Mean()));
ohlcDF.AddColumn("200_MA", ohlcDF.Window(200).Select(x => x.Value["Close"].Mean()));
```

The `Window` method in the Deedle framework helps us easily compute moving averages. The `Window` method takes a data frame and builds a series of data frames where each data frame contains a pre-defined number of records. For example, if your input to the `Window` method is `10`, then it is going to build a series of data frames, where the first data frame contains records from the 0th index to the 9th index, the second data frame contains records from the 1st index to the 11th index, and so forth. Using this method, we can easily compute moving averages for different time windows, as shown in the code. Now, let's plot a time series close price chart with these moving averages:



As you can see from this chart, moving averages smooth out volatile price movements. The red line shows moving averages of 10 days, the green line shows moving averages of 20 days, the black line for 50 days, and the pink line for 200 days. As you can see from this chart, the shorter the time window the closer it follows price actions and the less smooth the chart is. The code we used to generate this chart is as follows:

```
// Time-series line chart of close prices & moving averages
var maLineChart = DataSeriesBox.Show(
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).RowKeys.Select(x => (double)x),
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>
("Close").ValuesAll,
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>
("10_MA").ValuesAll,
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>
("20_MA").ValuesAll,
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>
("50_MA").ValuesAll,
    ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>
("200_MA").ValuesAll
);
```

With these moving averages we just calculated, the actual features that we are going to use for our model are the distances between the close price and the moving averages. As briefly mentioned, moving averages often work as support and resistance levels and by looking at how far each price point is from each of the moving averages, we can figure out whether we are approaching the support and resistance lines. The code to calculate the distances between the close price and the moving averages is as follows:

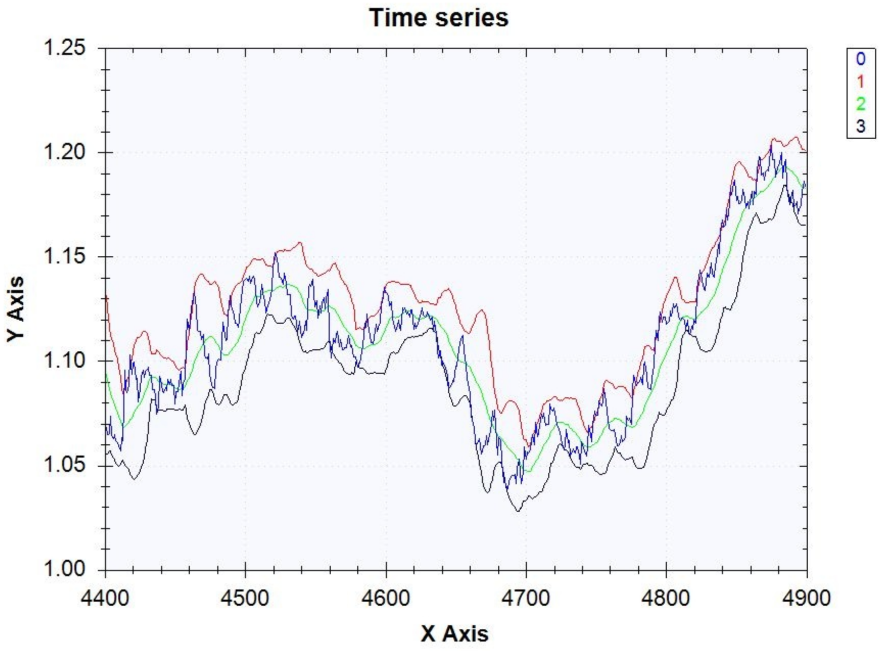
```
// Distance from moving averages
ohlcDF.AddColumn("Close_minus_10_MA", ohlcDF["Close"] - ohlcDF["10_MA"]);
ohlcDF.AddColumn("Close_minus_20_MA", ohlcDF["Close"] - ohlcDF["20_MA"]);
ohlcDF.AddColumn("Close_minus_50_MA", ohlcDF["Close"] - ohlcDF["50_MA"]);
ohlcDF.AddColumn("Close_minus_200_MA", ohlcDF["Close"] - ohlcDF["200_MA"]);
```

Bollinger Bands

The second technical indicator that we are going to look at is Bollinger Bands. Bollinger Bands comprise a moving average and the moving standard deviation of the same time window as the moving average that it is using. Then, Bollinger Bands are plotted two standard deviations above and below the moving average on the price time series chart. We will use a 20-day time window for computing Bollinger Bands. The code to compute Bollinger Bands is as follows:

```
// 2. Bollinger Band
ohlcDF.AddColumn("20_day_std", ohlcDF.Window(20).Select(x => x.Value["Close"].StdDev()));
ohlcDF.AddColumn("BollingerUpperBound", ohlcDF["20_MA"] + ohlcDF["20_day_std"] * 2);
ohlcDF.AddColumn("BollingerLowerBound", ohlcDF["20_MA"] - ohlcDF["20_day_std"] * 2);
```

As you can see from this code, we are using the `Window` and `StdDev` methods to calculate moving standard deviations. Then, we calculate the upper and lower boundaries of Bollinger Bands by adding and subtracting two standard deviations from 20-day moving averages. When you plot Bollinger Bands with price series, the result looks as follows:



The blue line shows the price movements, the green line shows 20-day moving averages, the red line shows the upper boundary of Bollinger Bands, which is two standard deviations above the moving averages, and the black line shows the lower boundary of Bollinger Bands, which is two standard deviations below the moving averages. As you can see from this chart, Bollinger Bands form bands around the price movements. The code to display this chart is as follows: // Time-series line chart of close prices & bollinger bands

```
var bbLineChart = DataSeriesBox.Show(
ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).RowKeys.Select(x => (double)x),
ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>("Close").ValuesAll,
ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>("BollingerUpperBound").ValuesAll,
ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>("20_MA").ValuesAll,
ohlcDF.Where(x => x.Key > 4400 && x.Key < 4900).GetColumn<double>("BollingerLowerBound").ValuesAll
);
```

Similar to the previous case of moving averages, we are going to use the distances between the close price and Bollinger Bands. Since most of the trades are made between the upper and lower bands, the distances between the price and the bands can be features for our ML models. The code to calculate the distances is as follows:

```
// Distance from Bollinger Bands
ohlcDF.AddColumn("Close_minus_BollingerUpperBound", ohlcDF["Close"] -
ohlcDF["BollingerUpperBound"]);
ohlcDF.AddColumn("Close_minus_BollingerLowerBound", ohlcDF["Close"] -
ohlcDF["BollingerLowerBound"]);
```

Lagged variables

Finally, the last set of features we are going to use is lagged variables. Lagged variables contain information about previous periods. For example, if we use the daily return value of a previous day as a feature for our model, then it is a lagged variable that lagged one period. We can also use the daily return of two days prior to the current date as a feature for our model. These types of variable are called **lagged variables** and are often used in time series modeling. We are going to use daily returns and previously built features as lagged variables. In this project, we look back as far as five periods, but you can experiment with longer or shorter look-back periods. The code to create lagged variables for daily returns is as follows:

```
// 3. Lagging Variables
ohlcDF.AddColumn("DailyReturn_T-1", ohlcDF["DailyReturn"].Shift(1));
ohlcDF.AddColumn("DailyReturn_T-2", ohlcDF["DailyReturn"].Shift(2));
ohlcDF.AddColumn("DailyReturn_T-3", ohlcDF["DailyReturn"].Shift(3));
ohlcDF.AddColumn("DailyReturn_T-4", ohlcDF["DailyReturn"].Shift(4));
ohlcDF.AddColumn("DailyReturn_T-5", ohlcDF["DailyReturn"].Shift(5));
```

Similarly, we can create lagged variables for the differences between moving averages and the close prices, using the following code:

```
ohlcDF.AddColumn("Close_minus_10_MA_T-1", ohlcDF["Close_minus_10_MA"].Shift(1));
ohlcDF.AddColumn("Close_minus_10_MA_T-2", ohlcDF["Close_minus_10_MA"].Shift(2));
ohlcDF.AddColumn("Close_minus_10_MA_T-3", ohlcDF["Close_minus_10_MA"].Shift(3));
ohlcDF.AddColumn("Close_minus_10_MA_T-4", ohlcDF["Close_minus_10_MA"].Shift(4));
ohlcDF.AddColumn("Close_minus_10_MA_T-5", ohlcDF["Close_minus_10_MA"].Shift(5));

ohlcDF.AddColumn("Close_minus_20_MA_T-1", ohlcDF["Close_minus_20_MA"].Shift(1));
ohlcDF.AddColumn("Close_minus_20_MA_T-2", ohlcDF["Close_minus_20_MA"].Shift(2));
ohlcDF.AddColumn("Close_minus_20_MA_T-3", ohlcDF["Close_minus_20_MA"].Shift(3));
ohlcDF.AddColumn("Close_minus_20_MA_T-4", ohlcDF["Close_minus_20_MA"].Shift(4));
ohlcDF.AddColumn("Close_minus_20_MA_T-5", ohlcDF["Close_minus_20_MA"].Shift(5));

ohlcDF.AddColumn("Close_minus_50_MA_T-1", ohlcDF["Close_minus_50_MA"].Shift(1));
ohlcDF.AddColumn("Close_minus_50_MA_T-2", ohlcDF["Close_minus_50_MA"].Shift(2));
ohlcDF.AddColumn("Close_minus_50_MA_T-3", ohlcDF["Close_minus_50_MA"].Shift(3));
ohlcDF.AddColumn("Close_minus_50_MA_T-4", ohlcDF["Close_minus_50_MA"].Shift(4));
ohlcDF.AddColumn("Close_minus_50_MA_T-5", ohlcDF["Close_minus_50_MA"].Shift(5));

ohlcDF.AddColumn("Close_minus_200_MA_T-1", ohlcDF["Close_minus_200_MA"].Shift(1));
ohlcDF.AddColumn("Close_minus_200_MA_T-2", ohlcDF["Close_minus_200_MA"].Shift(2));
ohlcDF.AddColumn("Close_minus_200_MA_T-3", ohlcDF["Close_minus_200_MA"].Shift(3));
ohlcDF.AddColumn("Close_minus_200_MA_T-4", ohlcDF["Close_minus_200_MA"].Shift(4));
ohlcDF.AddColumn("Close_minus_200_MA_T-5", ohlcDF["Close_minus_200_MA"].Shift(5));
```

Lastly, we can create lagged variables for Bollinger Band indicators, using the following code:

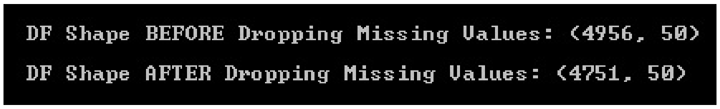
```
ohlcDF.AddColumn("Close_minus_BollingerUpperBound_T-1",
ohlcDF["Close_minus_BollingerUpperBound"].Shift(1));
ohlcDF.AddColumn("Close_minus_BollingerUpperBound_T-2",
ohlcDF["Close_minus_BollingerUpperBound"].Shift(2));
ohlcDF.AddColumn("Close_minus_BollingerUpperBound_T-3",
ohlcDF["Close_minus_BollingerUpperBound"].Shift(3));
ohlcDF.AddColumn("Close_minus_BollingerUpperBound_T-4",
ohlcDF["Close_minus_BollingerUpperBound"].Shift(4));
ohlcDF.AddColumn("Close_minus_BollingerUpperBound_T-5",
ohlcDF["Close_minus_BollingerUpperBound"].Shift(5));
```

As you can see from these code snippets, it is very simple and straightforward to create such lagged variables. We can simply use the `Shift` method in the Deedle framework and change the input to the method according to the look-back period.

One last thing we are going to do in this section is drop the missing values. Because we were building many time series features, we created a lot of missing values. For example, when we calculate 200-day moving averages, the first 199 records will have no moving averages, and as a result will have missing values. When you happen to have missing values in your dataset, there are two ways you can handle them—you can either encode them with certain values, or drop the missing values from the dataset. Since we have enough data, we are going to drop all the records with missing values. The code for dropping missing values from our data frame is as follows:

```
Console.WriteLine("\n\nDF Shape BEFORE Dropping Missing Values: ({0}, {1})",
ohlcDF.RowCount, ohlcDF.ColumnCount);
ohlcDF = ohlcDF.DropSparseRows();
Console.WriteLine("\nDF Shape AFTER Dropping Missing Values: ({0}, {1})\n\n",
ohlcDF.RowCount, ohlcDF.ColumnCount);
```

As you can see from this code, the Deedle framework has a handy function that we can use to drop missing values. We can use the `DropSparseRows` method to drop all the missing values. When you run this code, your output will look as follows:



```
DF Shape BEFORE Dropping Missing Values: (4956, 50)
DF Shape AFTER Dropping Missing Values: (4751, 50)
```

As you can see from this output, it dropped 250 records for having missing values. The full code to run the data analysis step from end to end can be found at this link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.4/Featur eEngineer.cs.

Linear regression versus SVM

In this section, we are going to build models that are completely different from previous chapters. We are going to build models that predict continuous variables and provide a daily return of EUR/USD exchange rates, and we are going to use two new learning algorithms, linear regression and SVM. Linear regression models try to find linear relationships between the target variables and the features, whereas SVM models try to build hyperplanes that maximize the distances between different classes. For this foreign exchange rate forecasting project, we are going to discuss how to build linear regression and SVM models for regression problems in C# using the Accord.NET Framework.

Before we build models, we will have to split our sample set into two subsets—one for training and another for testing. In the previous chapter, we used `SplitSetValidation` in the Accord.NET Framework to randomly split a sample set into train and test sets at a pre-defined proportion. However, we cannot apply the same approach in this chapter. Since we are dealing with time series data, we cannot randomly select and split records into train and test sets. If we do randomly split the sample set, then we are going to have cases where we train our ML models with future events and test our models on past events. So, we want to split our sample set at a certain point in time and take the records up to that point into a train set and the records after that point into a test set. The following code shows how we split our sample set into train and test sets:

```
// Read in the file we created in the previous step
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-data-dir>";

// Load the data into a data frame
Console.WriteLine("Loading data...");
var featuresDF = Frame.ReadCsv(
    Path.Combine(dataDirPath, "eurusd-features.csv"),
    hasHeaders: true,
    inferTypes: true
);

// Split the sample set into train and test sets
double trainProportion = 0.9;

int trainSetIndexMax = (int)(featuresDF.RowCount * trainProportion);

var trainSet = featuresDF.Where(x => x.Key < trainSetIndexMax);
var testSet = featuresDF.Where(x => x.Key >= trainSetIndexMax);
```

```
Console.WriteLine("\nTrain Set Shape: ({0}, {1})", trainSet.RowCount,
trainSet.ColumnCount);
Console.WriteLine("Test Set Shape: ({0}, {1})", testSet.RowCount, testSet.ColumnCount);
```

As you can see from this code snippet, we take the first 90% of the sample set for training and the remaining 10% for testing, using the `Where` method to filter records in the sample set by index. The next thing we need to do before training our ML models is select the features that we want to train our models with. Since we are only interested in using lagged variables and the distances between the prices and moving averages or Bollinger Bands, we do not want to include raw moving average or Bollinger Band numbers into our feature space. The following code snippet shows how we define the feature set for our models:

```
string[] features = new string[] {
    "DailyReturn",
    "Close_minus_10_MA", "Close_minus_20_MA", "Close_minus_50_MA",
    "Close_minus_200_MA", "20_day_std",
    "Close_minus_BollingerUpperBound", "Close_minus_BollingerLowerBound",
    "DailyReturn_T-1", "DailyReturn_T-2",
    "DailyReturn_T-3", "DailyReturn_T-4", "DailyReturn_T-5",
    "Close_minus_10_MA_T-1", "Close_minus_10_MA_T-2",
    "Close_minus_10_MA_T-3", "Close_minus_10_MA_T-4",
    "Close_minus_10_MA_T-5",
    "Close_minus_20_MA_T-1", "Close_minus_20_MA_T-2",
    "Close_minus_20_MA_T-3", "Close_minus_20_MA_T-4", "Close_minus_20_MA_T-5",
    "Close_minus_50_MA_T-1", "Close_minus_50_MA_T-2", "Close_minus_50_MA_T-3",
    "Close_minus_50_MA_T-4", "Close_minus_50_MA_T-5",
    "Close_minus_200_MA_T-1", "Close_minus_200_MA_T-2",
    "Close_minus_200_MA_T-3", "Close_minus_200_MA_T-4",
    "Close_minus_200_MA_T-5",
    "Close_minus_BollingerUpperBound_T-1",
    "Close_minus_BollingerUpperBound_T-2", "Close_minus_BollingerUpperBound_T-3",
    "Close_minus_BollingerUpperBound_T-4", "Close_minus_BollingerUpperBound_T-5"
};
```

Now we are ready to start building model objects and training our ML models. Let's first look at how to instantiate a linear regression model. The code we used to train a linear regression model is as follows:

```
Console.WriteLine("\n**** Linear Regression Model ****");

// OLS learning algorithm
var ols = new OrdinaryLeastSquares()
{
    UseIntercept = true
};

// Fit a linear regression model
MultipleLinearRegression regFit = ols.Learn(trainX, trainY);

// in-sample predictions
double[] regInSamplePreds = regFit.Transform(trainX);
```

```
// out-of-sample predictions
double[] regOutSamplePreds = regFit.Transform(testX);
```

As you can see from this code snippet, we are using `OrdinaryLeastSquares` as a learning algorithm and `MultipleLinearRegression` as a model. **Ordinary Least Squares** (**OLS**) is a way of training a linear regression model by minimizing and optimizing on the sum of squares of errors. A multiple linear regression model is a model where the number of input features is larger than 1. Lastly, in order to make predictions on data, we are using the `Transform` method of the `MultipleLinearRegression` object. We will be making predictions on both the train and test sets for our model validations in the following section.

Let's now look at another learning algorithm and model that we are going to use in this chapter. The following code shows how to build and train a SVM model for regression problems:

```
Console.WriteLine("\n**** Linear Support Vector Machine ****");
// Linear SVM Learning Algorithm
var teacher = new LinearRegressionNewtonMethod()
{
    Epsilon = 2.1,
    Tolerance = 1e-5,
    UseComplexityHeuristic = true
};

// Train SVM
var svm = teacher.Learn(trainX, trainY);

// in-sample predictions
double[] linSVMInSamplePreds = svm.Score(trainX);

// out-of-sample predictions
double[] linSVMOutSamplePreds = svm.Score(testX);
```

As you can see from this code, we are using `LinearRegressionNewtonMethod` as a learning algorithm to train a SVM model. `LinearRegressionNewtonMethod` is a learning algorithm for SVM using linear kernel. Simply put, a kernel is a way of projecting data points onto another space where the data points are more separable than in the original space. Other kernels, such as polynomial and Gaussian kernels, are also often used when training SVM models. We will experiment with and further discuss these other kernels in the next chapter, but you can certainly experiment with the model performances on other kernels for this project. When making predictions with a trained SVM model, you can use the `Score` method, as shown in the code snippet.

The full code that we used to train and validate linear regression and SVM models can be found here: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.4/Modeling.cs.

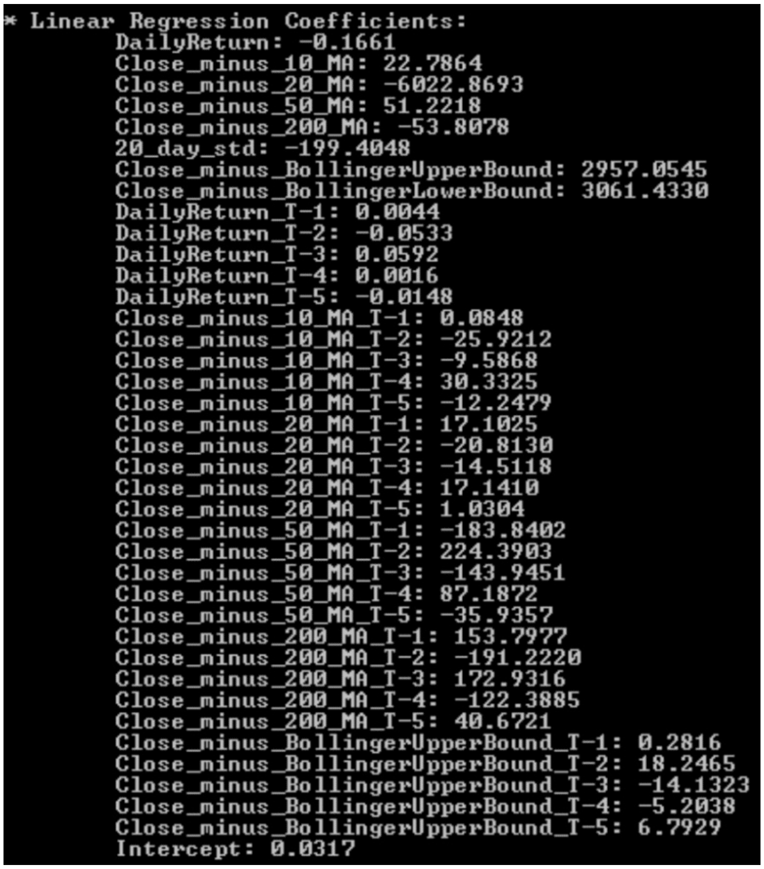
Model validations

Now that you have built and trained regression models for this chapter's foreign exchange rate forecast project, let's start looking into how our models performed. In this section, we are going to discuss two commonly used basic metrics, RMSE, and R^2, and a diagnostic plot, actual or observed values versus predicted values. Before we delve into those metrics and a diagnostic plot, let's first briefly discuss how to extract coefficient and intercept values from the linear regression model.

The following code snippet shows you how to extract coefficients and the intercept from the `MultipleLinearRegression` object:

```
Console.WriteLine("\n* Linear Regression Coefficients:");

for (int i = 0; i < features.Length; i++)
{
    Console.WriteLine("\t{0}: {1:0.0000}", features[i], regFit.Weights[i]);
}

Console.WriteLine("\tIntercept: {0:0.0000}", regFit.Intercept);
```

When you run this code, you will see an output like the following:



```
* Linear Regression Coefficients:
        DailyReturn: -0.1661
        Close_minus_10_MA: 22.7864
        Close_minus_20_MA: -6022.8693
        Close_minus_50_MA: 51.2218
        Close_minus_200_MA: -53.8078
        20_day_std: -199.4048
        Close_minus_BollingerUpperBound: 2957.0545
        Close_minus_BollingerLowerBound: 3061.4330
        DailyReturn_T-1: 0.0044
        DailyReturn_T-2: -0.0533
        DailyReturn_T-3: 0.0592
        DailyReturn_T-4: 0.0016
        DailyReturn_T-5: -0.0148
        Close_minus_10_MA_T-1: 0.0848
        Close_minus_10_MA_T-2: -25.9212
        Close_minus_10_MA_T-3: -9.5868
        Close_minus_10_MA_T-4: 30.3325
        Close_minus_10_MA_T-5: -12.2479
        Close_minus_20_MA_T-1: 17.1025
        Close_minus_20_MA_T-2: -20.8130
        Close_minus_20_MA_T-3: -14.5118
        Close_minus_20_MA_T-4: 17.1410
        Close_minus_20_MA_T-5: 1.0304
        Close_minus_50_MA_T-1: -183.8402
        Close_minus_50_MA_T-2: 224.3903
        Close_minus_50_MA_T-3: -143.9451
        Close_minus_50_MA_T-4: 87.1872
        Close_minus_50_MA_T-5: -35.9357
        Close_minus_200_MA_T-1: 153.7977
        Close_minus_200_MA_T-2: -191.2220
        Close_minus_200_MA_T-3: 172.9316
        Close_minus_200_MA_T-4: -122.3885
        Close_minus_200_MA_T-5: 40.6721
        Close_minus_BollingerUpperBound_T-1: 0.2816
        Close_minus_BollingerUpperBound_T-2: 18.2465
        Close_minus_BollingerUpperBound_T-3: -14.1323
        Close_minus_BollingerUpperBound_T-4: -5.2038
        Close_minus_BollingerUpperBound_T-5: 6.7929
        Intercept: 0.0317
```

Looking at the coefficients and intercept of the fitted linear regression model helps us understand the model and gain some insights into how each feature affects the prediction results. The fact that we can understand and visualize exactly how the relationships between the features and the target variable are formed and how they interact with each other makes linear regression models still attractive, even though other black-box models, such as random forest models or support vector machines, often outperform linear regression models. As you can see from this output, you can easily tell which features affect daily return predictions negatively or positively and the magnitudes of their impacts.

Let's now look at the first metrics that we are using for regression model validation in this chapter. You might already be familiar with RMSE, which measures the square root of the errors between the predicted values and the actual values. The lower RMSE values are, the better the model fit is. The following code shows how you can compute the RMSE of the model fit:

```
// RMSE for in-sample
double regInSampleRMSE = Math.Sqrt(new SquareLoss(trainX).Loss(regInSamplePreds));

// RMSE for out-sample
```

```
double regOutSampleRMSE = Math.Sqrt(new SquareLoss(testX).Loss(regOutSamplePreds));

Console.WriteLine("RMSE: {0:0.0000} (Train) vs. {1:0.0000} (Test)", regInSampleRMSE,
regOutSampleRMSE);
```

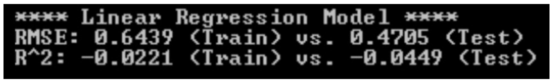
As you can see from this code, we are using the `SquareLoss` class in the Accord.NET framework, which computes the squared values of the differences between the predicted and the actual values. In order to get the RMSE, we need to take a square root of this value.

The next metric that we are going to look at is R^2. R^2 is frequently used as an indicator of the goodness of fit. The closer the value is to 1, the better the model fit. The following code shows how we can compute R^2 values:

```
// R^2 for in-sample
double regInSampleR2 = new RSquaredLoss(trainX[0].Length,
trainX).Loss(regInSamplePreds);

// R^2 for out-sample
double regOutSampleR2 = new RSquaredLoss(testX[0].Length,
testX).Loss(regOutSamplePreds);

Console.WriteLine("R^2: {0:0.0000} (Train) vs. {1:0.0000} (Test)", regInSampleR2,
regOutSampleR2);
```

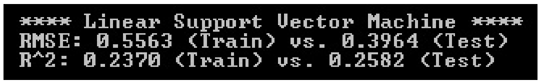
As you can see from this code, we are using the `RSquaredLoss` class in the Accord.NET framework. We are computing once for in-sample predictions (predictions on the train set) and once for out-of-sample predictions (predictions on the test set). The closer the two values are, the less overfitting the models.

When you run this code for RMSE and R^2 for the linear regression model, the following is an output you will get:



```
**** Linear Regression Model ****
RMSE: 0.6439 (Train) vs. 0.4705 (Test)
R^2: -0.0221 (Train) vs. -0.0449 (Test)
```

And for the SVM model, the output you will see is as follows:



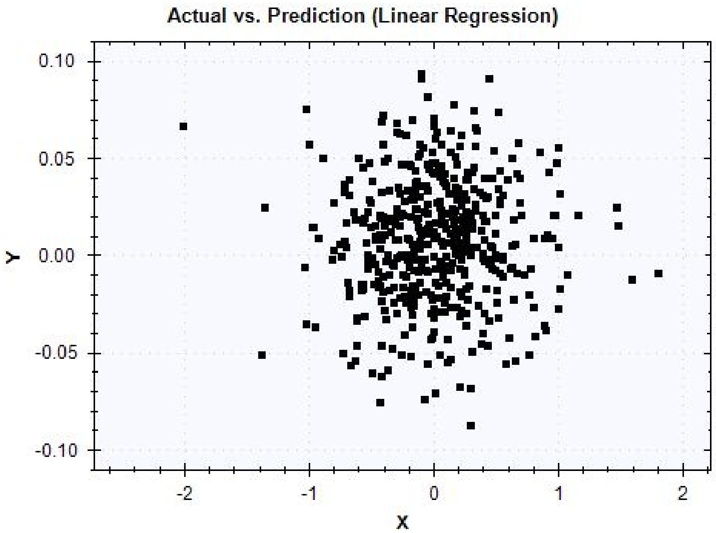
```
**** Linear Support Vector Machine ****
RMSE: 0.5563 (Train) vs. 0.3964 (Test)
R^2: 0.2370 (Train) vs. 0.2582 (Test)
```

From these outputs, we can see that the SVM model outperforms the linear regression model by a large amount. The SVM model has a much lower RMSE compared to the linear regression model. Also, the SVM model has a much higher R^2 value than the linear regression model. Note the R^2 value for the linear regression model. This happens when the fit of the model is worse than a simple horizontal line, and this suggests that our linear regression model fit is not good. On the other hand, the R^2 for the SVM model is about 0.26, which suggests that 26% of the target variable variance can be explained by this model.

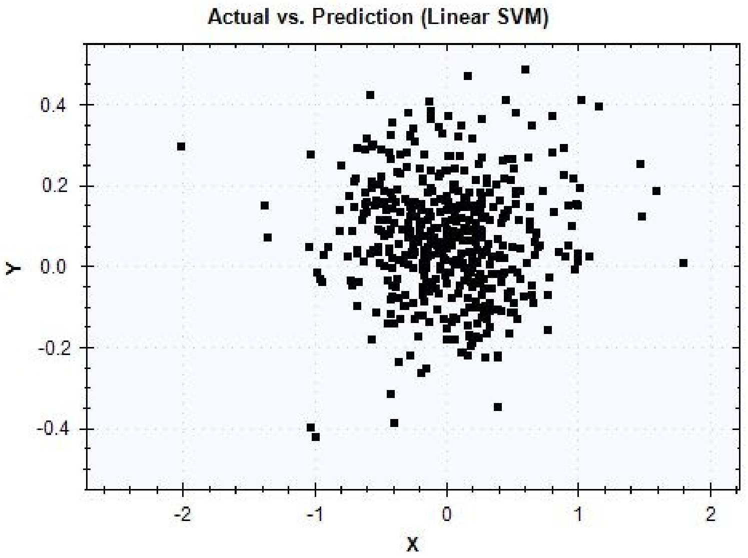
Lastly, we are going to look at a diagnostic plot; actual values versus predicted values. This diagnostic plot is a good way to visually see the goodness of the model fit. Ideally, we would want all the points to be on a diagonal line. For example, if the actual value is 1.0, then we would want to have predicted value close to 1.0. The closer the points are to the diagonal line, the better the model fit is. You can use the following code to plot actual values against predicted values:

```
// Scatter Plot of expected and actual
ScatterplotBox.Show(
    String.Format("Actual vs. Prediction ({0})", modelName), testY, regOutSamplePreds
);
```

We are using the `ScatterplotBox` class in the Accord.NET framework to build a scatter plot of actual values against predicted values. When you run this code for linear regression model results, you will see the following diagnostic plot:



When you run the same code for the SVM model results, the diagnostic plot appears as follows:



As you can see from these plots, predictions from the linear regression model are more clogged around 0, while those from the SVM model are more spread out across a wider range. Although both plots for the linear regression and SVM model results do now show a perfect diagonal line, the plot for the SVM model shows better results and is aligned with the results we have seen from the RMSE and R^2 metrics.

The method we wrote and used to run validations for the models is as follows:

```
private static void ValidateModelResults(string modelName, double[] regInSamplePreds,
double[] regOutSamplePreds, double[][] trainX, double[] trainY, double[][] testX,
double[] testY)
{
    // RMSE for in-sample
    double regInSampleRMSE = Math.Sqrt(new SquareLoss(trainX).Loss(regInSamplePreds));

    // RMSE for in-sample
    double regOutSampleRMSE = Math.Sqrt(new SquareLoss(testX).Loss(regOutSamplePreds));

    Console.WriteLine("RMSE: {0:0.0000} (Train) vs. {1:0.0000} (Test)",
regInSampleRMSE, regOutSampleRMSE);

    // R^2 for in-sample
    double regInSampleR2 = new RSquaredLoss(trainX[0].Length,
trainX).Loss(regInSamplePreds);

    // R^2 for in-sample
    double regOutSampleR2 = new RSquaredLoss(testX[0].Length,
testX).Loss(regOutSamplePreds);

    Console.WriteLine("R^2: {0:0.0000} (Train) vs. {1:0.0000} (Test)", regInSampleR2,
regOutSampleR2);

    // Scatter Plot of expected and actual
    ScatterplotBox.Show(
        String.Format("Actual vs. Prediction ({0})", modelName), testY,
regOutSamplePreds
    );
}
```

Summary

In this chapter, we built and trained our first regression models. We used a time series dataset that contains historical daily exchange rates between Euros and U.S. Dollars from 1999 to 2017. We first discussed how to create a target variable from an unlabeled raw dataset and how to apply the `Shift` and `Diff` methods in the Deedle framework in order to compute daily returns and create the target variable, which is the daily return for one period ahead. We further looked at the distributions of daily returns in a few different ways, such as a time series line chart, summary statistics using mean, standard deviation, and quantiles. We also looked at the histogram of daily returns and saw a well-drawn bell curve that follows a normal distribution. Then, we covered a few frequently used technical indicators in the foreign exchange market and how to apply them to our feature building processes. Using technical indicators, such as moving averages, Bollinger Bands, and lagged variables, we built various features that help our learning algorithms to learn how to predict future daily returns. With these features that we built in the feature engineering step, we built linear regression and SVM models to forecaste EUR/USD rates. We learned how to extract coefficients and the intercept from the `MultipleLinearRegression` object to gain insights into, and a better understanding, of how each feature affects the outcome of predictions. We briefly discussed the usage of kernels in building SVM models. Lastly, we went over two frequently used metrics for regression models, RMSE and R^2, and a diagnostic plot of actual values versus predicted values. From this model validation step, we observed how the SVM model outperformed the linear regression model by a large amount. We also discussed the comparative benefits of explainability we can gain from using the linear regression model, compared to other black-box models such as random forest and SVM models.

In the next chapter, we are going to extend our knowledge and experience by building regression models in C# using the Accord.NET framework. We will use a house price dataset that contains both continuous and categorical variables and learn how to build regression models for such a complex dataset. We will also discuss various other kernels we can use for SVMs and how they affect the performance of our SVM models.

Fair Value of House and Property

In this chapter, we are going to expand our knowledge and skills in building regression **machine learning** (**ML**) models in C#. In the last chapter, we built a linear regression and linear support vector machine model on a foreign exchange rate dataset, where all the features were continuous variables. However, we are going to be dealing with a more complex dataset, where some features are categorical variables and some others are continuous variables.

In this chapter, we will be using a house prices dataset that contains numerous attributes of houses with mixed variable types. Using this data, we will start looking at the two common types of categorical variables (ordinal versus non-ordinal) and the distributions of some of the categorical variables in the housing dataset. We will also look at the distributions of some of the continuous variables in the dataset and the benefits of using log transformations for variables that show skewed distributions. Then, we are going to learn how to encode and engineer such categorical features so that we can fit machine learning models. Unlike the last chapter, where we explored the basics of **Support Vector Machine** (**SVM**), we are going to apply different Kernel methods for our SVM models and see how it affects the model performances.

Similar to the last chapter, we will be using **root mean squared error** (**RMSE**), R^2, and a plot of actual versus predicted values to evaluate the performances of our ML models. By the end of this chapter, you will have a better understanding of how to handle categorical variables, how to encode and engineer such features for regression models, how to apply various kernel methods for building SVM models, and how to build models that predict the fair values of houses.

In this chapter, we will cover the following topics:

- Problem definition for the fair value of house/property project
- Data analysis for categorical versus continuous variables
- Feature engineering and encoding
- Linear regression versus Support Vector Machine with kernels
- Model validations using RMSE, R^2, and actual versus predicted plot

Problem definition

Let's start this chapter by understanding exactly what ML models we are going to build. When you are looking for a house or a property to purchase, you consider numerous attributes of those houses or properties that you look at. You might be looking at the number of bedrooms and bathrooms, how many cars you can park in your garage, the neighborhoods, the materials or finishes of the house, and so forth. All of these attributes of a house or property go into how you decide the price you want to pay for the given property or how you negotiate the price with the seller. However, it is very difficult to understand and estimate what the fair value of a property is. By having a model that predicts the fair value or the final price of each property, you can make better informed decisions when you are negotiating with the seller.

In order to build such models for fair value of a house predictions, we are going to use a dataset that contains 79 explanatory variables that cover almost all attributes of residential homes in Ames, Iowa, U.S.A. and their final sale prices from 2006 to 2010. This dataset was compiled by Dean De Cock (`https://ww2.amstat.org/publications/jse/v19n3/decock.pdf`) at the Truman State University and can be downloaded from this link: `https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data`. With this data, we are going to build features that contain information about square footage or sizes of different parts of the houses, the styles and materials used for the houses, the conditions and finishes of different parts of the houses, and various other attributes that further describe the information of each house. Using these features, we are going to explore different regression machine learning models, such as linear regression, Linear Support Vector Machine, and **Support Vector Machines** (**SVMs**) with polynomial and Gaussian kernels. Then, we will evaluate these models by looking at RMSE, R^2, and a plot of actual versus predicted values.

To summarize our problem definition for the fair value of house and property project:

- What is the problem? We need a regression model that predicts the fair values of residential homes in Ames, Iowa, U.S.A., so that we can understand and make better informed decisions when purchasing houses.
- Why is it a problem? Due to the complex nature and numerous moving parts in deciding the fair value of a house or a property, it is advantageous to have a machine learning model that can predict and inform home buyers what the expected values of houses that they are looking at are.
- What are some of the approaches to solving this problem? We are going to use a pre-compiled dataset that contains 79 explanatory variables that contain information of residential homes in Ames, Iowa, U.S.A., and build and encode features of mixed types (both categorical and continuous). Then, we will explore linear regression and support vector machines with different Kernels for making predictions of fair values of houses. We will evaluate the model candidates by looking at RMSE, R^2, and an actual versus predicted values plot.
- What are the success criteria? As we want our predictions of house prices to be as close to the actual house sale prices as possible, we want to gain as low an RMSE as possible, without hurting our goodness of fit measure, R^2, and the plot of actual versus predicted values.

Categorical versus continuous variables

Now let's start looking at the actual dataset. You can follow this link: https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data and download the train.csv and data_description.txt files. We are going to build models using the train.csv file, and the data_description.txt file will help us better understand the structure of the dataset, especially concerning the categorical variables we have.

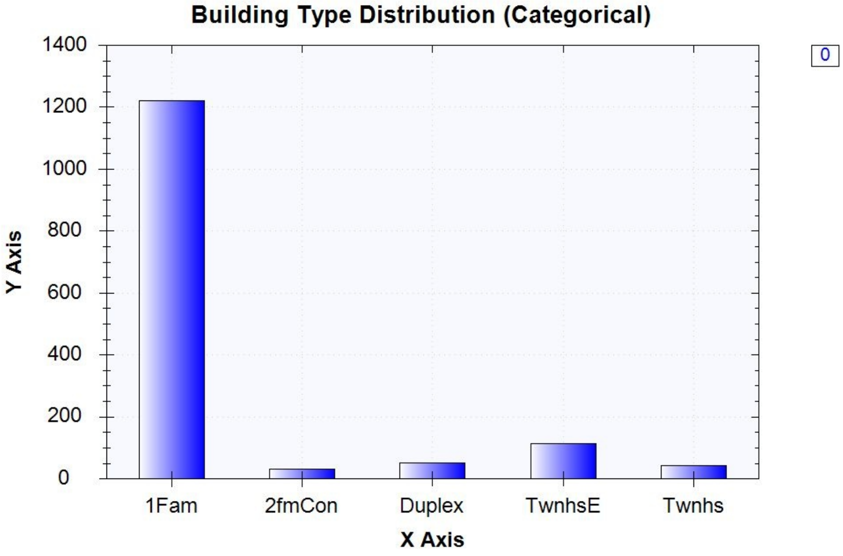
If you look at the train data file and the description file, you can easily find that there are some variables with certain names or codes that represent specific types of each house's attributes. For example, the Foundation variable can take one of the values among BrkTil, CBlock, PConc, Slab, Stone, and Wood, where each of those values or codes represents the type of foundation that a house is built with—Brick and Tile, Cinder Block, Poured Contrete, Slab, Stone, and Wood respectively. On the other hand, if you look at the TotalBsmtSF variable in the data, you can see that it can take any numerical values and the values are continuous. As mentioned previously, this dataset contains mixed types of variables and we need to approach carefully when we are dealing with a dataset with both categorical and continuous variables.

Non-ordinal categorical variables

Let's first look at some categorical variables and their distributions. The first house attribute that we are going to look at is the building type. The code to build a bar chart that shows the distributions of the building type is as follows:

```
// Categorical Variable #1: Building Type
Console.WriteLine("\nCategorical Variable #1: Building Type");
var buildingTypeDistribution = houseDF.GetColumn&lt;string&gt;(
"BldgType"
).GroupBy&lt;string&gt;(x =&gt; x.Value).Select(x =&gt;
(double)x.Value.KeyCount);
buildingTypeDistribution.Print();

var buildingTypeBarChart = DataBarBox.Show(
buildingTypeDistribution.Keys.ToArray(),
buildingTypeDistribution.Values.ToArray()
);
buildingTypeBarChart.SetTitle("Building Type Distribution (Categorical)");
```

When you run this code, it will display a bar chart like the following:

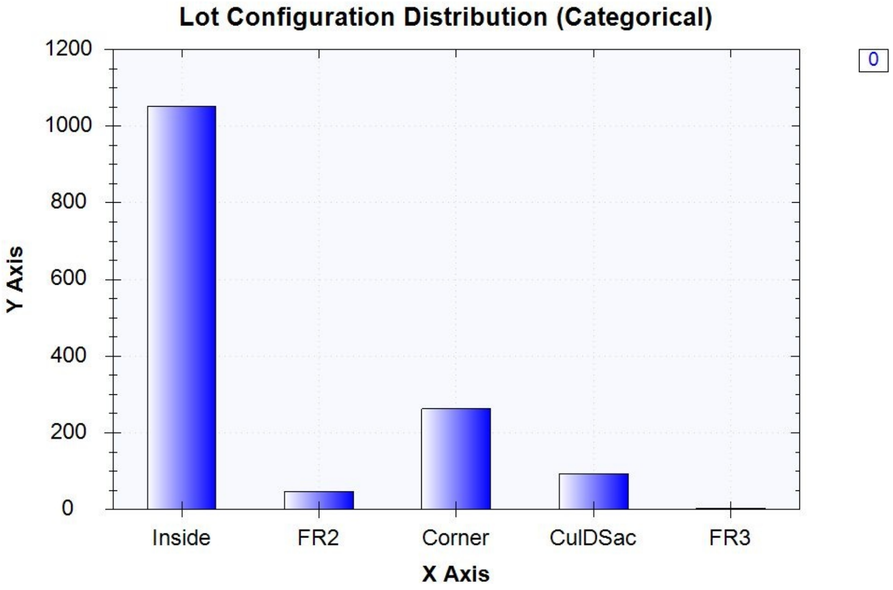


As you can tell from this bar chart, the majority of the building types in our dataset is 1Fam, which represents the *Single-family Detached* building type. The second most common building type is TwnhsE, which represents the *Townhouse End Unit* building type.

Let's take a look at one more categorical variable, Lot Configuration (`LotConfig` field in the dataset). The code to build a bar chart for lot configuration distributions is as follows:

```
// Categorical Variable #2: Lot Configuration
Console.WriteLine("\nCategorical Variable #1: Building Type");
var lotConfigDistribution = houseDF.GetColumn&lt;string&gt;(
    "LotConfig"
).GroupBy&lt;string&gt;(x =&gt; x.Value).Select(x =&gt; (double)x.Value.KeyCount);
lotConfigDistribution.Print();

var lotConfigBarChart = DataBarBox.Show(
    lotConfigDistribution.Keys.ToArray(),
    lotConfigDistribution.Values.ToArray()
);
lotConfigBarChart.SetTitle("Lot Configuration Distribution (Categorical)");
```

When you run this code, it will display the following bar chart:



As you can see from this bar chart, inside lot is the most common lot configuration in our dataset, and corner lot is the second most common log configuration.

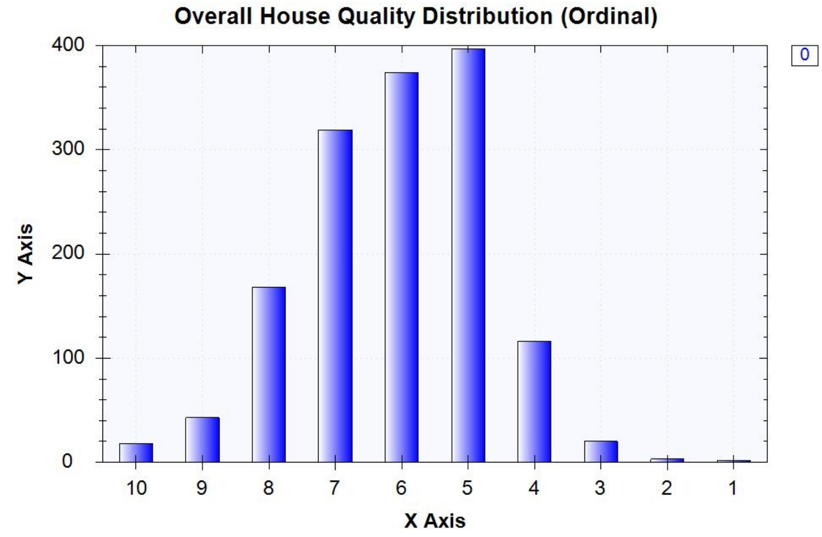
Ordinal categorical variable

The two categorical variables that we just looked at have no natural ordering. One type does not come before another or one type does not have more weight than another. However, there are some categorical variables that have natural ordering, and we call such categorical variables ordinal categorical variables. For example, when you rank a quality of a material from 1 to 10, where 10 represents the best and 1 represents the worst, there is a natural ordering. Let's look at some of the ordinal categorical variables in this dataset.

The first ordinal categorical variable that we are going to look at is the `OverallQual` attribute, which represents the overall material and finish of the house. The code to look at the distributions of this variable is as follows:

```
// Ordinal Categorical Variable #1: Overall material and finish of the house
Console.WriteLine("\nOrdinal Categorical #1: Overall material and finish of the
house");
var overallQualDistribution = houseDF.GetColumn&lt;string&gt;(
    "OverallQual"
).GroupBy&lt;int&gt;(
    x =&gt; Convert.ToInt32(x.Value)
).Select(
    x =&gt; (double)x.Value.KeyCount
).SortByKey().Reversed;
overallQualDistribution.Print();

var overallQualBarChart = DataBarBox.Show(
    overallQualDistribution.Keys.Select(x =&gt; x.ToString()),
    overallQualDistribution.Values.ToArray()
);
overallQualBarChart.SetTitle("Overall House Quality Distribution (Ordinal)");
```

When you run this code, it will display the following bar chart in order from 10 to 1:

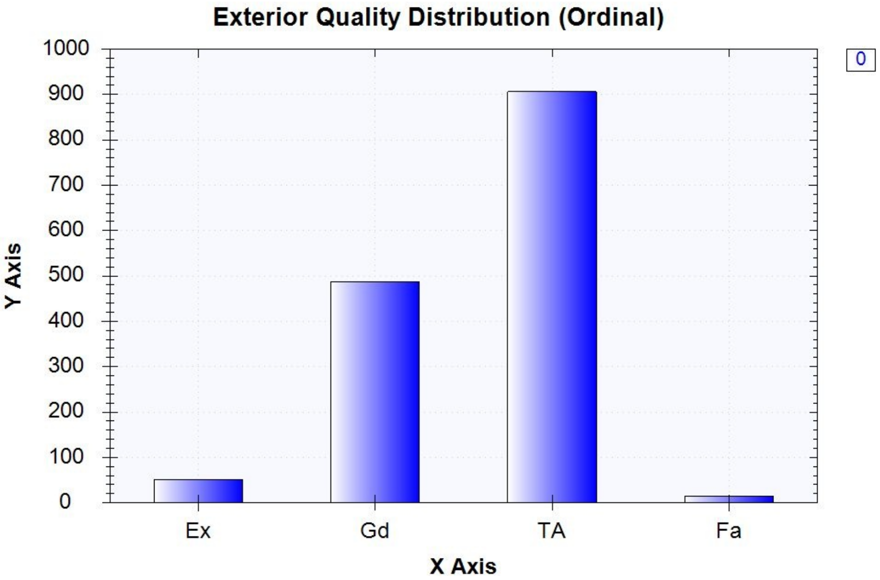


As expected, there is a smaller number of houses in the *Very Excellent*, encoded as 10, or *Excellent*, encoded as 9, categories than there are in the *Above Average*, encoded as 6, or *Average* categories, encoded as 5.

Another ordinal categorical variable that we will be looking at is the `ExterQual` variable, which represents the exterior quality. The code to look at the distributions of this variable is as follows:

```
// Ordinal Categorical Variable #2: Exterior Quality
Console.WriteLine("\nOrdinal Categorical #2: Exterior Quality");
var exteriorQualDistribution = houseDF.GetColumn&lt;string&gt;(
    "ExterQual"
).GroupBy&lt;string&gt;(x =&gt; x.Value).Select(
    x =&gt; (double)x.Value.KeyCount
)[new string[] { "Ex", "Gd", "TA", "Fa" }];
exteriorQualDistribution.Print();

var exteriorQualBarChart = DataBarBox.Show(
    exteriorQualDistribution.Keys.Select(x =&gt; x.ToString()),
    exteriorQualDistribution.Values.ToArray()
);
exteriorQualBarChart.SetTitle("Exterior Quality Distribution (Ordinal)");
```

When you run this code, it will display the following bar chart:



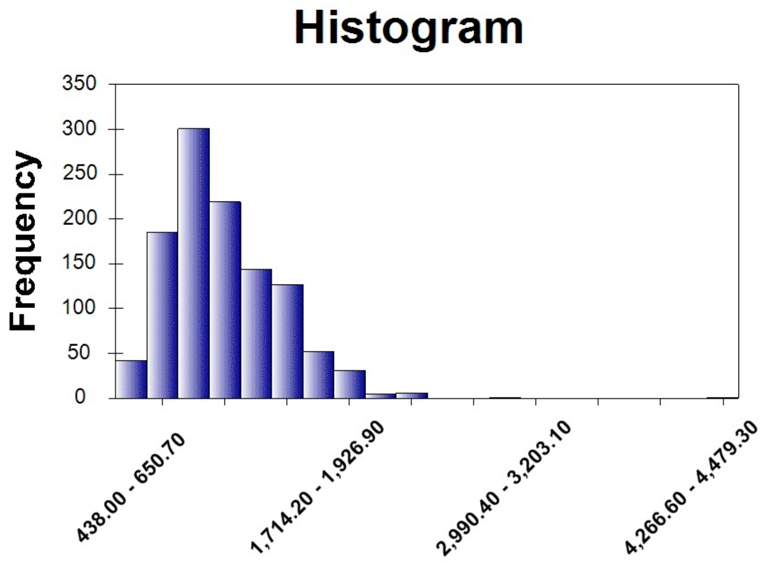
Unlike the `OverallQual` variable, the `ExterQual` variable does not have numerical values for the ordering. In our dataset, it has one of the following values: `Ex`, `Gd`, `TA`, and `FA`, and these represent excellent, good, average/typical, and fair respectively. Although this variable does not have numerical values, it clearly has a natural ordering, where the excellent category (Ex) represents the best quality of material on the exterior and the good category (Gd) represents the second best quality of material on the exterior. In the feature engineering step, we will discuss how we can encode this type of variable for our future model building step.

Continuous variable

We have so far looked at two types of categorical variables in our dataset. However, there is another type of variable in the dataset; the continuous variable. Unlike categorical variables, continuous variables have no limited number of values they can take. For example, square footage for basement area of a house can be any positive number. A house can have a 0 square foot basement area (or no basement) or a house can have a 1,000 square feet basement area. The first continuous variable that we are going to look at is `1stFlrSF`, which represents the first floor square feet. The following code shows how we can build a histogram for `1stFlrSF`:

```
// Continuous Variable #1-1: First Floor Square Feet
var firstFloorHistogram = HistogramBox
.Show(
    houseDF.DropSparseRows()["1stFlrSF"].ValuesAll.ToArray(),
    title: "First Floor Square Feet (Continuous)"
)
.SetNumberOfBins(20);
```

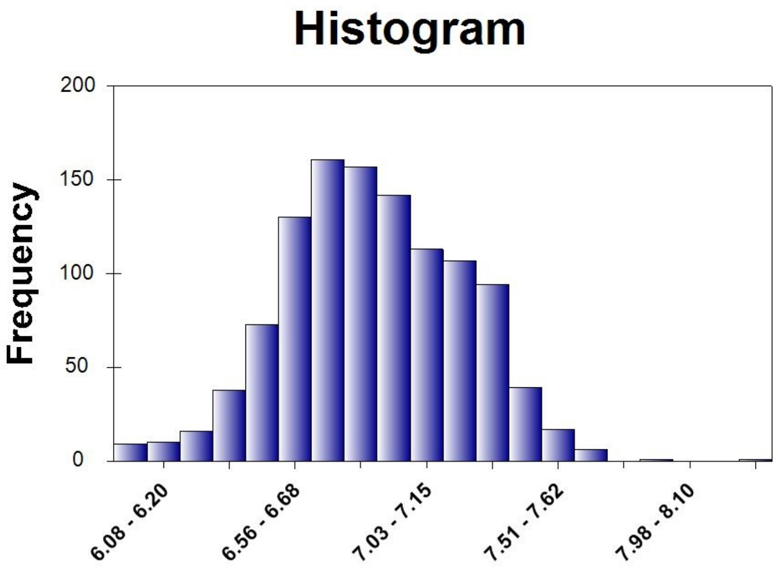
When you run this code, the following histogram will be displayed:



One thing that is obvious from this chart is that it has a long tail in the positive direction, or in other words, the distribution is right skewed. The skewness in the data can adversely affect us when we build ML models. One way to handle this skewness in the dataset is to apply some transformations. One frequently used transformation is the log transformation, where you take log values of a given variable. In this example, the following code shows how we can apply log transformation to the `1stFlrSF` variable and show a histogram for the transformed variable:

```
// Continuous Variable #1-2: Log of First Floor Square Feet
var logFirstFloorHistogram = HistogramBox
.Show(
    houseDF.DropSparseRows()["1stFlrSF"].Log().ValuesAll.ToArray(),
    title: "First Floor Square Feet - Log Transformed (Continuous)"
)
.SetNumberOfBins(20);
```

When you run this code, you will see the following histogram:

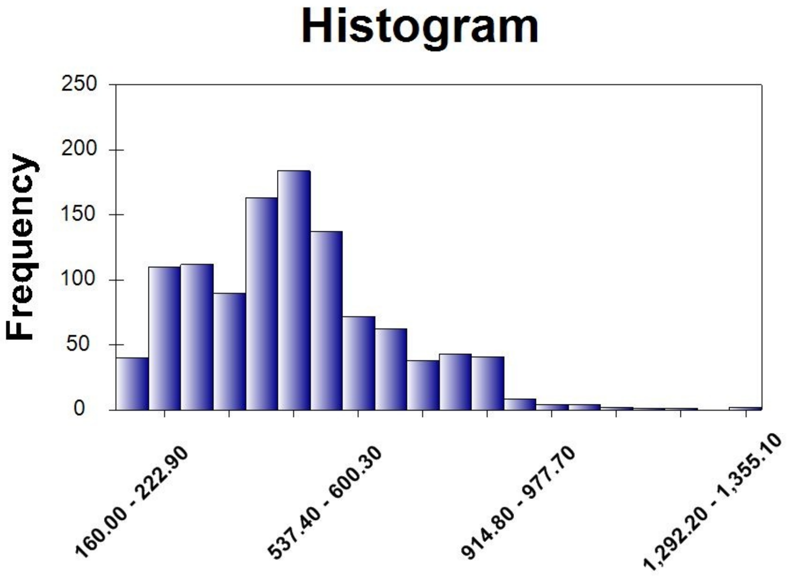


As you can see from this chart, the distribution looks more symmetric and closer to the bell shape that we are familiar with, compared to the previous histogram that we looked at for the same variable. Log transformation is frequently used to handle skewness in the dataset and make the distribution closer to the normal distribution. Let's look at another continuous variable in our dataset. The following code is used to show the distribution of the `GarageArea` variable, which represents the size of the garage in square feet:

```
// Continuous Variable #2-1: Size of garage in square feet
var garageHistogram = HistogramBox
.Show(
    houseDF.DropSparseRows()["GarageArea"].ValuesAll.ToArray(),
    title: "Size of garage in square feet (Continuous)"
)
```

```
.SetNumberOfBins(20);
```

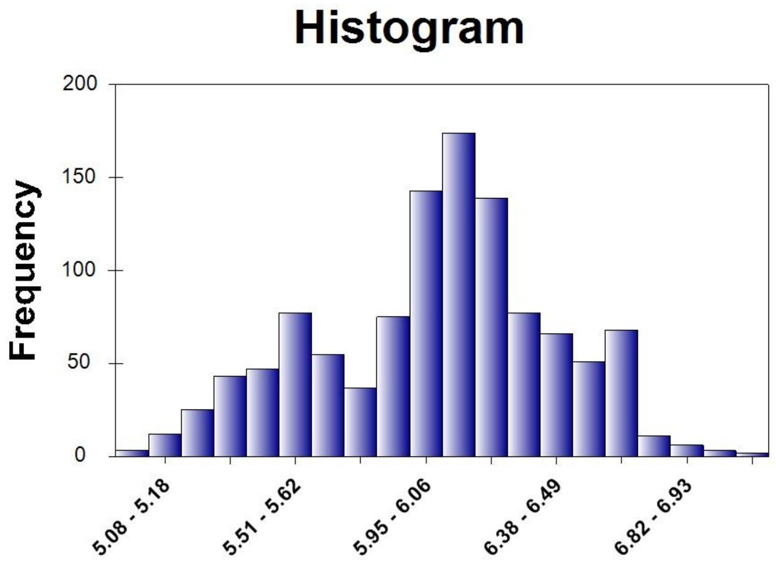
When you run this code, you will see the following histogram:



Similar to the previous case of `1stFlrSF`, it is also right skewed, although it seems the degree of skewness is less than `1stFlrSF`. We used the following code to apply log transformation for the `GarageArea` variable:

```
// Continuous Variable #2-2: Log of Value of miscellaneous feature
var logGarageHistogram = HistogramBox
.Show(
    houseDF.DropSparseRows()["GarageArea"].Log().ValuesAll.ToArray(),
    title: "Size of garage in square feet - Log Transformed (Continuous)"
)
.SetNumberOfBins(20);
```

The following histogram chart will be displayed when you run this code:



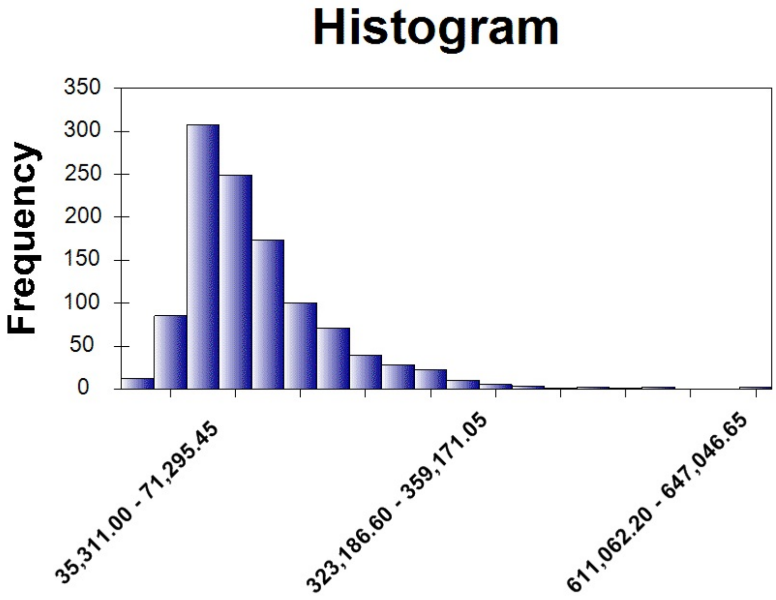
As expected, the distribution looks closer to the normal distribution when the log transformation is applied to the variable.

Target variable – sale price

There is one last variable we need to take a look at before we move onto the feature engineering step; the target variable. In this fair value of a house project, our target variable for predictions is `SalePrice`, which represents the final sale price in U.S. dollar amounts for each residential home sold in Ames, Iowa, U.S.A. from 2006 to 2010. Since the sale price can take any positive numerical value, it is a continuous variable. Let's first look at how we built a histogram for the sale price variable:

```
// Target Variable: Sale Price
var salePriceHistogram = HistogramBox
.Show(
houseDF.DropSparseRows()["SalePrice"].ValuesAll.ToArray(),
title: "Sale Price (Continuous)"
)
.SetNumberOfBins(20);
```

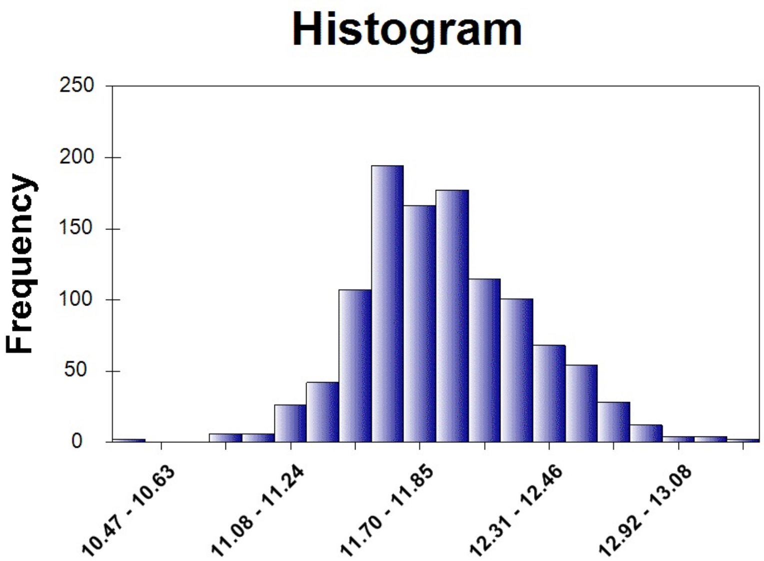
When you run this code, the following histogram chart will be shown:



Similar to the previous cases of continuous variables, the distribution of *SalePrice* has a long right tail and it's heavily skewed to the right. This skewness often adversely affects the regression models, as some of those models, such as the linear regression model, assume that variables are normally distributed. As discussed previously, we can fix this issue by applying log transformation. The following code shows how we log transformed the sale price variable and built a histogram chart:

```
// Target Variable: Sale Price - Log Transformed
var logSalePriceHistogram = HistogramBox
.Show(
houseDF.DropSparseRows()["SalePrice"].Log().ValuesAll.ToArray(),
title: "Sale Price - Log Transformed (Continuous)"
)
.SetNumberOfBins(20);
```

When you run this code, you will see the following histogram for the log-transformed sale price variable:



As expected, the distribution of the `SalePrice` variable looks much closer to the normal distribution. We are going to use this log-transformed `SalePrice` variable as the target variable for our future model building steps.

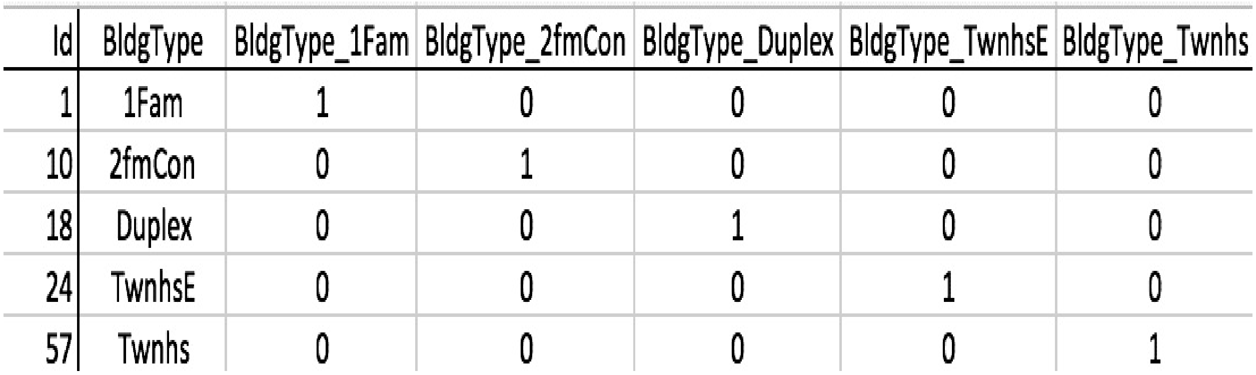
The full code for this data analysis step can be found at this link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.5/DataAnalyzer.cs.

Feature engineering and encoding

Now that we have looked at our dataset and the distributions of the categorical, continuous, and target variables, let's start building features for our ML models. As we discussed previously, categorical variables in our dataset have certain string values to represent each type of variable. However, as it might already be clear to you, we cannot use string types to train our ML models. All the values of variables need to be numerical to be able to used for fitting the models. One way to handle categorical variables with multiple types or categories is to create dummy variables.

Dummy variables

A dummy variable is a variable that takes a value of 0 or 1 to indicate whether a given category or type exists or not. For example, in the case of `BldgType` variable, where it has the five different categories `1Fam`, `2FmCon`, `Duplx`, `TwnhsE`, and `Twnhs`, we will create five dummy variables, where each dummy variable represents the existence or absence of each of those five categories in a given record. The following shows an example of how dummy variable encoding works:



Id	BldgType	BldgType_1Fam	BldgType_2fmCon	BldgType_Duplex	BldgType_TwnhsE	BldgType_Twnhs
1	1Fam	1	0	0	0	0
10	2fmCon	0	1	0	0	0
18	Duplex	0	0	1	0	0
24	TwnhsE	0	0	0	1	0
57	Twnhs	0	0	0	0	1

As you can see from this example, the absence and existence of each category of the building types is encoded into a separate dummy variable as `0` or `1`. For example, for the record with the ID `1`, the building type is `1Fam` and this is encoded with the value 1 for the new variable, `BldgType_1Fam`, and 0 for the other four new variables, `BldgType_2fmCon`, `BldgType_Duplex`, `BldgType_TwnhsE`, and `BldgType_Twnhs`. On the other hand, for the record with the ID `10`, the building type is `2fmCon` and this is encoded with the value 1 for the variable `BldgType_2fmCon` and 0 for the other four new variables, `BldgType_1Fam`, `BldgType_Duplex`, `BldgType_TwnhsE`, and `BldgType_Twnhs`.

For this chapter, we created dummy variables for the following list of categorical variables:

```
string[] categoricalVars = new string[]
{
    "Alley", "BldgType", "BsmtCond", "BsmtExposure", "BsmtFinType1", "BsmtFinType2",
    "BsmtQual", "CentralAir", "Condition1", "Condition2", "Electrical", "ExterCond",
    "Exterior1st", "Exterior2nd", "ExterQual", "Fence", "FireplaceQu", "Foundation",
    "Functional", "GarageCond", "GarageFinish", "GarageQual", "GarageType",
    "Heating", "HeatingQC", "HouseStyle", "KitchenQual", "LandContour", "LandSlope",
    "LotConfig", "LotShape", "MasVnrType", "MiscFeature", "MSSubClass", "MSZoning",
    "Neighborhood", "PavedDrive", "PoolQC", "RoofMatl", "RoofStyle",
    "SaleCondition", "SaleType", "Street", "Utilities"
};
```

The following code shows a method we wrote to create and encode dummy variables:

```
private static Frame&lt;int, string&gt; CreateCategories(Series&lt;int, string&gt;
rows, string originalColName)
{

    var categoriesByRows = rows.GetAllValues().Select((x, i) =&gt;
    {
        // Encode the categories appeared in each row with 1
        var sb = new SeriesBuilder&lt;string, int&gt;();
        sb.Add(String.Format("{0}_{1}", originalColName, x.Value), 1);

        return KeyValue.Create(i, sb.Series);
    });

    // Create a data frame from the rows we just created
    // And encode missing values with 0
    var categoriesDF = Frame.FromRows(categoriesByRows).FillMissing(0);

    return categoriesDF;
}
```

As you can see from line 8 of this method, we prefix the newly created dummy variables with the original categorical variable's names and append them with each category. For example, `BldgType` variables in the `1Fam` category will be encoded as `BldgType_1Fam`. Then, in line 15 of the `CreateCategories` method, we are encoding all the other values with 0s to indicate the absence of such categories in the given categorical variable.

Feature encoding

Now that we know which categorical variables to encode and have created a method for dummy variable encoding for those categorical variables, it is time to build a data frame with features and their values. Let's first take a look at how we went about creating a features data frame in the following code snippet:

```
var featuresDF = Frame.CreateEmpty&lt;int, string&gt;();

foreach(string col in houseDF.ColumnKeys)
{
if (categoricalVars.Contains(col))
{
var categoryDF = CreateCategories(houseDF.GetColumn&lt;string&gt;(col), col);

foreach (string newCol in categoryDF.ColumnKeys)
{
featuresDF.AddColumn(newCol, categoryDF.GetColumn&lt;int&gt;(newCol));
}
}
else if (col.Equals("SalePrice"))
{
featuresDF.AddColumn(col, houseDF[col]);
featuresDF.AddColumn("Log"+col, houseDF[col].Log());
}
else
{
featuresDF.AddColumn(col, houseDF[col].Select((x, i) =&gt; x.Value.Equals("NA")? 0.0: (double) x.Value));
}
}
```

As you can see from this code snippet, we are starting with an empty Deedle data frame, `featuresDF` (in line 1), and start adding in features one by one. For those categorical variables for which we are going to create dummy variables, we are calling the encoding method, `CreateCategories`, that we wrote previously and then adding the newly created dummy variable columns to the `featuresDF` data frame (in lines 5-12). For the `SalePrice` variable, which is the target variable for this project, we are applying log transformation and adding it to the `featuresDF` data frame (in lines 13-17). Lastly, we append all the other continuous variables, after replacing the `NA` values with 0s, to the `featuresDF` data frame (in lines 18-20).

Once we have created and encoded all the features for our model training, we then export this `featuresDF` data frame into a `.csv` file. The following code shows how we export the data frame into a `.csv` file:

```
string outputPath = Path.Combine(dataDirPath, "features.csv");
Console.WriteLine("Writing features DF to {0}", outputPath);
featuresDF.SaveCsv(outputPath);
```

We now have all the necessary features that we can use to start building machine learning models to predict fair values of houses. The full code for feature encoding and engineering can be found in this link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.5/FeatureEngineering.cs.

Linear regression versus SVM with kernels

The first thing we need to do before we start training our machine learning models is to split our dataset into train and test sets. In this section, we will split the sample set into train and test sets by randomly sub-selecting and dividing the indexes at a pre-defined proportion. The code we used to split the dataset into train and test sets is as follows:

```
// Split the sample set into train and test sets
double trainProportion = 0.8;

int[] shuffledIndexes = featuresDF.RowKeys.ToArray();
shuffledIndexes.Shuffle();

int trainSetIndexMax = (int)(featuresDF.RowCount * trainProportion);
int[] trainIndexes = shuffledIndexes.Where(i =&gt; i &lt; trainSetIndexMax).ToArray();
int[] testIndexes = shuffledIndexes.Where(i =&gt; i &gt;= trainSetIndexMax).ToArray();

var trainSet = featuresDF.Where(x =&gt; trainIndexes.Contains(x.Key));
var testSet = featuresDF.Where(x =&gt; testIndexes.Contains(x.Key));

Console.WriteLine("\nTrain Set Shape: ({0}, {1})", trainSet.RowCount,
trainSet.ColumnCount);
Console.WriteLine("Test Set Shape: ({0}, {1})", testSet.RowCount, testSet.ColumnCount);
```

You can choose to have different proportions for training and testing; however, in this example, we reserved 80% of our dataset to be used for training and the remaining 20% for testing. In lines 4-5 of the code snippet, we first randomly shuffle the indexes of our dataset. Then, in lines 7-8, we sub-select indexes for the train set and the test set, and in lines 10-11, we split the `featuresDF` data frame that we created in the previous feature engineering and encoding step into train and test sets.

Once we have these train and test data frames ready, we need to filter out unnecessary columns from the data frames, since the train and test data frames currently have values for columns, such as `SalePrice` and `Id`. Then, we will have to cast the two data frames into arrays of double arrays, which will be input to our learning algorithms. The code to filter out unwanted columns from the train and test data frames and to cast the two data frames into arrays of arrays is as follows:

```
string targetVar = "LogSalePrice";
string[] features = featuresDF.ColumnKeys.Where(
    x =&gt; !x.Equals("Id") && !x.Equals(targetVar) && !x.Equals("SalePrice")
).ToArray();

double[][] trainX = BuildJaggedArray(
    trainSet.Columns[features].ToArray2D&lt;double&gt;(),
    trainSet.RowCount,
    features.Length
);
double[][] testX = BuildJaggedArray(
    testSet.Columns[features].ToArray2D&lt;double&gt;(),
    testSet.RowCount,
    features.Length
);

double[] trainY = trainSet[targetVar].ValuesAll.ToArray();
double[] testY = testSet[targetVar].ValuesAll.ToArray();
```

Linear regression

The first ML model we are going to explore for this chapter's housing price prediction project is the linear regression model. You should already be familiar with building linear regression models in C# using the Accord.NET framework. We use the following code to build a linear regression model:

```
Console.WriteLine("\n**** Linear Regression Model ****");
// OLS learning algorithm
var ols = new OrdinaryLeastSquares()
{
    UseIntercept = true,
    IsRobust = true
};

// Fit a linear regression model
MultipleLinearRegression regFit = ols.Learn(
    trainX,
    trainY
);

// in-sample predictions
double[] regInSamplePreds = regFit.Transform(trainX);
// out-of-sample predictions
double[] regOutSamplePreds = regFit.Transform(testX);
```

The only difference between this chapter's linear regression model code and the previous chapter's code is the `IsRobust` parameter to the `OrdinaryLeastSquares` learning algorithm. As the name suggests, it makes the learning algorithm fit a more robust linear regression model, meaning it is less sensitive to outliers. When we have variables that are not normally distributed, as is the case for this project, it often causes problems when fitting a linear regression model as traditional linear regression models are sensitive to outliers from non-normal distributions. Setting this parameter to `true` helps resolve this issue.

Linear SVM

The second learning algorithm we are going to experiment with in this chapter is the linear SVM. The following code shows how we build a linear SVM model:

```
Console.WriteLine("\n**** Linear Support Vector Machine ****");
// Linear SVM Learning Algorithm
var teacher = new LinearRegressionNewtonMethod()
{
Epsilon = 0.5,
Tolerance = 1e-5,
UseComplexityHeuristic = true
};

// Train SVM
var svm = teacher.Learn(trainX, trainY);

// in-sample predictions
double[] linSVMInSamplePreds = svm.Score(trainX);
// out-of-sample predictions
double[] linSVMOutSamplePreds = svm.Score(testX);
```

As you might have noticed, and similar to the previous chapter, we used `LinearRegressionNewtonMethod` as a learning algorithm to fit a linear SVM.

SVM with a polynomial kernel

The next model we are going to experiment with is an SVM with a polynomial kernel. We will not go into too much detail about the kernel methods, but simply put, kernels are functions of input feature variables that can transform and project the original variables into a new feature space that is more linearly separable. The polynomial kernel looks at the combinations of input features, on top of the original input features. These combinations of input feature variables are often called **interaction variables** in regression analysis. Using different kernel methods will make SVM models learn and behave differently with the same dataset.

The following code shows how you can build a SVM model with a polynomial kernel:

```
Console.WriteLine("\n**** Support Vector Machine with a Polynomial Kernel ****");
// SVM with Polynomial Kernel
var polySVMLearner = new FanChenLinSupportVectorRegression&lt;Polynomial&gt;()
{
    Epsilon = 0.1,
    Tolerance = 1e-5,
    UseKernelEstimation = true,
    UseComplexityHeuristic = true,
    Kernel = new Polynomial(3)
};

// Train SVM with Polynomial Kernel
var polySvm = polySVMLearner.Learn(trainX, trainY);

// in-sample predictions
double[] polySVMInSamplePreds = polySvm.Score(trainX);
// out-of-sample predictions
double[] polySVMOutSamplePreds = polySvm.Score(testX);
```

We are using the `FanChenLinSupportVectorRegression` learning algorithm for a support vector machine with a polynomial kernel. In this example, we used a degree 3 polynomial, but you can experiment with different degrees. However, the higher the degrees are, the more likely it is to overfit to the training data. So, you will have to take cautious steps when you are using high degree polynomial kernels.

SVM with a Gaussian kernel

Another commonly used kernel method is the Gaussian kernel. Simply put, the Gaussian kernel looks at the distance between the input feature variables and results in higher values for close or similar features and lower values for more distanced features. The Gaussian kernel can help transform and project a linearly inseparable dataset into a more linearly separable feature space and can improve the model performances.

The following code shows how you can build a SVM model with a Gaussian kernel:

```
Console.WriteLine("\n**** Support Vector Machine with a Gaussian Kernel ****");
// SVM with Gaussian Kernel
var gaussianSVMLearner = new FanChenLinSupportVectorRegression&lt;Gaussian&gt;()
{
    Epsilon = 0.1,
    Tolerance = 1e-5,
    Complexity = 1e-4,
    UseKernelEstimation = true,
    Kernel = new Gaussian()
};

// Train SVM with Gaussian Kernel
var gaussianSvm = gaussianSVMLearner.Learn(trainX, trainY);

// in-sample predictions
double[] guassianSVMInSamplePreds = gaussianSvm.Score(trainX);
// out-of-sample predictions
double[] guassianSVMOutSamplePreds = gaussianSvm.Score(testX);
```

Similar to the case of the polynomial kernel, we used the `FanChenLinSupportVectorRegression` learning algorithm, but replaced the kernel with the `Gaussian` method.

We have discussed how we can use different kernel methods for SVMs so far. We will now compare the performances of these models on the housing price dataset. You can find the full code we used for building and evaluating models at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.5/Modeling.cs`.

Model validations

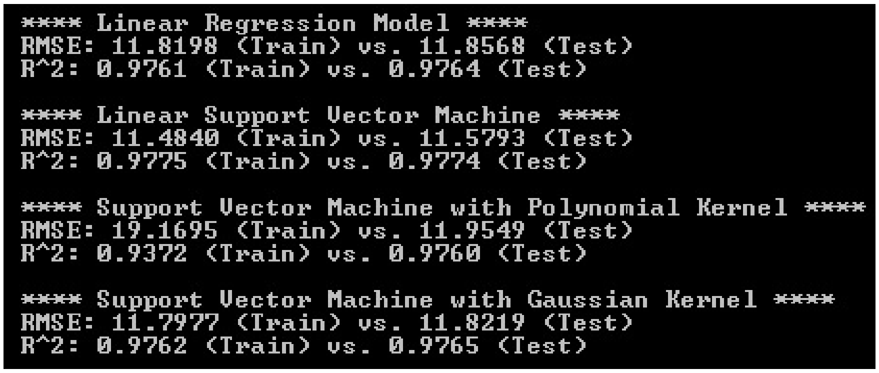
Before we start looking into the performances of the linear regression and SVM models that we built in the previous section, let's refresh our memory on the metrics and the diagnostics plot we discussed in the previous chapter. We are going to look at RMSE, R^2, and a plot of actual versus predicted values to evaluate the performances of our models. The code we are going to use throughout this section for model evaluation is as follows:

```
private static void ValidateModelResults(string modelName, double[] regInSamplePreds,
double[] regOutSamplePreds, double[][] trainX, double[] trainY, double[][] testX,
double[] testY)
{
    // RMSE for in-sample
    double regInSampleRMSE = Math.Sqrt(new SquareLoss(trainX).Loss(regInSamplePreds));
    // RMSE for out-sample
    double regOutSampleRMSE = Math.Sqrt(new SquareLoss(testX).Loss(regOutSamplePreds));

    Console.WriteLine("RMSE: {0:0.0000} (Train) vs. {1:0.0000} (Test)",
regInSampleRMSE, regOutSampleRMSE);

    // R^2 for in-sample
    double regInSampleR2 = new RSquaredLoss(trainX[0].Length,
trainX).Loss(regInSamplePreds);
    // R^2 for out-sample
    double regOutSampleR2 = new RSquaredLoss(testX[0].Length,
testX).Loss(regOutSamplePreds);

    Console.WriteLine("R^2: {0:0.0000} (Train) vs. {1:0.0000} (Test)", regInSampleR2,
regOutSampleR2);

    // Scatter Plot of expected and actual
    var scatterplot = ScatterplotBox.Show(
        String.Format("Actual vs. Prediction ({0})", modelName), testY,
regOutSamplePreds
    );

}
```

The way we use this method for our models is as follows:

```
ValidateModelResults("Linear Regression", regInSamplePreds, regOutSamplePreds, trainX,
trainY, testX, testY);
ValidateModelResults("Linear SVM", linSVMInSamplePreds, linSVMOutSamplePreds, trainX,
trainY, testX, testY);
ValidateModelResults("Polynomial SVM", polySVMInSamplePreds, polySVMOutSamplePreds,
trainX, trainY, testX, testY);
ValidateModelResults("Guassian SVM", guassianSVMInSamplePreds,
guassianSVMOutSamplePreds, trainX, trainY, testX, testY);
```

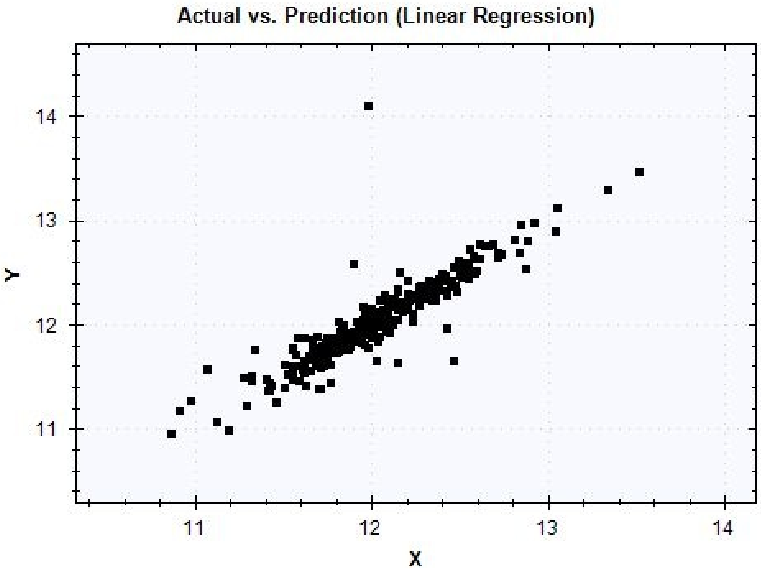
As you can see from this code snippet, we pass the in-sample and out-of-sample predictions by the models, along with the train and test sets, onto the `ValidateModelResults` method. When you run this code, you will see the following output on your console:



```
**** Linear Regression Model ****
RMSE: 11.8198 (Train) vs. 11.8568 (Test)
R^2: 0.9761 (Train) vs. 0.9764 (Test)

**** Linear Support Vector Machine ****
RMSE: 11.4840 (Train) vs. 11.5793 (Test)
R^2: 0.9775 (Train) vs. 0.9774 (Test)

**** Support Vector Machine with Polynomial Kernel ****
RMSE: 19.1695 (Train) vs. 11.9549 (Test)
R^2: 0.9372 (Train) vs. 0.9760 (Test)

**** Support Vector Machine with Gaussian Kernel ****
RMSE: 11.7977 (Train) vs. 11.8219 (Test)
R^2: 0.9762 (Train) vs. 0.9765 (Test)
```

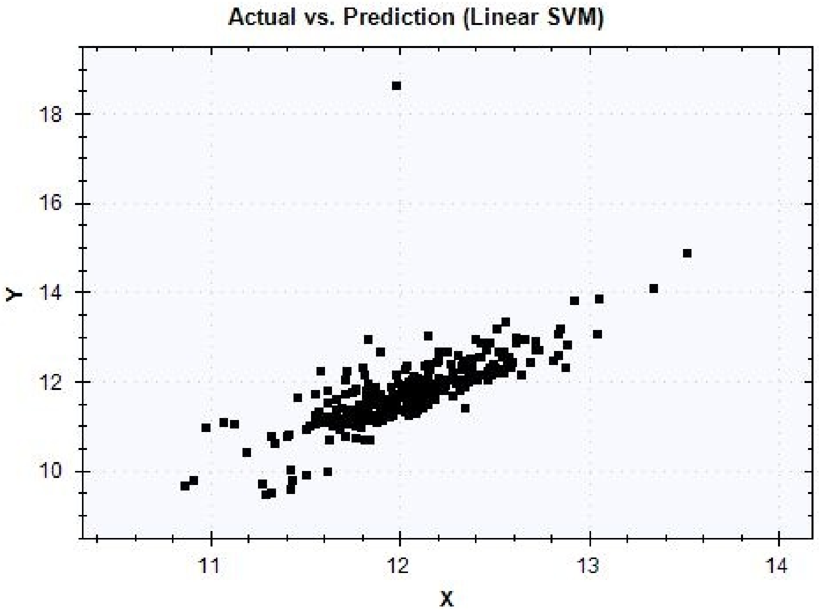
When looking at the values of the goodness of fit, R^2, and the RMSE values, the linear SVM model seems to have the best fit to the dataset, and the SVM model with the Gaussian kernel seems to have the second best fit to the dataset. Looking at this output, the SVM model with the polynomial kernel does not seem to work well for predicting the fair values of house prices. Now, let's look at the diagnostic plots to evaluate how well our models predict the house prices.

The following plot shows the diagnostic plot for the linear regression model:



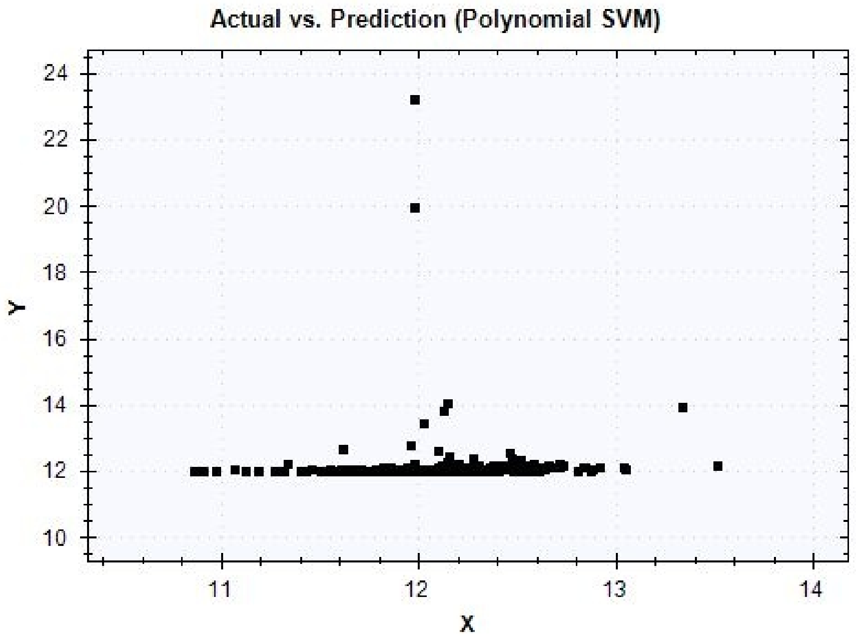
This diagnostic plot for the linear regression model looks good. Most of the points seem to be aligned on a diagonal line, which suggests that the linear regression model's predictions are well aligned with the actual values.

The following plot shows the diagnostic plot for the linear SVM model:



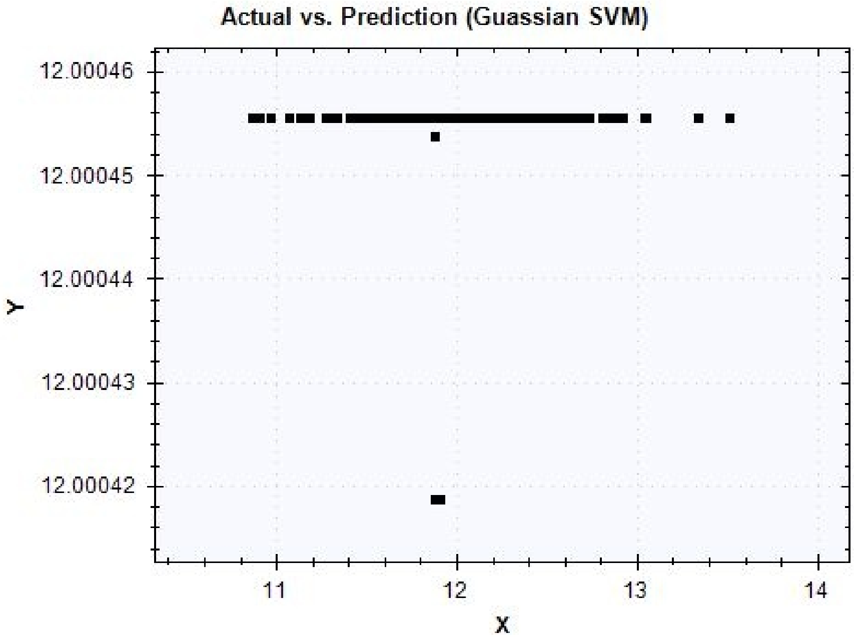
As expected from the previous R^2 metrics value, the goodness of fit for the linear SVM model looks good, even though there seems to be one prediction that is far off from the actual value. Most of the points seem to be aligned on a diagonal line, which suggests that the linear SVM model's predictions are well aligned with the actual values.

The following plot shows the diagnostic plot for the SVM model with the polynomial kernel:



This diagnostic plot for the SVM model with the polynomial kernel suggests that the goodness of fit for this model is not so good. Most of the predictions lie on a straight line at around 12. This is well aligned with the other metrics, where we have seen that RMSE and R^2 measures were the worst among the four models we tried.

The following plot shows the diagnostic plot for the SVM model with the Gaussian kernel:



This diagnostic plot result for the SVM model with the Gaussian kernel is rather surprising. From the RMSE and R^2 measures, we expected the model fit using SVM with Gaussian kernel will be good. However, most of the predictions by this model are on a straight line, without showing any patterns of a diagonal line. Looking at this diagnostic plot, we cannot conclude that the model fit for the SVM model with the Gaussian kernel is good, even though the R^2 metrics showed a strong positive sign of the goodness of model fit.

By looking at both the metrics numbers and the diagnostic plots, we can conclude that the linear regression model and the linear SVM model seem to work the best for predicting the fair values of house prices. This project shows us a good example of the importance of looking at the diagnostic plots. Looking at and optimizing for single metrics might be tempting, but it is always better to evaluate models with more than one validation metric, and looking at diagnostic plots, such as the plot of actual values against predicted values, is especially helpful for regression models.

Summary

In this chapter, we expanded our knowledge and skills regarding building regression models. We built prediction models using the sale price data of residential homes in Ames, Iowa, U.S.A. Unlike other chapters, we had a more complex dataset, where the variables had mixed types, categorical and continuous. We looked at the categorical variables, where there were no natural orderings (non-ordinal) and where there were natural orderings (ordinal) in the categories. We then looked at continuous variables, whose distributions had long right tails. We also discussed how we can use log transformations on such variables with high skewness in the data to mediate the skewness and make those variables' distributions closer to normal distributions.

We discussed how to handle categorical variables in our dataset. We learned how to create and encode dummy variables for each type of categorical variable. Using these features, we experimented with four different machine learning models—linear regression, linear support vector machine, SVM with a polynomial kernel, and SVM with a Gaussian kernel. We briefly discussed the purpose and usage of kernel methods and how they can be used for linearly inseparable datasets. Using RMSE, R^2, and the plot of the actual values against the predicted values, we evaluated the performances of those four models we built for predicting the fair values of house prices in Ames, Iowa, U.S.A. During our model validation step, we saw a case where the validation metrics results contradict with the diagnostic plots results and we have learned the importance of looking at more than one metric and diagnostic plots to be sure of our model's performance.

In the next chapter, we are going to switch gear again. So far, we have been learning how to use and build supervised learning algorithms. However, in the next chapter, we are going to learn unsupervised learning and more specifically clustering algorithms. We will discuss how to use clustering algorithms to gain insights on the customer segments using an online retail dataset.

Customer Segmentation

In this chapter, we are going to learn about unsupervised learning models and how they can be used to extract insights from the data. Up until now, we have been focusing on supervised learning, where our **machine learning** (**ML**) models have known target variables that they try to predict. We have built classification models for spam email filtering and Twitter sentiment analysis. We have also built regression models for foreign exchange rate forecasting and predicting the fair value of house prices. All of these ML models that we have built so far are supervised learning algorithms, where the models learn to map the given input to expected outcomes. However, there are cases where we are more interested in finding hidden insights and drawing inferences from datasets, and we can use unsupervised learning algorithms for such tasks.

In this chapter, we are going to use an online retail dataset that contains information about the prices and quantities of items that customers bought. We will explore the data by looking at how the distributions of item prices and quantities for purchase orders differ from those of cancel orders. We will also look at how online store activities are spread across different countries. Then, we are going to take this transaction-level data and transform and aggregate it into customer-level data. As we transform this data to have a customer-centric view, we are going to discuss ways to build scale-independent features for unsupervised learning algorithms. With this feature set, we are going to use a k-means clustering algorithm to build customer segments and extract insights on the customer behaviors within each segment. We will introduce a new validation metric, Silhouette Coefficient, to evaluate the clustering results.

In this chapter, we will cover the following topics:

- Problem definition for a customer segmentation project
- Data analysis for an online retail dataset
- Feature engineering and aggregation
- Unsupervised learning using a k-means clustering algorithm
- Clustering model validations using the Silhouette Coefficient

Problem definition

Let's discuss in more detail what problems we are going to solve and build clustering models for. Whether you are trying to send marketing emails to your customers or you simply want to better understand your customers and their behaviors on your online store, you will want to analyze and identify different types and segments of your customers. Some customers might buy lots of items at once (bulk buyers), some might primarily buy expensive or luxury items (luxury product buyers), or some might have bought one or two items and never come back (unengaged customers). Depending on these behavioral patterns, your marketing campaigns should vary. For example, sending out emails with promotions on luxury items is likely to provoke luxury product buyers to log in to the online store and purchase certain items, but such an email campaign is not going to work well for bulk buyers. On the other hand, sending out emails with promotions on items that are frequently bought in bulk, such as pens and notepads for office supplies, is likely to make bulk buyers log in to the online store and place purchase orders, but it might not be attractive for luxury product buyers. By identifying customer segments based on their behavioral patterns and using customized marketing campaigns, you can optimize your marketing channels.

In order to build models for customer segmentation, we are going to use an online retail dataset that contains all the transactions that occurred between Jan. 12th 2010 and Sep. 12th 2011 for a UK-based online retail store. This dataset is available in the UCI Machine Learning Repository and can be downloaded from this link: `http://archive.ics.uci.edu/ml/datasets/online+retail#`. With this data, we are going to build features that contain information about the net revenue, average item price, and average purchase quantity per customer. Using these features, we are going to build a clustering model using a **k-means clustering algorithm** that clusters the customer base into different segments. We will be using **Silhouette Coefficient** metrics to evaluate the quality of the clusters and deduce the optimal number of customer segments to build.

To summarize our problem definition for the customer segmentation project:

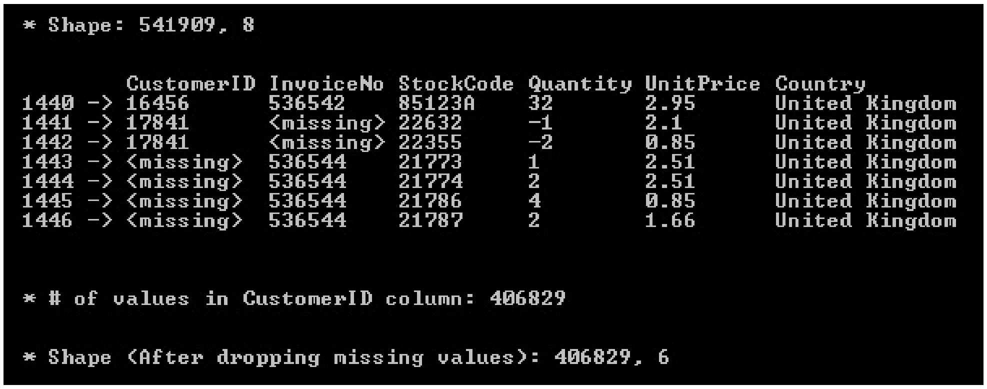
- What is the problem? We need a clustering model that segments customers into different clusters, so that we can understand and draw insights about the behavioral patterns of the customers better.
- Why is it a problem? There is no one-fits-all marketing campaign that works for all different types of customers. We will need to build custom-tailored marketing campaigns for bulk buyers and luxury product buyers separately. Also, we will have to target unengaged customers differently from the other customer types to have them re-engage with the products. The more customized the marketing messages are, the more likely customers will engage. It will be a big advantage if we have an ML model that clusters our customer base into different segments based on their behavioral patterns on the online store.
- What are some of the approaches to solving this problem? We are going to use the online retail dataset that contains all transactions from 2010 to mid-2011 to aggregate the key features, such as net revenue, average unit price, and average purchase quantity for each customer. Then, we will use a k-means clustering algorithm to build a clustering model and use the Silhouette Coefficient to evaluate the quality of clusters and choose the optimal number of clusters.
- What are the success criteria? We do not want too many clusters, as this would make it more difficult to explain and understand different patterns of customers. We will use the Silhouette Coefficient score to tell us the best number of clusters to use for customer segmentation.

Data analysis for the online retail dataset

It is now time to look into the dataset. You can follow this link: `http://archive.ics.uci.edu/ml/datasets/online+retail#`, click on the `Data Folder` link in the top left corner, and download the `Online Retail.xlsx` file. You can save the file as a CSV format and load it into a Deedle data frame.

Handling missing values

Since we will be aggregating the transaction data for each customer, we need to check whether there are any missing values in the `CustomerID` column. The following screenshot shows a few records with no `CustomerID`:



```
* Shape: 541909, 8


        CustomerID InvoiceNo StockCode Quantity UnitPrice Country
1440 -> 16456      536542    85123A    32       2.95      United Kingdom
1441 -> 17841      <missing> 22632     -1       2.1       United Kingdom
1442 -> 17841      <missing> 22355     -2       0.85      United Kingdom
1443 -> <missing>  536544    21773     1        2.51      United Kingdom
1444 -> <missing>  536544    21774     2        2.51      United Kingdom
1445 -> <missing>  536544    21786     4        0.85      United Kingdom
1446 -> <missing>  536544    21787     2        1.66      United Kingdom



* # of values in CustomerID column: 406829

* Shape (After dropping missing values): 406829, 6
```

We are going to drop those records with missing values from the `CustomerID`, `Description`, `Quantity`, `UnitPrice`, and `Country` columns. The following code snippet shows how we can drop records with missing values for those columns:

```
// 1. Missing CustomerID Values
ecommerceDF
    .Columns[new string[] { "CustomerID", "InvoiceNo", "StockCode", "Quantity",
"UnitPrice", "Country" }]
    .GetRowsAt(new int[] { 1440, 1441, 1442, 1443, 1444, 1445, 1446 })
    .Print();
Console.WriteLine("\n\n* # of values in CustomerID column: {0}",
ecommerceDF["CustomerID"].ValueCount);

// Drop missing values
ecommerceDF = ecommerceDF
    .Columns[new string[] { "CustomerID", "Description", "Quantity", "UnitPrice",
"Country" }]
    .DropSparseRows();

// Per-Transaction Purchase Amount = Quantity * UnitPrice
ecommerceDF.AddColumn("Amount", ecommerceDF["Quantity"] * ecommerceDF["UnitPrice"]);

Console.WriteLine("\n\n* Shape (After dropping missing values): {0}, {1}\n",
ecommerceDF.RowCount, ecommerceDF.ColumnCount);
Console.WriteLine("* After dropping missing values and unnecessary columns:");
ecommerceDF.GetRowsAt(new int[] { 0, 1, 2, 3, 4 }).Print();
// Export Data
ecommerceDF.SaveCsv(Path.Combine(dataDirPath, "data-clean.csv"));
```

We use the `DropSparseRows` method of the Deedle data frame to drop all the records with missing values in the columns of our interest. Then, we append the data frame with an additional column `Amount`, which is the total price for the given transaction. We can calculate this value by multiplying the unit price with the quantity.

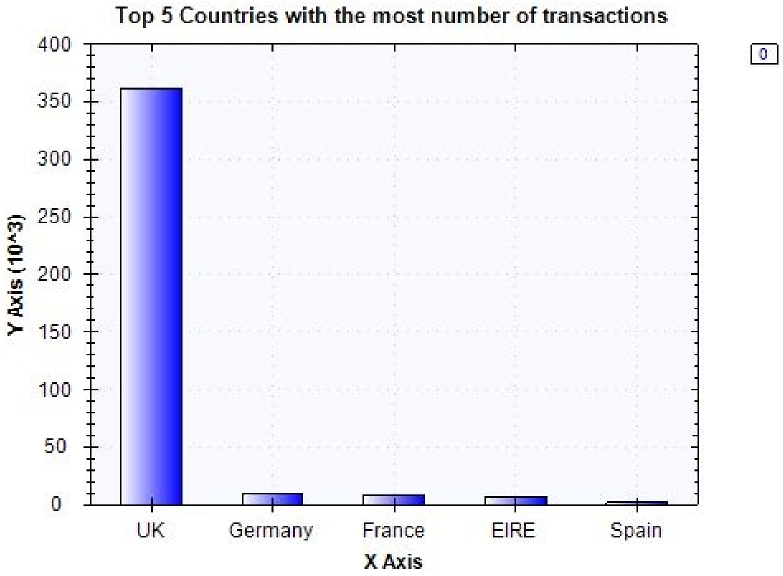
As you can see from the previous image, we had 541,909 records before we dropped the missing values. After dropping the records with missing values from the columns of our interest, the number of records in the data frame ends up being 406,829. Now, we have a data frame that contains the information about `CustomerID`, `Description`, `Quantity`, `UnitPrice`, and `Country` for all the transactions.

Variable distributions

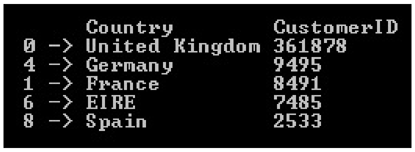
Let's start looking at the distributions in our dataset. First, we will take a look at the top five countries by the volume of transactions. The code we used to aggregate the records by the countries and count the number of transactions that occurred in each country is as follows:

```
// 2. Number of transactions by country
var numTransactionsByCountry = ecommerceDF
    .AggregateRowsBy<string, int>(
        new string[] { "Country" },
        new string[] { "CustomerID" },
        x => x.ValueCount
    ).SortRows("CustomerID");

var top5 = numTransactionsByCountry
    .GetRowsAt(new int[] {
        numTransactionsByCountry.RowCount-1, numTransactionsByCountry.RowCount-2,
        numTransactionsByCountry.RowCount-3, numTransactionsByCountry.RowCount-4,
        numTransactionsByCountry.RowCount-5 });
top5.Print();

var topTransactionByCountryBarChart = DataBarBox.Show(
    top5.GetColumn<string>("Country").Values.ToArray().Select(x => x.Equals("United
Kingdom") ? "UK" : x),
    top5["CustomerID"].Values.ToArray()
);
topTransactionByCountryBarChart.SetTitle(
    "Top 5 Countries with the most number of transactions"
 );
```

As you can see from this code snippet, we are using the `AggregateRowsBy` method in the Deedle data frame to group the records by country and count the total number of transactions for each country. Then, we sort the resulting data frame using the `SortRows` method and take the top five countries. When you run this code, you will see the following bar chart:



The number of transactions for each of the top five countries looks as follows:



```
       Country          CustomerID
0 -> United Kingdom 361878
4 -> Germany        9495
1 -> France         8491
6 -> EIRE           7485
8 -> Spain          2533
```

As expected, the largest number of transactions occurred in the United Kingdom. Germany and France come in as the countries with the second and third most transactions.

Let's start looking at the distributions of the features that we will be using for our clustering model—purchase quantity, unit price, and net amount. We will be looking at these distributions in three ways. First, we will get the overall distribution of each feature, regardless of whether the transaction was for purchase or cancellation. Second, we will take a look at the purchase orders only, excluding the cancel orders. Third, we will look at the distributions for cancel orders only.

The code to get distributions of transaction quantity is as follows:

```
// 3. Per-Transaction Quantity Distributions
Console.WriteLine("\n\n-- Per-Transaction Order Quantity Distribution-- ");
double[] quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Quantity"].ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);

Console.WriteLine("\n\n-- Per-Transaction Purchase-Order Quantity Distribution-- ");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Quantity"].Where(x => x.Value >= 0).ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);

Console.WriteLine("\n\n-- Per-Transaction Cancel-Order Quantity Distribution-- ");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Quantity"].Where(x => x.Value < 0).ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);
```

As in the previous chapter, we are using the `Quantiles` method to compute `quartiles` —min, 25% percentile, median, 75% percentile, and max. Once we get the overall distribution of order quantities per transaction, we then look at the distribution for purchase orders and cancel orders. In our dataset, cancel orders are encoded with negative numbers in the `Quantity` column. In order to separate cancel orders from purchase orders, we can simply filter out positive and negative quantities from our data fame as in the following code:

```
// Filtering out cancel orders to get purchase orders only
ecommerceDF["Quantity"].Where(x => x.Value >= 0)
// Filtering out purchase orders to get cancel orders only
ecommerceDF["Quantity"].Where(x => x.Value < 0)
```

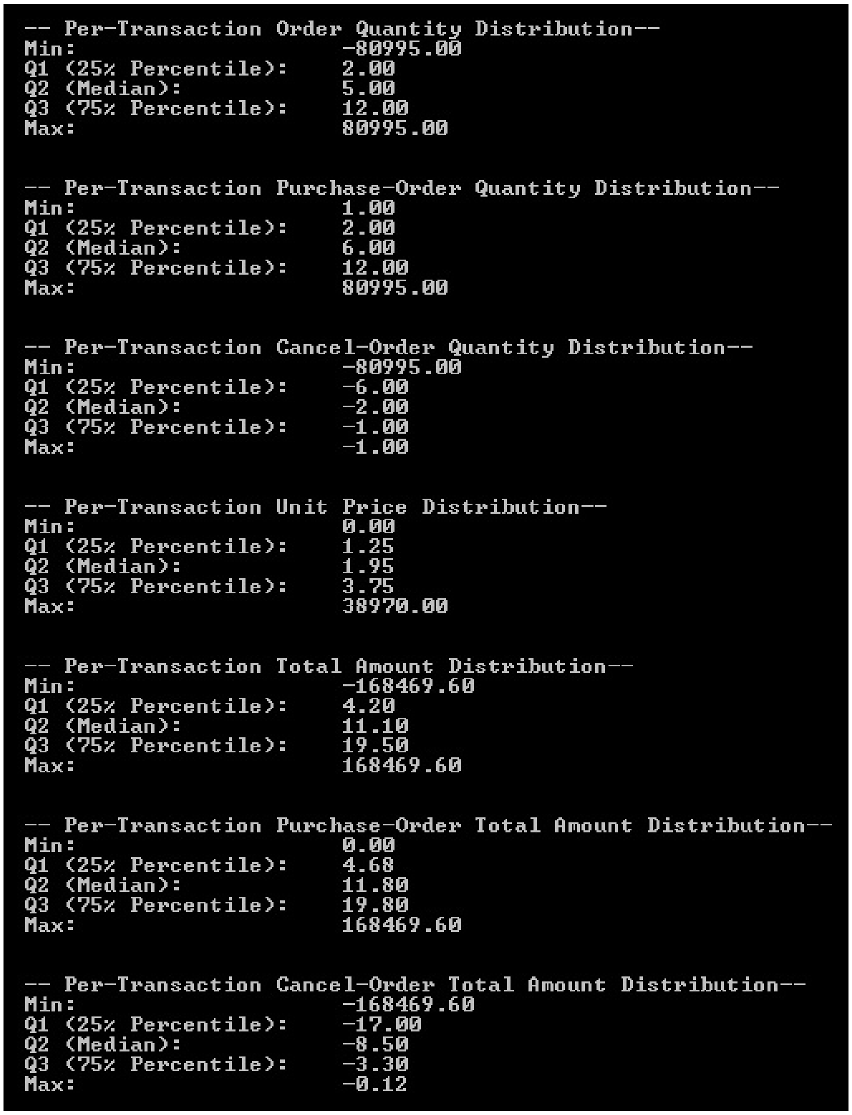
In order to get the `quartiles` of per-transaction unit prices, we use the following code:

```
// 4. Per-Transaction Unit Price Distributions
Console.WriteLine("\n\n-- Per-Transaction Unit Price Distribution-- ");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["UnitPrice"].ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);
```

Similarly, we can compute the `quartiles` of the per-transaction total amount using the following code:

```
// 5. Per-Transaction Purchase Price Distributions
Console.WriteLine("\n\n-- Per-Transaction Total Amount Distribution-- ");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Amount"].ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);

Console.WriteLine("\n\n-- Per-Transaction Purchase-Order Total Amount Distribution--
");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Amount"].Where(x => x.Value >= 0).ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);

Console.WriteLine("\n\n-- Per-Transaction Cancel-Order Total Amount Distribution-- ");
quantiles = Accord.Statistics.Measures.Quantiles(
    ecommerceDF["Amount"].Where(x => x.Value < 0).ValuesAll.ToArray(),
    new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
);
Console.WriteLine(
    "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
    quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
);
```

When you run the code, you will see the following output for the distributions of per-transaction order quantity, unit price, and total amount:



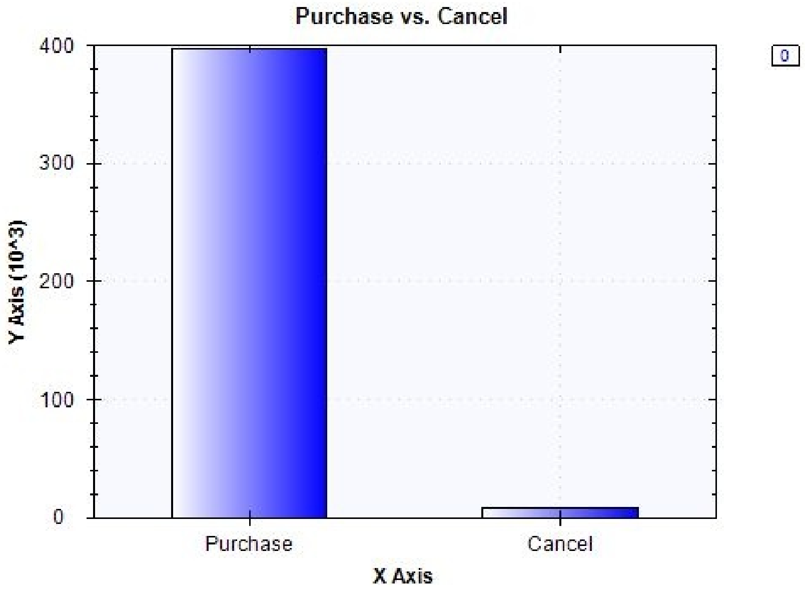
```
-- Per-Transaction Order Quantity Distribution--
Min:                        -80995.00
Q1 (25% Percentile):        2.00
Q2 (Median):                5.00
Q3 (75% Percentile):        12.00
Max:                        80995.00


-- Per-Transaction Purchase-Order Quantity Distribution--
Min:                        1.00
Q1 (25% Percentile):        2.00
Q2 (Median):                6.00
Q3 (75% Percentile):        12.00
Max:                        80995.00


-- Per-Transaction Cancel-Order Quantity Distribution--
Min:                        -80995.00
Q1 (25% Percentile):        -6.00
Q2 (Median):                -2.00
Q3 (75% Percentile):        -1.00
Max:                        -1.00


-- Per-Transaction Unit Price Distribution--
Min:                        0.00
Q1 (25% Percentile):        1.25
Q2 (Median):                1.95
Q3 (75% Percentile):        3.75
Max:                        38970.00


-- Per-Transaction Total Amount Distribution--
Min:                        -168469.60
Q1 (25% Percentile):        4.20
Q2 (Median):                11.10
Q3 (75% Percentile):        19.50
Max:                        168469.60


-- Per-Transaction Purchase-Order Total Amount Distribution--
Min:                        0.00
Q1 (25% Percentile):        4.68
Q2 (Median):                11.80
Q3 (75% Percentile):        19.80
Max:                        168469.60


-- Per-Transaction Cancel-Order Total Amount Distribution--
Min:                        -168469.60
Q1 (25% Percentile):        -17.00
Q2 (Median):                -8.50
Q3 (75% Percentile):        -3.30
Max:                        -0.12
```

If you look at the distribution of the overall order quantities in this output, you will notice that from the first quartile (25% percentile), the quantities are positive. This suggests that there are far less cancel orders than purchase orders, which is actually a good thing for an online retail store. Let's look at how the purchase orders and cancel orders are divided in our dataset.

Using the following code, you can draw a bar chart to compare the number of purchase orders against cancel orders:

```
// 6. # of Purchase vs. Cancelled Transactions
var purchaseVSCancelBarChart = DataBarBox.Show(
    new string[] { "Purchase", "Cancel" },
    new double[] {
        ecommerceDF["Quantity"].Where(x => x.Value >= 0).ValueCount ,
        ecommerceDF["Quantity"].Where(x => x.Value < 0).ValueCount
    }
);
purchaseVSCancelBarChart.SetTitle(
    "Purchase vs. Cancel"
 );
```

When you run this code, you will see the following bar chart:



As expected and shown in the previous distribution output, the number of cancel orders is much less than the number of purchase orders. With these analysis results, we are going to start building features for our clustering model for customer segmentation in the next section.

The full code for this data analysis step can be found by following this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.6/DataAnalyzer.cs`.

Feature engineering and data aggregation

The records in the dataset we have now represent individual transactions. However, we want to build a clustering model that clusters customers into different segments. In order to do that, we need to transform and aggregate our data by customer. In other words, we will need to group our data by `CustomerID` and `aggregate` all the transactions that belong to each customer by summing, counting, or taking averages of the values. Let's look at an example first. The following code groups the transaction-level data by `CustomerID` and computes the net revenue, total number of transactions, total number of cancel orders, average unit price, and average order quantity:

```
// 1. Net Revenue per Customer
var revPerCustomerDF = ecommerceDF.AggregateRowsBy<double, double>(
    new string[] { "CustomerID" },
    new string[] { "Amount" },
    x => x.Sum()
);
// 2. # of Total Transactions per Customer
var numTransactionsPerCustomerDF = ecommerceDF.AggregateRowsBy<double, double>(
    new string[] { "CustomerID" },
    new string[] { "Quantity" },
    x => x.ValueCount
);
// 3. # of Cancelled Transactions per Customer
var numCancelledPerCustomerDF = ecommerceDF.AggregateRowsBy<double, double>(
    new string[] { "CustomerID" },
    new string[] { "Quantity" },
    x => x.Select(y => y.Value >= 0 ? 0.0 : 1.0).Sum()
);
// 4. Average UnitPrice per Customer
var avgUnitPricePerCustomerDF = ecommerceDF.AggregateRowsBy<double, double>(
    new string[] { "CustomerID" },
    new string[] { "UnitPrice" },
    x => x.Sum() / x.ValueCount
);
// 5. Average Quantity per Customer
var avgQuantityPerCustomerDF = ecommerceDF.AggregateRowsBy<double, double>(
    new string[] { "CustomerID" },
    new string[] { "Quantity" },
    x => x.Sum() / x.ValueCount
);
```

As you may see from this code, we are using the `AggregateRowsBy` method in the Deedle data frame and passing a custom `aggFunc` for each aggregation. In the first example, where we compute the net revenue per customer, we sum all the purchase amounts for each customer. For the second feature, we count the number of transactions to compute the total number of orders for each customer. In order to compute the average order quantity for each customer, we sum up all the order quantities and divide it by the number of transactions. As you can see from this case, the `AggregateRowsBy` method comes in handy when you need to transform and aggregate a data frame with a custom `aggregation` function.

Once we have computed all these features, we need to combine all the data into one place. We created a new empty data frame and added each of these aggregated features as separate columns to the new data frame. The following code shows how we created a features data frame:

```
// Aggregate all results
var featuresDF = Frame.CreateEmpty<int, string>();
featuresDF.AddColumn("CustomerID", revPerCustomerDF.GetColumn<double>("CustomerID"));
featuresDF.AddColumn("Description", ecommerceDF.GetColumn<string>("Description"));
featuresDF.AddColumn("NetRevenue", revPerCustomerDF.GetColumn<double>("Amount"));
featuresDF.AddColumn("NumTransactions", numTransactionsPerCustomerDF.GetColumn<double>
("Quantity"));
featuresDF.AddColumn("NumCancelled", numCancelledPerCustomerDF.GetColumn<double>
("Quantity"));
featuresDF.AddColumn("AvgUnitPrice", avgUnitPricePerCustomerDF.GetColumn<double>
("UnitPrice"));
featuresDF.AddColumn("AvgQuantity", avgQuantityPerCustomerDF.GetColumn<double>
("Quantity"));
featuresDF.AddColumn("PercentageCancelled", featuresDF["NumCancelled"] /
featuresDF["NumTransactions"]);

Console.WriteLine("\n\n* Feature Set:");
featuresDF.Print();
```

As you can see from this code snippet, we created one additional feature, `PercentageCancelled`, while we were appending those aggregated features to the new data frame. The `PercentageCancelled` feature simply holds information about how many of the transactions or orders were cancelled.

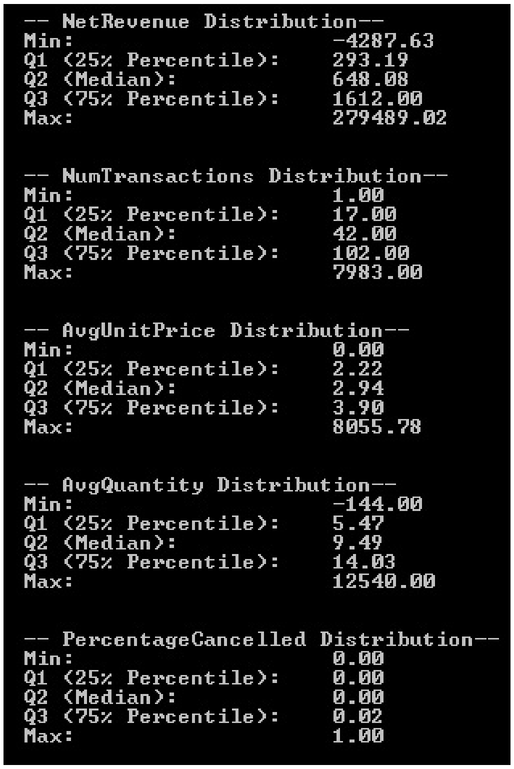
To take a closer look at the distributions of these features, we wrote a helper function that computes the `quartiles` of a given feature and prints out the results. The code for this helper function is as follows:

```
private static void PrintQuartiles(Frame<int, string> df, string colname)
{
    Console.WriteLine("\n\n-- {0} Distribution-- ", colname);
    double[] quantiles = Accord.Statistics.Measures.Quantiles(
        df[colname].ValuesAll.ToArray(),
        new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
    );
    Console.WriteLine(
        "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
        quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
    );
}
```

Using this helper function, `PrintQuartiles`, the following code snippet shows how we computed and displayed `quartiles` for the features we just created:

```
// NetRevenue feature distribution
PrintQuartiles(featuresDF, "NetRevenue");
// NumTransactions feature distribution
PrintQuartiles(featuresDF, "NumTransactions");
// AvgUnitPrice feature distribution
PrintQuartiles(featuresDF, "AvgUnitPrice");
// AvgQuantity feature distribution
PrintQuartiles(featuresDF, "AvgQuantity");
// PercentageCancelled feature distribution
PrintQuartiles(featuresDF, "PercentageCancelled");
```

The output of this code looks like the following:

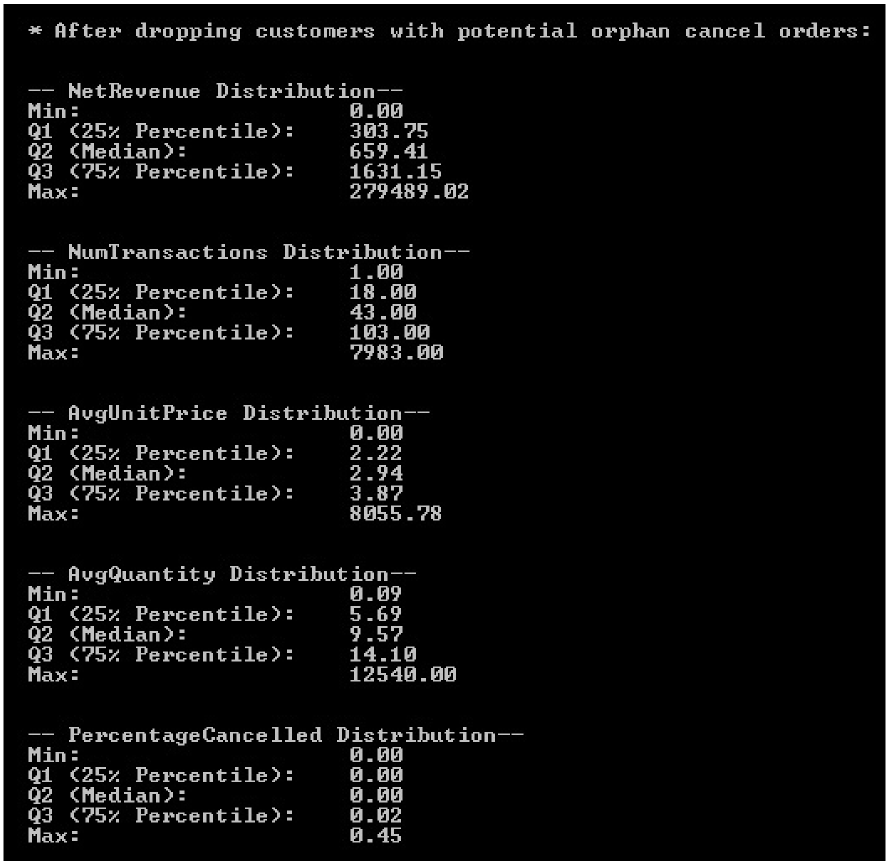


```
-- NetRevenue Distribution--
Min:                        -4287.63
Q1 (25% Percentile):        293.19
Q2 (Median):                648.08
Q3 (75% Percentile):        1612.00
Max:                        279489.02


-- NumTransactions Distribution--
Min:                        1.00
Q1 (25% Percentile):        17.00
Q2 (Median):                42.00
Q3 (75% Percentile):        102.00
Max:                        7983.00


-- AvgUnitPrice Distribution--
Min:                        0.00
Q1 (25% Percentile):        2.22
Q2 (Median):                2.94
Q3 (75% Percentile):        3.90
Max:                        8055.78


-- AvgQuantity Distribution--
Min:                        -144.00
Q1 (25% Percentile):        5.47
Q2 (Median):                9.49
Q3 (75% Percentile):        14.03
Max:                        12540.00


-- PercentageCancelled Distribution--
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.02
Max:                        1.00
```

If you look closely, there is one thing that is concerning. There is a small number of customers that have negative net revenue and negative average quantity. This suggests some customers have more cancel orders than purchase orders. However, this is odd. To cancel an order, there needs to be a purchase order first. This suggests that our dataset is not complete and there are some orphan cancel orders that do not have matching previous purchase orders. Since we cannot go back in time and pull out more data for those customers with orphan cancel orders, the simplest way to handle this problem is to drop those customers with orphan cancel orders. The following code shows some criteria we can use to drop such customers:

```
// 1. Drop Customers with Negative NetRevenue
featuresDF = featuresDF.Rows[
    featuresDF["NetRevenue"].Where(x => x.Value >= 0.0).Keys
];
// 2. Drop Customers with Negative AvgQuantity
featuresDF = featuresDF.Rows[
    featuresDF["AvgQuantity"].Where(x => x.Value >= 0.0).Keys
];
// 3. Drop Customers who have more cancel orders than purchase orders
featuresDF = featuresDF.Rows[
    featuresDF["PercentageCancelled"].Where(x => x.Value < 0.5).Keys
];
```

As you can see from this code snippet, we drop any customer who has a negative net revenue, negative average quantity, and percentage of cancel orders more than 50%. After dropping these customers, the resulting distributions look like the following:



```
* After dropping customers with potential orphan cancel orders:


-- NetRevenue Distribution--
Min:                        0.00
Q1 (25% Percentile):        303.75
Q2 (Median):                659.41
Q3 (75% Percentile):        1631.15
Max:                        279489.02


-- NumTransactions Distribution--
Min:                        1.00
Q1 (25% Percentile):        18.00
Q2 (Median):                43.00
Q3 (75% Percentile):        103.00
Max:                        7983.00


-- AvgUnitPrice Distribution--
Min:                        0.00
Q1 (25% Percentile):        2.22
Q2 (Median):                2.94
Q3 (75% Percentile):        3.87
Max:                        8055.78


-- AvgQuantity Distribution--
Min:                        0.09
Q1 (25% Percentile):        5.69
Q2 (Median):                9.57
Q3 (75% Percentile):        14.10
Max:                        12540.00


-- PercentageCancelled Distribution--
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.02
Max:                        0.45
```

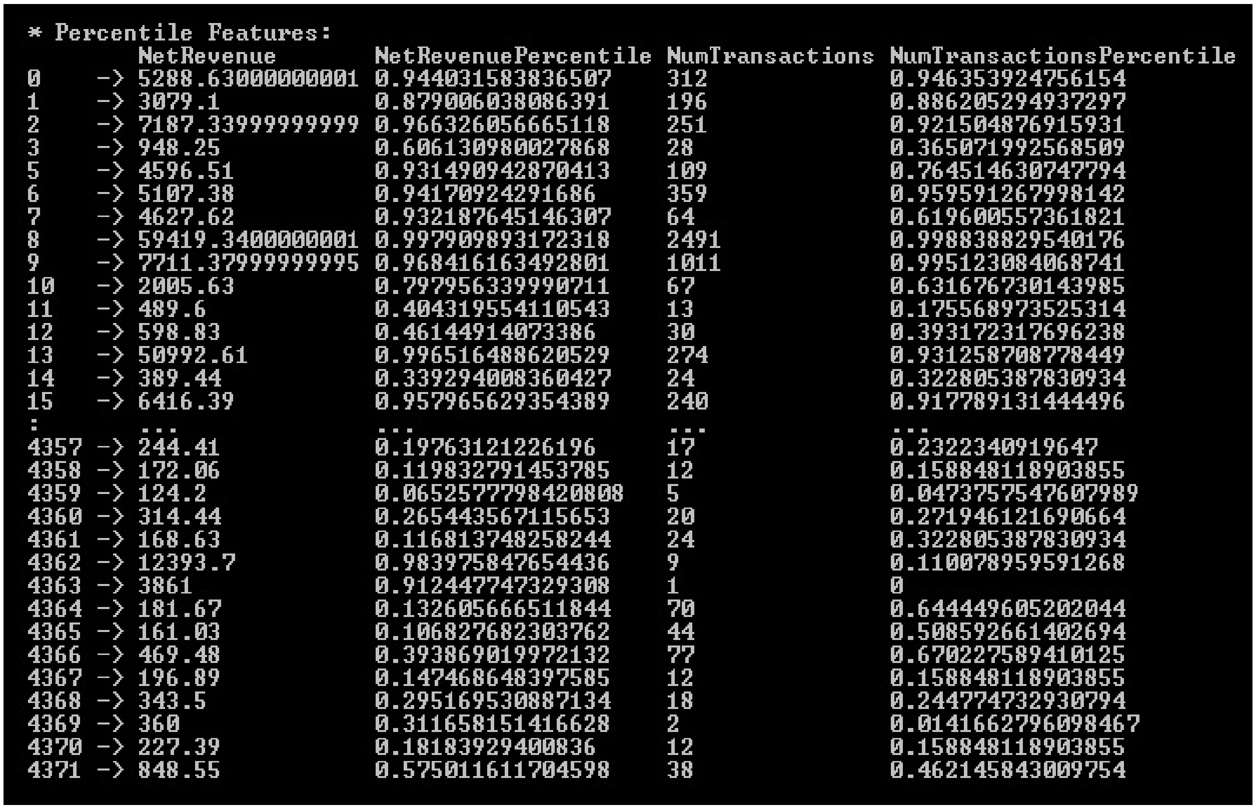
As you can see from these distributions, the scales for each feature are very different. `NetRevenue` rages from 0 to 279,489.02, while `PercentageCancelled` ranges from 0 to 0.45. We are going to transform these features into percentiles, so that we can have all of our features on the same scale of 0 to 1. The following code shows how to compute percentiles for each feature:

```
// Create Percentile Features
featuresDF.AddColumn(
    "NetRevenuePercentile",
    featuresDF["NetRevenue"].Select(
        x => StatsFunctions.PercentileRank(featuresDF["NetRevenue"].Values.ToArray(),
x.Value)
    )
);
featuresDF.AddColumn(
    "NumTransactionsPercentile",
    featuresDF["NumTransactions"].Select(
        x =>
StatsFunctions.PercentileRank(featuresDF["NumTransactions"].Values.ToArray(), x.Value)
    )
);
featuresDF.AddColumn(
    "AvgUnitPricePercentile",
    featuresDF["AvgUnitPrice"].Select(
        x => StatsFunctions.PercentileRank(featuresDF["AvgUnitPrice"].Values.ToArray(),
x.Value)
    )
);
featuresDF.AddColumn(
    "AvgQuantityPercentile",
    featuresDF["AvgQuantity"].Select(
        x => StatsFunctions.PercentileRank(featuresDF["AvgQuantity"].Values.ToArray(),
x.Value)
    )
);
featuresDF.AddColumn(
    "PercentageCancelledPercentile",
    featuresDF["PercentageCancelled"].Select(
        x =>
StatsFunctions.PercentileRank(featuresDF["PercentageCancelled"].Values.ToArray(),
x.Value)
    )
);
Console.WriteLine("\n\n\n* Percentile Features:");
featuresDF.Columns[
    new string[] { "NetRevenue", "NetRevenuePercentile", "NumTransactions",
"NumTransactionsPercentile" }
].Print();
```

As you can notice from this code snippet, we are using the `StatsFunctions.PercentileRank` method, which is part of the `CenterSpace.NMath.Stats` package. You can easily install this package using the following command in the **Package Manager** console:

```
Install-Package CenterSpace.NMath.Stats
```

Using the `StatsFunctions.PercentileRank` method, we can compute the percentile for each record. The following output shows the results for the `NetRevenue` and `NumTransactions` features:



```
* Percentile Features:
        NetRevenue       NetRevenuePercentile NumTransactions NumTransactionsPercentile
0    -> 5288.63000000001 0.944031583836507    312             0.946353924756154
1    -> 3079.1           0.879006038086391    196             0.886205294937297
2    -> 7187.33999999999 0.966326056665118    251             0.921504876915931
3    -> 948.25           0.606130980027868    28              0.365071992568509
5    -> 4596.51          0.931490942870413    109             0.764514630747794
6    -> 5107.38          0.94170924291686     359             0.959591267998142
7    -> 4627.62          0.932187645146307    64              0.619600557361821
8    -> 59419.3400000001 0.997909893172318    2491            0.998838829540176
9    -> 7711.37999999995 0.968416163492801    1011            0.995123084068741
10   -> 2005.63          0.797956339990711    67              0.631676730143985
11   -> 489.6            0.404319554110543    13              0.175568973525314
12   -> 598.83           0.46144914073386     30              0.393172317696238
13   -> 50992.61         0.996516488620529    274             0.931258708778449
14   -> 389.44           0.339294008360427    24              0.322805387830934
15   -> 6416.39          0.957965629354389    240             0.917789131444496
:       ...              ...                  ...             ...
4357 -> 244.41           0.19763121226196     17              0.2322340919647
4358 -> 172.06           0.119832791453785    12              0.158848118903855
4359 -> 124.2            0.0652577798420808   5               0.0473757547607989
4360 -> 314.44           0.265443567115653    20              0.271946121690664
4361 -> 168.63           0.116813748258244    24              0.322805387830934
4362 -> 12393.7          0.983975847654436    9               0.110078959591268
4363 -> 3861             0.912447747329308    1               0
4364 -> 181.67           0.132605666511844    70              0.644449605202044
4365 -> 161.03           0.106827682303762    44              0.508592661402694
4366 -> 469.48           0.393869019972132    77              0.670227589410125
4367 -> 196.89           0.147468648397585    12              0.158848118903855
4368 -> 343.5            0.295169530887134    18              0.244774732930794
4369 -> 360              0.311658151416628    2               0.0141662796098467
4370 -> 227.39           0.18183929400836     12              0.158848118903855
4371 -> 848.55           0.575011611704598    38              0.462145843009754
```

As you can see from this output, instead of a wide range, the values for both features now range between 0 and 1. We will use these percentile features when we build our clustering model in the following section.

The full code for this feature engineering step can be found at this link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.6/FeatureEngineering.cs.

Unsupervised learning – k-means clustering

It is now time to start building our clustering models. In this project, we are going to try clustering customers into different segments based on the following three features: `NetRevenuePercentile`, `AvgUnitPricePercentile`, and `AvgQuantityPercentile`, so that we can analyze the item selections based on the spending habits of the customers. Before we start fitting a k-means clustering algorithm to our feature set, there is an important step we need to take. We need to normalize our features, so that our clustering model does not put more weight on certain features over the others. If variances of features are different, then a clustering algorithm can put more weight on those with small variances and can tend to cluster them together. The following code shows how you can normalize each feature:

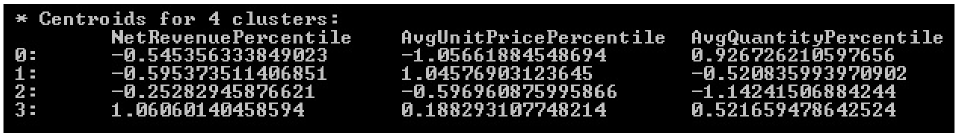
```
string[] features = new string[] { "NetRevenuePercentile", "AvgUnitPricePercentile",
"AvgQuantityPercentile" };
Console.WriteLine("* Features: {0}\n\n", String.Join(", ", features));

var normalizedDf = Frame.CreateEmpty<int, string>();
var average = ecommerceDF.Columns[features].Sum() / ecommerceDF.RowCount;
foreach(string feature in features)
{
    normalizedDf.AddColumn(feature, (ecommerceDF[feature] - average[feature]) /
ecommerceDF[feature].StdDev());
}
```

Now that we have normalized our variables, let's start building clustering models. In order to build a k-means clustering model, we need to know the number of clusters we want in advance. Since we do not know what the best number of clusters is, we are going to try a few different numbers of clusters and rely on the validation metrics, the Silhouette Score, to tell us what the optimal number of clusters is. The following code shows how to build clustering models that use a k-means clustering algorithm:

```
int[] numClusters = new int[] { 4, 5, 6, 7, 8 };
List<string> clusterNames = new List<string>();
List<double> silhouetteScores = new List<double>();
for(int i = 0; i < numClusters.Length; i++)
{
    KMeans kmeans = new KMeans(numClusters[i]);
    KMeansClusterCollection clusters = kmeans.Learn(sampleSet);
    int[] labels = clusters.Decide(sampleSet);

    string colname = String.Format("Cluster-{0}", numClusters[i]);
    clusterNames.Add(colname);

    normalizedDf.AddColumn(colname, labels);
    ecommerceDF.AddColumn(colname, labels);

    Console.WriteLine("\n\n\n##################### {0} ###########################",
colname);

    Console.WriteLine("\n\n* Centroids for {0} clusters:", numClusters[i]);

    PrintCentroidsInfo(clusters.Centroids, features);
    Console.WriteLine("\n");

    VisualizeClusters(normalizedDf, colname, "NetRevenuePercentile",
"AvgUnitPricePercentile");
    VisualizeClusters(normalizedDf, colname, "AvgUnitPricePercentile",
"AvgQuantityPercentile");
    VisualizeClusters(normalizedDf, colname, "NetRevenuePercentile",
"AvgQuantityPercentile");

    for (int j = 0; j < numClusters[i]; j++)
    {
        GetTopNItemsPerCluster(ecommerceDF, j, colname);
    }

    double silhouetteScore = CalculateSilhouetteScore(normalizedDf, features,
numClusters[i], colname);
    Console.WriteLine("\n\n* Silhouette Score: {0}",
silhouetteScore.ToString("0.0000"));

    silhouetteScores.Add(silhouetteScore);

Console.WriteLine("\n\n###############################################################\n\
}
```

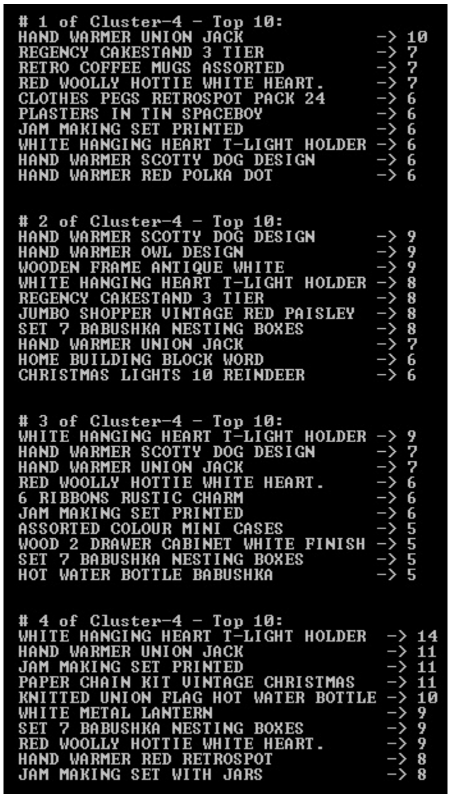
As you can see from this code snippet, we are going to try building clustering models with `4`, `5`, `6`, `7`, and `8` clusters. We can instantiate a k-means clustering algorithm object using the `KMeans` class in the `Accord.NET` framework. Using the `Learn` method, we can train a k-means clustering model with the feature set we have. Then, we can use the `Decide` method to get the cluster labels for each record.

When you run this code, it will output the centroids for each cluster. The following is an output of cluster centroids from a 4-cluster clustering model:



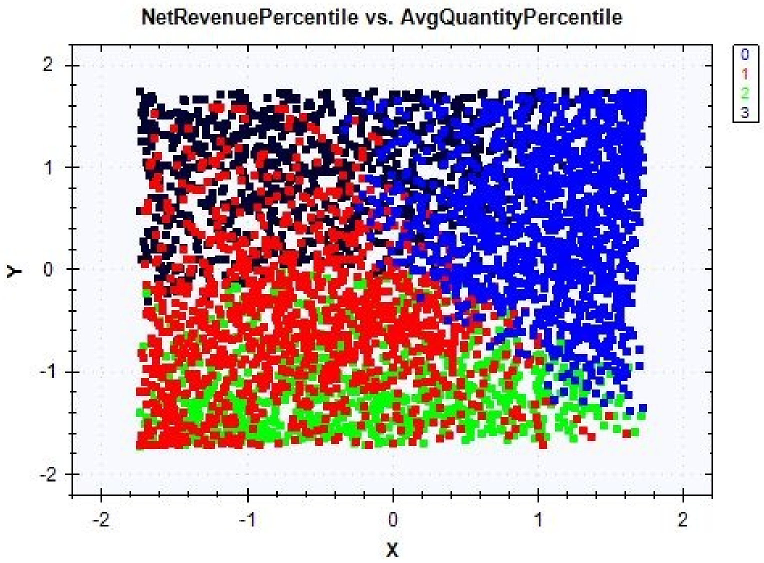
```
* Centroids for 4 clusters:
        NetRevenuePercentile    AvgUnitPricePercentile  AvgQuantityPercentile
0:      -0.545356333849023      -1.05661884548694       0.926726210597656
1:      -0.595373511406851      1.04576903123645        -0.520835993970902
2:      -0.25282945876621       -0.596960875995866      -1.14241506884244
3:      1.06060140458594        0.188293107748214       0.521659478642524
```

As you can see from this output, the cluster with label 3 is a cluster of customers who have high net revenue, middle-high average unit price, and middle-high average quantity. So, these customers are high value customers who bring in the most revenue and buy items with prices above average in above-average quantities. In contrast, the cluster labeled as 1 is a cluster of customers who have low net revenue, high average unit price and middle-low average quantity. So, these customers buy expensive items in average quantities and do not bring in that much revenue for the online store. As you may notice from this example, you can already see some patterns among different clusters. Let's now look at which customers in each segment buy the most. The following is the top 10 items bought for each segment of the 4-cluster clustering model:

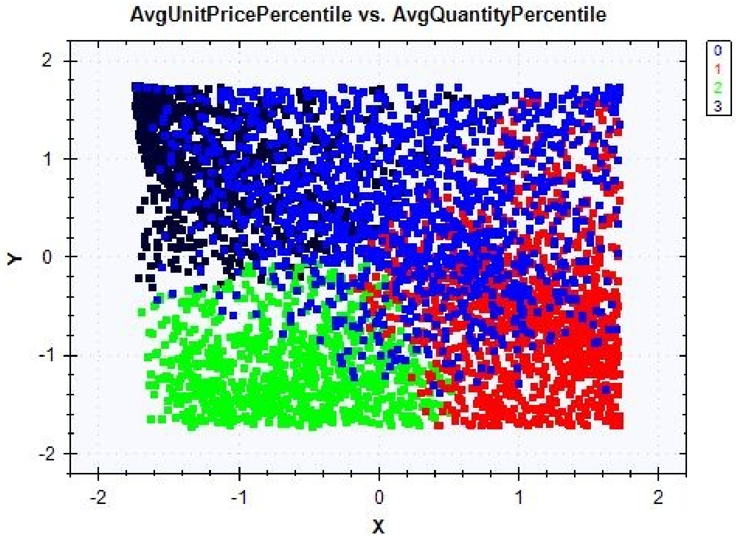


```
# 1 of Cluster-4 - Top 10:
HAND WARMER UNION JACK              -> 10
REGENCY CAKESTAND 3 TIER            -> 7
RETRO COFFEE MUGS ASSORTED          -> 7
RED WOOLLY HOTTIE WHITE HEART.      -> 7
CLOTHES PEGS RETROSPOT PACK 24      -> 6
PLASTERS IN TIN SPACEBOY            -> 6
JAM MAKING SET PRINTED              -> 6
WHITE HANGING HEART T-LIGHT HOLDER -> 6
HAND WARMER SCOTTY DOG DESIGN       -> 6
HAND WARMER RED POLKA DOT           -> 6


# 2 of Cluster-4 - Top 10:
HAND WARMER SCOTTY DOG DESIGN       -> 9
HAND WARMER OWL DESIGN              -> 9
WOODEN FRAME ANTIQUE WHITE          -> 9
WHITE HANGING HEART T-LIGHT HOLDER -> 8
REGENCY CAKESTAND 3 TIER            -> 8
JUMBO SHOPPER VINTAGE RED PAISLEY   -> 8
SET 7 BABUSHKA NESTING BOXES        -> 8
HAND WARMER UNION JACK              -> 7
HOME BUILDING BLOCK WORD            -> 6
CHRISTMAS LIGHTS 10 REINDEER        -> 6


# 3 of Cluster-4 - Top 10:
WHITE HANGING HEART T-LIGHT HOLDER -> 9
HAND WARMER SCOTTY DOG DESIGN       -> 7
HAND WARMER UNION JACK              -> 7
RED WOOLLY HOTTIE WHITE HEART.      -> 6
6 RIBBONS RUSTIC CHARM              -> 6
JAM MAKING SET PRINTED              -> 6
ASSORTED COLOUR MINI CASES          -> 5
WOOD 2 DRAWER CABINET WHITE FINISH -> 5
SET 7 BABUSHKA NESTING BOXES        -> 5
HOT WATER BOTTLE BABUSHKA           -> 5


# 4 of Cluster-4 - Top 10:
WHITE HANGING HEART T-LIGHT HOLDER   -> 14
HAND WARMER UNION JACK               -> 11
JAM MAKING SET PRINTED               -> 11
PAPER CHAIN KIT VINTAGE CHRISTMAS    -> 11
KNITTED UNION FLAG HOT WATER BOTTLE -> 10
WHITE METAL LANTERN                  -> 9
SET 7 BABUSHKA NESTING BOXES         -> 9
RED WOOLLY HOTTIE WHITE HEART.       -> 9
HAND WARMER RED RETROSPOT            -> 8
JAM MAKING SET WITH JARS             -> 8
```

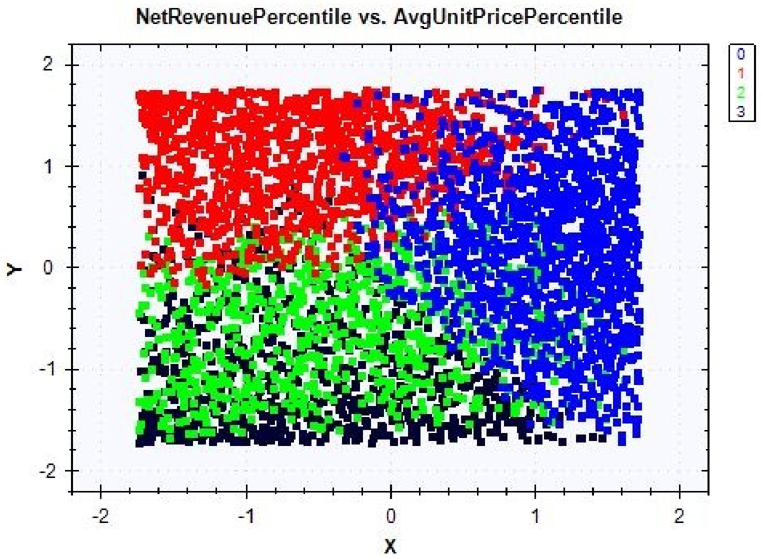
This top 10 item list for each segment gives you a rough idea of what kinds of items the customers in each segment buy the most. This is out of scope for this chapter, but you can take a step further and analyze individual words in the item description and use word frequency analysis, such as we did in `Chapter 2`, *Spam Email Filtering* and `Chapter 3`, *Twitter Sentiment Analysis*. Another way to visualize the clustering results is to draw scatter plots for the segments. The following chart shows a scatter plot of `NetRevenuePercentile` versus `AvgQuantityPercentile` for the 4-cluster clustering model:



The following chart shows a scatter plot of `AvgUnitPricePercentile` versus `AvgQuantityPercentile` for the 4-cluster clustering model:



The following chart shows a scatter plot of `NetRevenuePercentile` versus `AvgUnitPricePercentile` for the 4-cluster clustering model:



As you can see from these plots, a scatter plot is a good way to visualize how each cluster is formed and what the boundaries look like for each cluster. For example, if you look at the scatter plot of `NetRevenuePercentile` versus `AvgUnitPricePercentile`, cluster 1 has high average unit price and low net revenue. This corresponds to the findings we have from looking at the cluster centroids. For higher dimensions and larger number of clusters, it gets more difficult to visualize using scatter plots. However, very often, visualizing in charts helps draw insights more easily from these clustering analyses. Let's start looking at how we can evaluate the cluster quality and choose the optimal number of clusters using the Silhouette Coefficient.

The full code that was used in this k-means clustering step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.6/Clustering.cs`.

Clustering model validations using the Silhouette Coefficient

The **Silhouette Coefficient** or **Silhouette Score** provides an easy way to evaluate the quality of clusters. The Silhouette Coefficient measures how closely related an object is to its own cluster against the other clusters. The way to compute the Silhouette Coefficient is as follows; for each record, i, calculate the average distance between the record and all the other records in the same cluster and call this number, a_i. Then, calculate the average distances between the record and all the records in each other cluster for all the other clusters, take the lowest average distance, and call this number, b_i. Once you have these two numbers, subtract a_i from b_i and divide it by the maximum number between a_i and b_i. You iterate this process for all the records in the dataset and calculate the average value to get the Silhouette Coefficient. The following is a formula to calculate the Silhouette Coefficient for a single data point:

$$S_i = \frac{b_i - a_i}{\mathrm{Max}(a_i, b_i)}$$

In order to get the final Silhouette value, you will need to iterate through the data points and take the average of Silhouette values. The Silhouette Coefficient ranges between -1 and 1. The closer to 1, the better the cluster qualities are. The following code shows how we implemented this formula:

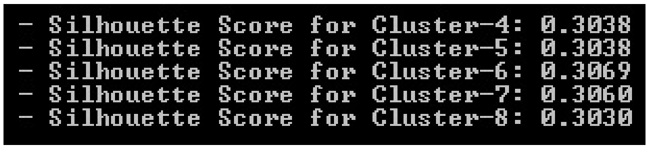
```
private static double CalculateSilhouetteScore(Frame<int, string> df, string[]
features, int numCluster, string clusterColname)
{
    double[][] data = BuildJaggedArray(df.Columns[features].ToArray2D<double>(),
df.RowCount, features.Length);

    double total = 0.0;
    for(int i = 0; i < df.RowCount; i++)
    {
        double sameClusterAverageDistance = 0.0;
        double differentClusterDistance = 1000000.0;

        double[] point = df.Columns[features].GetRowAt<double>(i).Values.ToArray();
        double cluster = df[clusterColname].GetAt(i);

        for(int j = 0; j < numCluster; j++)
        {
            double averageDistance = CalculateAverageDistance(df, features,
clusterColname, j, point);

            if (cluster == j)
            {
                sameClusterAverageDistance = averageDistance;
            } else
            {
                differentClusterDistance = Math.Min(averageDistance,
differentClusterDistance);
            }
        }

        total += (differentClusterDistance - sameClusterAverageDistance) /
Math.Max(sameClusterAverageDistance, differentClusterDistance);
    }

    return total / df.RowCount;
}
```

A helper function to calculate the average distance between a data point and all the points in a cluster is as follows:

```
private static double CalculateAverageDistance(Frame<int, string> df, string[]
features, string clusterColname, int cluster, double[] point)
{
    var clusterDF = df.Rows[
        df[clusterColname].Where(x => (int)x.Value == cluster).Keys
    ];
    double[][] clusterData = BuildJaggedArray(
        clusterDF.Columns[features].ToArray2D<double>(),
        clusterDF.RowCount,
        features.Length
    );

    double averageDistance = 0.0;
    for (int i = 0; i < clusterData.Length; i++)
    {
        averageDistance += Math.Sqrt(
            point.Select((x, j) => Math.Pow(x - clusterData[i][j], 2)).Sum()
        );
    }
    averageDistance /= (float)clusterData.Length;

    return averageDistance;
}
```

As you can see from the code, we iterate through each data point and start calculating the average distances between the given data point and all the other records in different clusters. Then, we take the difference between the lowest average distance to different clusters and the average distance within the same cluster and divide it by the maximum of those two numbers. Once we have iterated through all the data points, we take the average of this Silhouette value and return it as the Silhouette Coefficient for the clustering model.

When you run this code for the clustering models with different numbers of clusters, you will see an output similar to the following:



```
- Silhouette Score for Cluster-4: 0.3038
- Silhouette Score for Cluster-5: 0.3038
- Silhouette Score for Cluster-6: 0.3069
- Silhouette Score for Cluster-7: 0.3060
- Silhouette Score for Cluster-8: 0.3030
```

As you can see from this output, the Silhouette Score increased as we increased the number of clusters to a certain point and then it dropped. In our case, a k-means clustering model with six clusters performed the best and six clusters seem to be the best choice for our dataset.

Oftentimes, just looking at the Silhouette Coefficient is not enough to make a decision on the best number of clusters. For example, a clustering model with a really large number of clusters can have a great Silhouette Score, but it would not help us draw any insights from such a clustering model. As clustering analysis is primarily used for explanatory analysis to draw insights and identify hidden patterns from the data, it is important that the clustering results can be explained. Pairing the Silhouette Score with two-dimensional or three-dimensional scatter plots will help you come up with the best number of clusters to choose and decide what makes the most sense to your dataset and project.

Summary

In this chapter, we explored unsupervised learning and how it can be used to draw insights and identify hidden patterns in the data. Unlike other projects we have worked on so far, we did not have specific target variables that our ML models can learn from. We just had a raw online retail dataset, in which we had information about the items, quantities, and unit prices that customers bought on the online store. With this given dataset, we transformed transaction-level data into customer-level data and created numerous aggregate features. We learned how we can utilize the `AggregateRowsBy` method in Deedle's data frame to create aggregate features and transform the dataset to have a customer-centric view. We then briefly discussed a new library, `CenterSpace.NMath.Stats`, which we can use for various statistical computations. More specifically, we used the `StatsFunctions.PercentileRank` method to compute the percentiles of each record for a given feature.

We covered how we can fit a k-means clustering algorithm using the `Accord.NET` framework. Using the k-means clustering algorithm, we were able to build a few clustering models with different numbers of clusters. We discussed how we can draw insights using the 4-cluster clustering model as an example and how we can cluster customers into different customer segments, where one segment's customer characteristics were high net revenue, above-average unit price, and above-average quantity, the other segment's customer characteristics were low net revenue, high average unit price, and below-average quantity, and so forth. We then looked at the top 10 items each customer segment purchased the most frequently and created scatter plots of different segments on our feature space.

Lastly, we used the **Silhouette Coefficient** to evaluate the cluster qualities, and learned how we can use this as one of the criteria for choosing the optimal number of clusters.

From the next chapter, we are going to start building models for audio and image datasets. In the next chapter, we are going to discuss how to build a music genre recommendation model using a music audio dataset. We will learn how to build a ranking system where the output is the ranks of likelihood of individual categories. We will also learn what types of metrics to use to evaluate such a ranking model.

Music Genre Recommendation

In this chapter, we are going to go back to supervised learning. We have built numerous supervised learning algorithms for both classification and regression problems using learning algorithms such as logistic regression, Naive Bayes, random forest, and **Support Vector Machine** (**SVM**). However, the number of outputs from these models we have built has always been one. In our Twitter sentiment analysis project, the output could only be one of positive, negative, or neutral. On the other hand, in our housing price prediction project, the output was a log of house prices predicted. Unlike our previous projects, there are cases where we want our **machine learning** (**ML**) models to output multiple values. A recommendation system is one example of where we need ML models that can produce rank-ordered predictions.

In this chapter, we are going to use a dataset that contains various audio features, compiled from numerous music recordings. With this data, we are going to explore how the values of audio features, such as kurtosis and skewness of the sound spectrum, are distributed across different genres of songs. Then, we are going to start building multiple ML models that output the predicted probabilities of the given song belonging to each music genre, instead of producing just one prediction output of the most likely genre for a given song. Once we have these models built, we are going to take it a step further and ensemble the prediction results of these base models to build a meta model for the final recommendations of song music genres. We are going to use a different model validation metric, **Mean Reciprocal Rank** (**MRR**), to evaluate our ranking models.

In this chapter, we will cover the following topics:

- Problem definition for the Music Genre Recommendation project
- Data analysis for the audio features dataset
- ML models for music genre classification
- Ensembling base learning models
- Evaluating recommendation/rank-ordering models

Problem definition

Let's get into greater detail and properly define what problems we are going to solve and what machine learning models we are going to build for this project. Music streaming services, such as Pandora and Spotify, require music recommendation systems, with which they can recommend and play songs that their listeners might like. There is more than one way to build a music recommendation system. One way is to look at what other similar users listened to, and the way to define similar users is to look at the history of songs that they listened to. However, this approach will not work well if the user is new to the platform and/or if we do not have enough of a history of songs he or she listened to. In this case, we cannot rely on the historical data. Instead, it will be better to use the attributes of the songs that the user is currently listening to recommend other music. One song attribute that can play an important role in music recommendation is the genre. It is highly likely that a user who is currently listening to music on the platform will like to continue listening to the same or similar music. Imagine you were listening to instrumental music and the music streaming application then suddenly played rock music. It would not be a smooth transition and it would not be a good user experience, as you most likely would have wanted to continue listening to instrumental music. By correctly identifying the genre of the songs and recommending the right song type to play, you can avoid disturbing the user experience of your music streaming service.

In order to build a music genre recommendation model, we are going to use **FMA: A Dataset For Music Analysis**, which contains a large amount of data for over 100,000 tracks. The dataset contains information about the album, title, audio attributes, and so forth, and the full dataset can be found and downloaded from this link: `https://github.com/mdeff/fma`. With this data, we are going to sub-select the features that are of interest and build numerous ML models that output the probability of each song belonging to different music genres. Then, we are going to rank-order the genres by probability. We will be experimenting with various learning algorithms, such as logistic regression, Naive Bayes, and SVM. We are going to take it a step further by using the ensembling technique to take the output of these models as an input to another ML model that produces the final prediction and recommendation output. We are going to use MRR as the metric to evaluate our music genre recommendation models.

To summarize our problem definition for the music genre recommendation project:

- What is the problem? We need a recommendation model that rank-orders music genres by how likely it is that a song belongs to each genre, so that we can properly identify the genre of a song and recommend what song to play next.
- Why is it a problem? Use of historical data for music recommendation is not reliable for those users who are new to the platform, as they will not have enough historical data for good music recommendations. In this case, we will have to use audio and other features to identify what music to play next. Correctly identifying and recommending the genre of music is the first step to figuring out what song to play next.
- What are some approaches to solving this problem? We are going to use publicly available music data, which not only contains information about the album, title, and artist of the song, but also contains information about numerous audio features. Then, we are going to build ML models that output the probabilities and use this probability output to rank-order genres for given song.
- What are the success criteria? We want the correct music genre to come up as one of the top predicted genres. We will use MRR as the metric to evaluate ranking models.

Data analysis for the audio features dataset

Let's start looking into the audio features dataset. In order to focus on building recommendation models for music genres, we trimmed down the original dataset from **FMA: A Dataset For Music Analysis**. You can download this data from this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.7/sample.csv`.

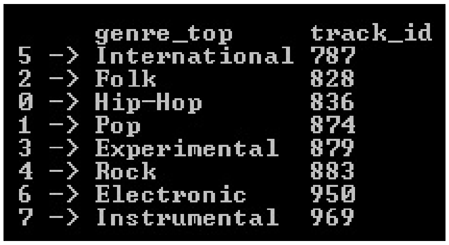
Target variable distribution

We will first look at the distribution of our target variable for this project and figure out how many records we have for each genre in our sample set. The following code snippet shows how we aggregated our sample set by the target variable, `genre_top`, and counted the number of records for each genre:

```
var genreCount = featuresDF.AggregateRowsBy<string, int>(
    new string[] { "genre_top" },
    new string[] { "track_id" },
    x => x.ValueCount
).SortRows("track_id");

genreCount.Print();

var barChart = DataBarBox.Show(
    genreCount.GetColumn<string>("genre_top").Values.ToArray().Select(x =>
x.Substring(0,3)),
    genreCount["track_id"].Values.ToArray()
).SetTitle(
    "Genre Count"
);
```

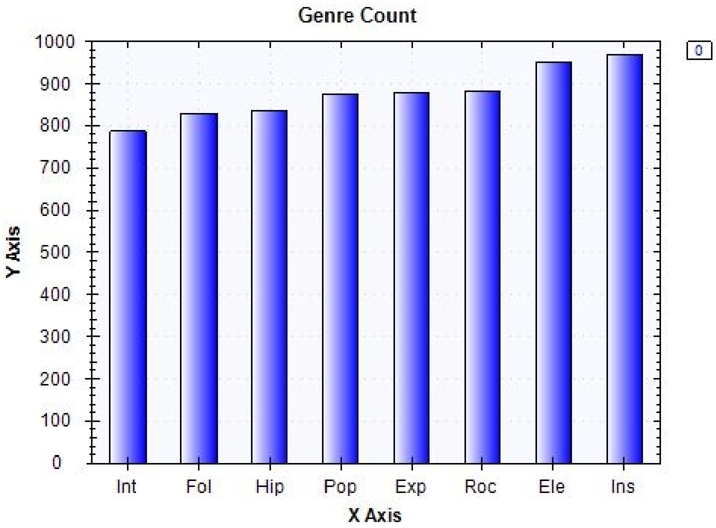
Similar to previous chapters, we used the `AggregateRowsBy` method in the Deedle data frame to count the number of records per genre. Then, we used the `DataBarBox` class to create a bar chart that shows the distribution of the target variable visually. As you can see from this code snippet (in line 10), we are using the first three letters of each genre as a label for each genre in the bar chart.

When you run this code, you will see the following output for the distribution of the target variable:



```
       genre_top      track_id
5 -> International 787
2 -> Folk          828
0 -> Hip-Hop       836
1 -> Pop           874
3 -> Experimental  879
4 -> Rock          883
6 -> Electronic    950
7 -> Instrumental  969
```

The following plot shows the bar chart for the distribution of the target variable:



As you can see from this chart, we have the largest number for Instrumental (**Ins**) music in our sample set and Electronic (**Ele**) and Rock (**Roc**) follow as the second and third. Although this sample set contains some songs in certain genres more so than others, this is a relatively well balanced set, where one or two genres do not take up the majority of the sample records. Now, let's look at the distributions of some of our features.

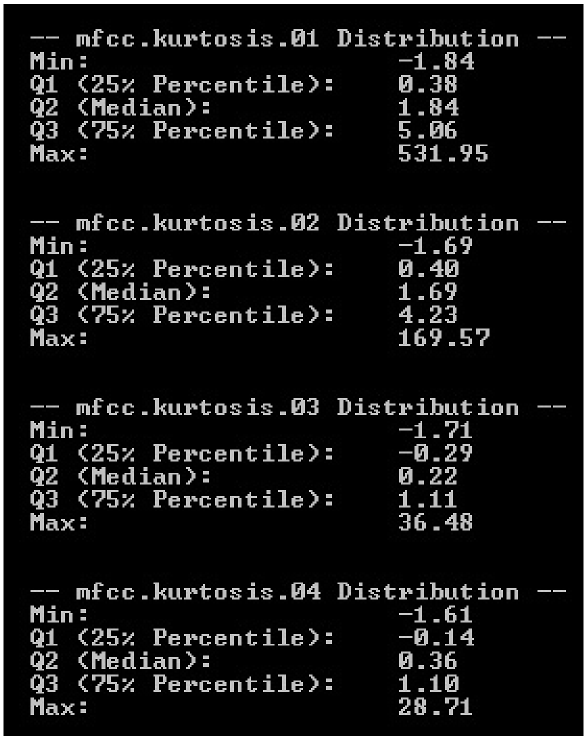
Audio features – MFCC

For this project, we are going to focus on a subset of features that the full dataset has. We are going to use **Mel Frequency Cepstral Coefficients** (**MFCCs**) and their statistical distributions as the features to our ML models. Simply put, **MFCC** is a representation of the sound spectrum and we will use its statistical distributions, kurtosis, skewness, min, max, mean, median, and standard deviation. If you look at the sample set you have downloaded from the previous step, you will see the columns are named according to the corresponding statistical distribution. We are going to first look at the distributions of each of these features. The following code snippet shows how we computed the quartiles for each feature:

```
foreach (string col in featuresDF.ColumnKeys)
{
    if (col.StartsWith("mfcc"))
    {
        int idx = int.Parse(col.Split('.')[2]);
        if(idx <= 4)
        {
            Console.WriteLine(String.Format("\n\n-- {0} Distribution -- ", col));
            double[] quantiles = Accord.Statistics.Measures.Quantiles(
                featuresDF[col].ValuesAll.ToArray(),
                new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
            );
            Console.WriteLine(
                "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
                quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
            );
        }
    }
}
```

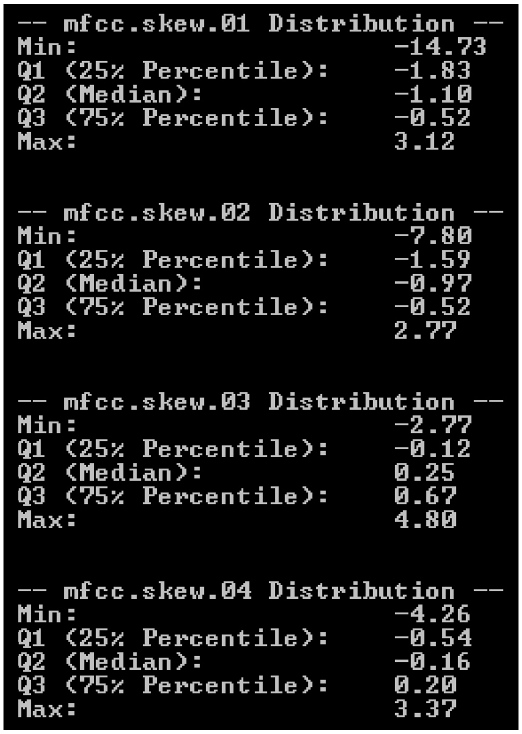
Similar to previous chapters, we are using the `Quantiles` method in the `Accord.Statistics.Measures` class to compute quartiles, which are the three numbers that separate the values into four subsets—the middle number between the min and median (25th percentile), median (50th percentile), and the middle number between the median and max (75th percentile). As you can see in line 6 of this code snippet, we are only showing the first four coefficients' statistical distributions. For your further experiments, you can look at the distributions of all the MFCC features, not limited to only these four. Let's quickly take a look at just a couple of the distributions.

The distribution for the kurtosis of the first four coefficients looks like the following:



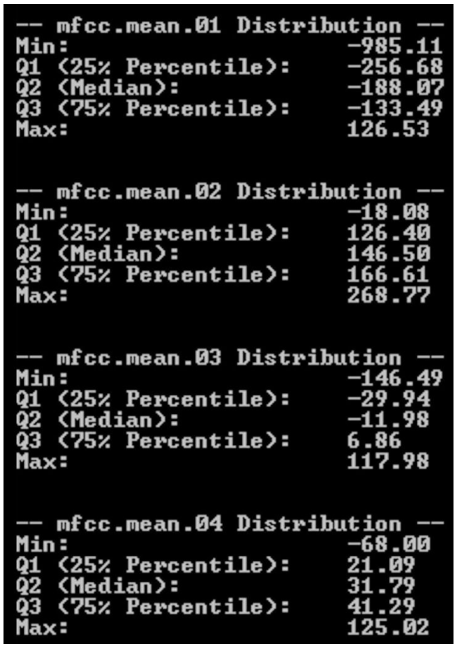
```
-- mfcc.kurtosis.01 Distribution --
Min:                        -1.84
Q1 (25% Percentile):        0.38
Q2 (Median):                1.84
Q3 (75% Percentile):        5.06
Max:                        531.95

-- mfcc.kurtosis.02 Distribution --
Min:                        -1.69
Q1 (25% Percentile):        0.40
Q2 (Median):                1.69
Q3 (75% Percentile):        4.23
Max:                        169.57

-- mfcc.kurtosis.03 Distribution --
Min:                        -1.71
Q1 (25% Percentile):        -0.29
Q2 (Median):                0.22
Q3 (75% Percentile):        1.11
Max:                        36.48

-- mfcc.kurtosis.04 Distribution --
Min:                        -1.61
Q1 (25% Percentile):        -0.14
Q2 (Median):                0.36
Q3 (75% Percentile):        1.10
Max:                        28.71
```

As you can see from this output, the majority of the kurtosis values fall between -2 and 5, but there are cases where the kurtosis can take large values. Let's now look at the skewness distributions for the first four coefficients:



```
-- mfcc.skew.01 Distribution --
Min:                        -14.73
Q1 (25% Percentile):        -1.83
Q2 (Median):                -1.10
Q3 (75% Percentile):        -0.52
Max:                        3.12


-- mfcc.skew.02 Distribution --
Min:                        -7.80
Q1 (25% Percentile):        -1.59
Q2 (Median):                -0.97
Q3 (75% Percentile):        -0.52
Max:                        2.77


-- mfcc.skew.03 Distribution --
Min:                        -2.77
Q1 (25% Percentile):        -0.12
Q2 (Median):                0.25
Q3 (75% Percentile):        0.67
Max:                        4.80


-- mfcc.skew.04 Distribution --
Min:                        -4.26
Q1 (25% Percentile):        -0.54
Q2 (Median):                -0.16
Q3 (75% Percentile):        0.20
Max:                        3.37
```

Skewness varies between narrower ranges. Typically, the skewness values seem to fall between -15 and 5. Lastly, let's look at the distributions of the mean of the first four coefficients:

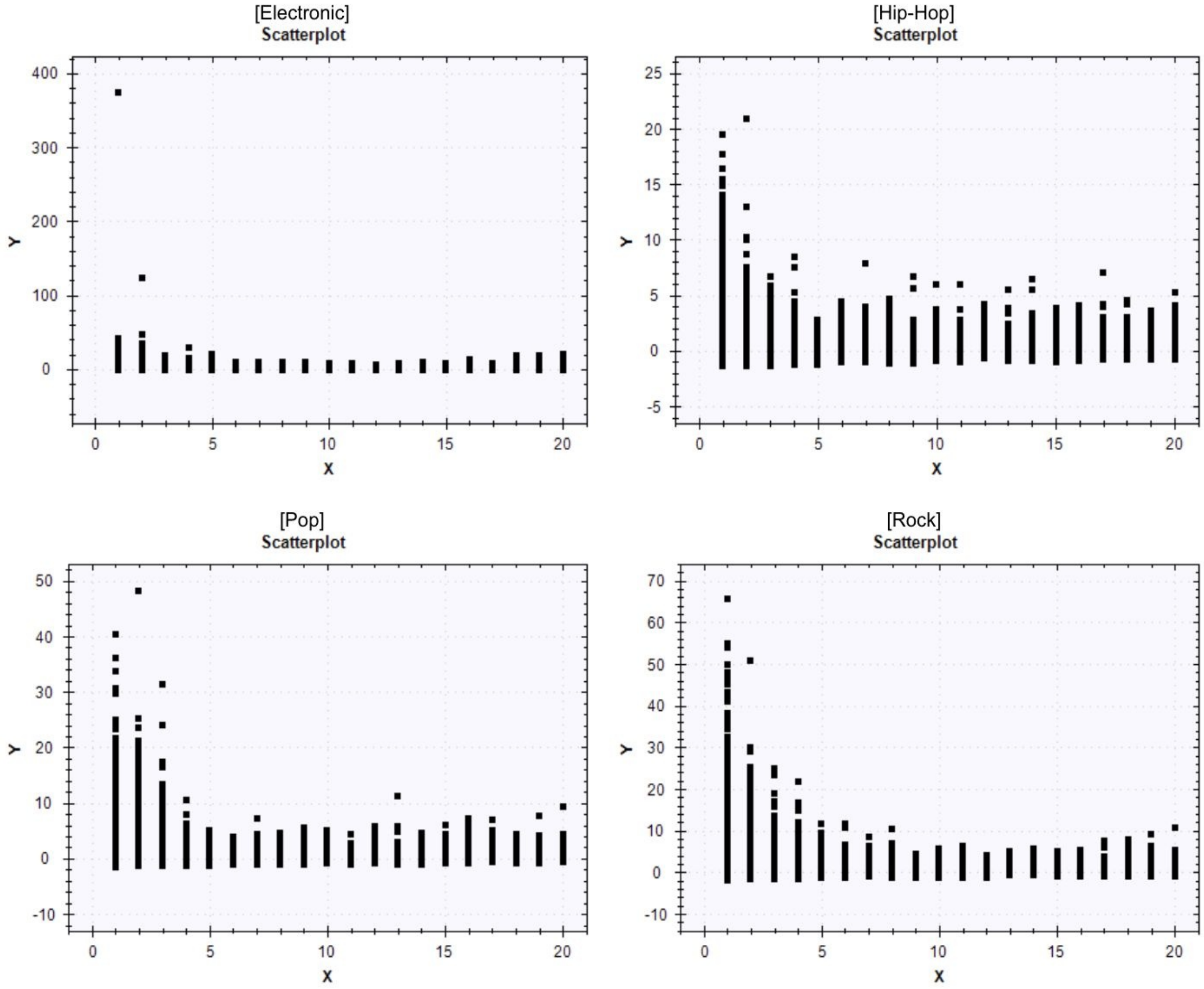


```
-- mfcc.mean.01 Distribution --
Min:                        -985.11
Q1 (25% Percentile):        -256.68
Q2 (Median):                -188.07
Q3 (75% Percentile):        -133.49
Max:                        126.53


-- mfcc.mean.02 Distribution --
Min:                        -18.08
Q1 (25% Percentile):        126.40
Q2 (Median):                146.50
Q3 (75% Percentile):        166.61
Max:                        268.77


-- mfcc.mean.03 Distribution --
Min:                        -146.49
Q1 (25% Percentile):        -29.94
Q2 (Median):                -11.98
Q3 (75% Percentile):        6.86
Max:                        117.98


-- mfcc.mean.04 Distribution --
Min:                        -68.00
Q1 (25% Percentile):        21.09
Q2 (Median):                31.79
Q3 (75% Percentile):        41.29
Max:                        125.02
```

As you can see from this output, the mean values seem to vary and have wide ranges. It can take any values between -1,000 and 300.

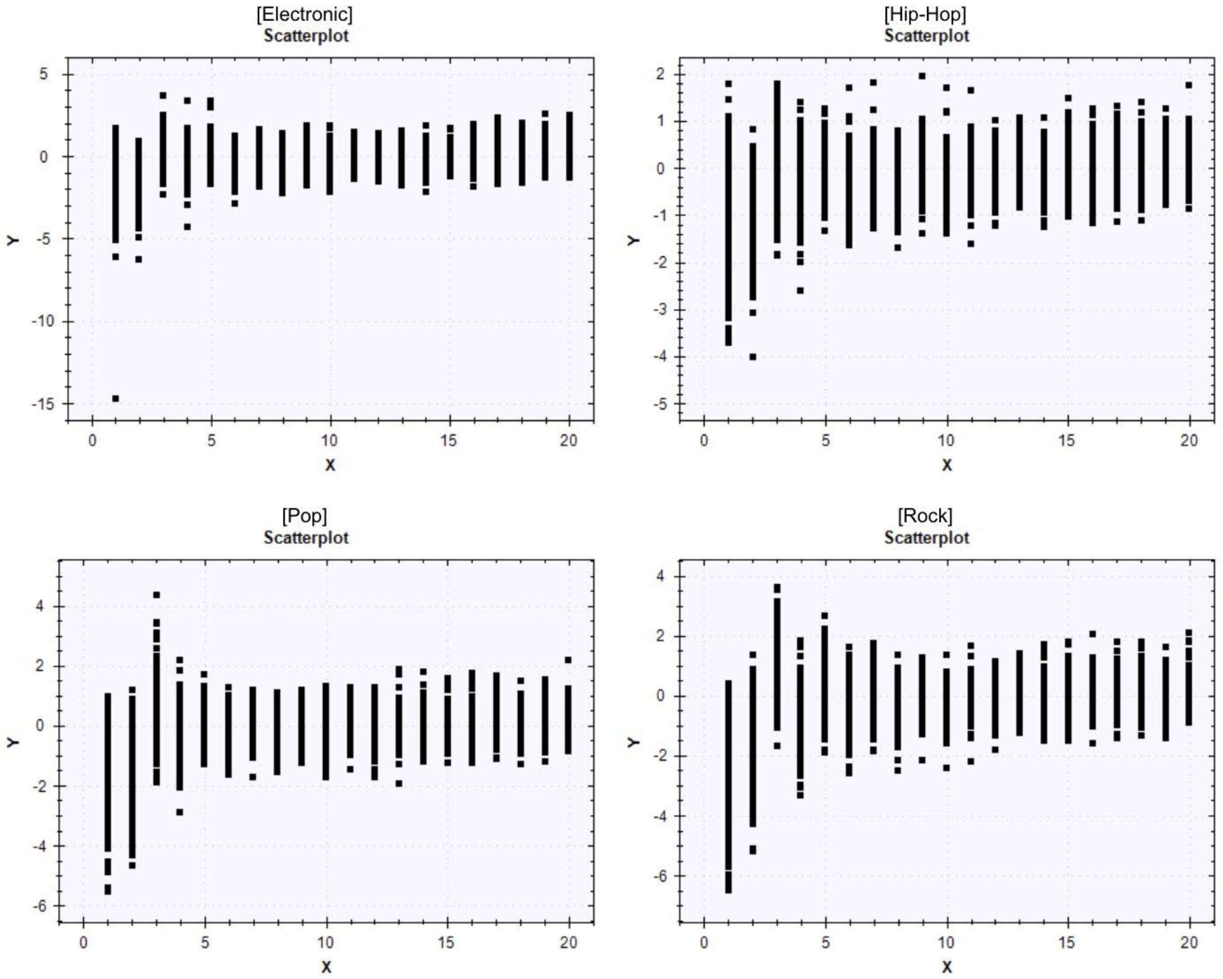
Now that we have a rough idea of how the audio features' distributions look, let's see if we can find any discrepancies in the feature distributions among different genres. We are going to plot a scatter plot where the *x* axis is the index of each feature and the *y* axis is the values for the given feature. Let's look at these plots first, as it will be easier to understand with some visuals.

The following plots show the distributions of kurtosis for four different genres:

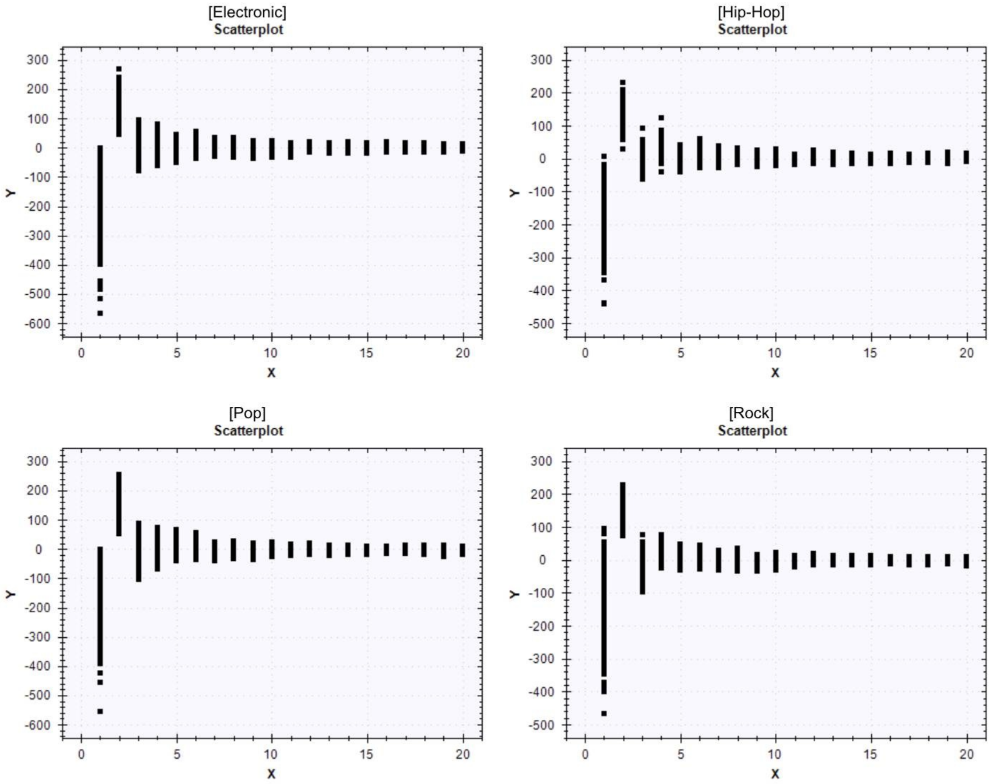


As briefly mentioned previously, the *x* axis refers to the index of each feature. Since we have 20 individual features for kurtosis of MFCCs, the x-values span from 1 to 20. On the other hand, the *y* axis shows the distributions of the given feature. As you can see from this chart, there are some differences in the feature distributions among different genres, which will help our ML models to learn how to correctly predict the genre of a given song.

The following plots show the distributions of skewness for four different genres:



Lastly, the following plots show the mean distributions for four different genres:



The distributions of the mean values for each feature seem more similar among different genres, when compared to the kurtosis and skewness.

In order to create these charts, we have used the `ScatterplotBox` class. The following code shows how we created the previous charts:

```
string[] attributes = new string[] { "kurtosis", "min", "max", "mean", "median",
"skew", "std" };
foreach (string attribute in attributes)
{
    string[] featureColumns = featuresDF.ColumnKeys.Where(x =>
x.Contains(attribute)).ToArray();
    foreach (string genre in genreCount.GetColumn<string>("genre_top").Values)
    {
        var genreDF = featuresDF.Rows[
            featuresDF.GetColumn<string>("genre_top").Where(x => x.Value == genre).Keys
        ].Columns[featureColumns];

        ScatterplotBox.Show(
            BuildXYPairs(
                genreDF.Columns[featureColumns].ToArray2D<double>(),
                genreDF.RowCount,
                genreDF.ColumnCount
            )
        ).SetTitle(String.Format("{0}-{1}", genre, attribute));
    }
}
```

As you can see from this code, we start iterating through different statistical distributions (`kurtosis`, `min`, `max`, and so on) from line 2 and, for each of those statistical distributions, we sub-select the columns that we are interested in from `featuresDF` in line 7. Then, we wrote and used a helper function that builds an array of x-y pairs for the scatter plot and display it using the `Show` method of the `ScatterplotBox` class.

The code for the helper function that builds x-y pairs for scatter plots is as follows:

```
private static double[][] BuildXYPairs(double[,] ary2D, int rowCount, int columnCount)
{
    double[][] ary = new double[rowCount*columnCount][];
    for (int i = 0; i < rowCount; i++)
    {
        for (int j = 0; j < columnCount; j++)
        {
            ary[i * columnCount + j] = new double[2];
            ary[i * columnCount + j][0] = j + 1;
            ary[i * columnCount + j][1] = ary2D[i, j];
        }
    }
    return ary;
}
```

As you can see from this code, this method takes the index of the feature as an x value and takes the value of the feature as a y value.

The full code for this data analysis step can be found at this link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.7/DataAnalyzer.cs.

ML models for music genre classification

We will now start building ML models for music genre classification. In this project, the output of our ML models will take a slightly different form. Unlike other supervised learning models that we have built, we want our models to output the likelihoods or probabilities for each genre for a given song. So, instead of the model output being one value, we would like our models to output eight values, where each value will represent the probability of the given song belonging to each of the eight genres—electronic, experimental, folk, hip-hop, instrumental, international, pop, and rock. In order to achieve this, we will be using the `Probabilities` method from each of the model classes, on top of the `Decide` method that we have been using so far.

Logistic regression

The first model we are going to experiment with is logistic regression. The following code shows how we built a logistic regression classifier with an 80/20 split for training and testing sets:

```
// 1. Train a LogisticRegression Classifier
Console.WriteLine("\n---- Logistic Regression Classifier ----\n");
var logitSplitSet = new SplitSetValidation<MultinomialLogisticRegression, double[]>()
{
    Learner = (s) => new MultinomialLogisticLearning<GradientDescent>()
    {
        MiniBatchSize = 500
    },

    Loss = (expected, actual, p) => new ZeroOneLoss(expected).Loss(actual),

    Stratify = false,

    TrainingSetProportion = 0.8,

    ValidationSetProportion = 0.2,

};

var logitResult = logitSplitSet.Learn(input, output);

var logitTrainedModel = logitResult.Model;

// Store train & test set indexes to train other classifiers on the same train set
// and test on the same validation set
int[] trainSetIDX = logitSplitSet.IndicesTrainingSet;
int[] testSetIDX = logitSplitSet.IndicesValidationSet;
```

As you should be familiar with it already, we used `SplitSetValidation` to split our sample set into train and test sets. We are using 80% of our sample set for training and the other 20% for testing and evaluating our models. We are using `MultinomialLogisticRegression` as our model for the multi-class classifier and `MultinomialLogisticLearning` with `GradientDescent` as our learning algorithm. Similar to the previous chapters, we are using `ZeroOneLoss` for our `Loss` function for the classifier.

As you can see at the base of this code, we are storing the trained logistic regression classifier model into a separate variable, `logitTrainedModel`, and also the indexes of the train and test sets for use in training and testing other learning algorithms. We do this so that we can do head-to-head comparisons of model performance among different ML models.

The code to do in-sample and out-of-sample predictions using this trained logistic regression model is as follows:

```
// Get in-sample & out-of-sample predictions and prediction probabilities for each
class
double[][] trainProbabilities = new double[trainSetIDX.Length][];
int[] logitTrainPreds = new int[trainSetIDX.Length];
for (int i = 0; i < trainSetIDX.Length; i++)
{
    logitTrainPreds[i] = logitTrainedModel.Decide(input[trainSetIDX[i]]);
    trainProbabilities[i] = logitTrainedModel.Probabilities(input[trainSetIDX[i]]);
}

double[][] testProbabilities = new double[testSetIDX.Length][];
int[] logitTestPreds = new int[testSetIDX.Length];
for (int i = 0; i < testSetIDX.Length; i++)
{
    logitTestPreds[i] = logitTrainedModel.Decide(input[testSetIDX[i]]);
    testProbabilities[i] = logitTrainedModel.Probabilities(input[testSetIDX[i]]);
}
```

As briefly mentioned before, we are using the `Probabilities` method from the `MultinomialLogisticRegression` model, which outputs an array of probabilities, and each index represents the probability of the given song being the corresponding music genre. The following code shows how we encoded each of the genres:

```
IDictionary<string, int> targetVarCodes = new Dictionary<string, int>
{
    { "Electronic", 0 },
    { "Experimental", 1 },
    { "Folk", 2 },
    { "Hip-Hop", 3 },
    { "Instrumental", 4 },
    { "International", 5 },
    { "Pop", 6 },
    { "Rock", 7 }
};
featuresDF.AddColumn("target", featuresDF.GetColumn<string>("genre_top").Select(x =>
targetVarCodes[x.Value]));
```

Let's try training another ML model using the same indexes for train and test sets that we used for the logistic regression model.

SVM with the Gaussian kernel

Using the following code, you can train a multi-class SVM model:

```
// 2. Train a Gaussian SVM Classifier
Console.WriteLine("\n---- Gaussian SVM Classifier ----\n");
var teacher = new MulticlassSupportVectorLearning<Gaussian>()
{
    Learner = (param) => new SequentialMinimalOptimization<Gaussian>()
    {
        Epsilon = 2,
        Tolerance = 1e-2,
        Complexity = 1000,
        UseKernelEstimation = true
    }
};
// Train SVM model using the same train set that was used for Logistic Regression
Classifier
var svmTrainedModel = teacher.Learn(
    input.Where((x,i) => trainSetIDX.Contains(i)).ToArray(),
    output.Where((x, i) => trainSetIDX.Contains(i)).ToArray()
);
```

As you can see from this code, there is one minor difference between the SVM model that we built previously. We are using `MulticlassSupportVectorLearning` instead of `LinearRegressionNewtonMethod` or `FanChenLinSupportVectorRegression`, which we used in `Chapter 5`, *Fair Value of House and Property*. This is because we now have a multi-class classification problem and need to use a different learning algorithm for such SVM models. As we discussed in another chapter previously, the hyper-parameters, such as `Epsilon`, `Tolerance`, and `Complexity`, can be tuned and you should experiment with other values for better-performing models.

One thing to note here is that when we are training our SVM model, we use the same train set that we used for building the logistic regression model. As you can see at the base of the code, we sub-select records with the same indexes in the train set that we used previously for the logistic regression model. This is to make sure that we can correctly do a head-to-head comparison of the performance of this SVM model against that of the logistic regression model.

Similar to the case of the previous logistic regression model, we are using the following code for in-sample and out-of-sample predictions, using the trained SVM model:

```
// Get in-sample & out-of-sample predictions and prediction probabilities for each
```

```
class
double[][] svmTrainProbabilities = new double[trainSetIDX.Length][];
int[] svmTrainPreds = new int[trainSetIDX.Length];
for (int i = 0; i < trainSetIDX.Length; i++)
{
    svmTrainPreds[i] = svmTrainedModel.Decide(input[trainSetIDX[i]]);
    svmTrainProbabilities[i] = svmTrainedModel.Probabilities(input[trainSetIDX[i]]);
}

double[][] svmTestProbabilities = new double[testSetIDX.Length][];
int[] svmTestPreds = new int[testSetIDX.Length];
for (int i = 0; i < testSetIDX.Length; i++)
{
    svmTestPreds[i] = svmTrainedModel.Decide(input[testSetIDX[i]]);
    svmTestProbabilities[i] = svmTrainedModel.Probabilities(input[testSetIDX[i]]);
}
```

The `MulticlassSupportVectorMachine` class also provides the `Probabilities` method, with which we can get the likelihoods of a song belonging to each of the eight genres. We store these probability outputs into separate variables, `svmTrainProbabilities` and `svmTestProbabilities`, for our future model evaluation and for ensembling the models.

Naive Bayes

We are going to build one more machine learning model for music genre classification. We are going to train a Naive Bayes classifier. The following code shows how you can build a Naive Bayes classifier for input with continuous values:

```
// 3. Train a NaiveBayes Classifier
Console.WriteLine("\n---- NaiveBayes Classifier ----\n");
var nbTeacher = new NaiveBayesLearning<NormalDistribution>();

var nbTrainedModel = nbTeacher.Learn(
    input.Where((x, i) => trainSetIDX.Contains(i)).ToArray(),
    output.Where((x, i) => trainSetIDX.Contains(i)).ToArray()
);
```

As you can see from this code, we are using `NormalDistribution` as a distribution for `NaiveBayesLearning`. Unlike in the previous chapters, where we had word counts as features of our Naive Bayes classifiers, we have continuous values for our audio features. In this case, we need to build a Gaussian Naive Bayes classifier. Similar to when we were building an SVM model, we are training our Naive Bayes classifier with the same train set that we used for the logistic regression model.

The following code shows how we can get the probability output for in-sample and out-of-sample predictions using the trained Naive Bayes classifier:

```
// Get in-sample & out-of-sample predictions and prediction probabilities for each
class
double[][] nbTrainProbabilities = new double[trainSetIDX.Length][];
int[] nbTrainPreds = new int[trainSetIDX.Length];
for (int i = 0; i < trainSetIDX.Length; i++)
{
    nbTrainProbabilities[i] = nbTrainedModel.Probabilities(input[trainSetIDX[i]]);
    nbTrainPreds[i] = nbTrainedModel.Decide(input[trainSetIDX[i]]);
}

double[][] nbTestProbabilities = new double[testSetIDX.Length][];
int[] nbTestPreds = new int[testSetIDX.Length];
for (int i = 0; i < testSetIDX.Length; i++)
{
    nbTestProbabilities[i] = nbTrainedModel.Probabilities(input[testSetIDX[i]]);
    nbTestPreds[i] = nbTrainedModel.Decide(input[testSetIDX[i]]);
}
```

Similar to the `MulticlassSupportVectorMachine` and `MultinomialLogisticRegression` classes, the `NaiveBayes` model also provides the `Probabilities` method. As you can see from the code, we store the predicted probabilities for both in-sample and out-of-sample records into two separate variables, `nbTrainProbabilities` and `nbTestProbabilities`.

In the following section, we will take a look at how we can combine and ensemble these models we have built so far. The full code for building ML models can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.7/Modeling.cs`.

Ensembling base learning models

Ensemble learning is where you combine trained models together in order to improve their predictive power. The random forest classifier that we built in previous chapters is an example of ensemble learning. It builds a forest of decision trees, where the individual trees are trained with a portion of the samples and features in the sample set. This method of ensemble learning is called **bagging**. The ensemble method that we are going to use in this chapter is **stacking**. Stacking is when you build a new ML model using the outputs of the other models, which are called **base learning models**.

In this project, we are going to built a new Naive Bayes classifier model on top of the predicted probability output from those logistic regression, SVM, and Naive Bayes models that we built in the previous section. The first thing we need to do to build a new model with the probability output of the base models is to build the training input. The following code shows how we combined all the outputs from the base models:

```
// 4. Ensembling Base Models
Console.WriteLine("\n-- Building Meta Model --");
double[][] combinedTrainProbabilities = new double[trainSetIDX.Length][];
for (int i = 0; i < trainSetIDX.Length; i++)
{
    List<double> combined = trainProbabilities[i]
        .Concat(svmTrainProbabilities[i])
        .Concat(nbTrainProbabilities[i])
        .ToList();
    combined.Add(logitTrainPreds[i]);
    combined.Add(svmTrainPreds[i]);
    combined.Add(nbTrainPreds[i]);

    combinedTrainProbabilities[i] = combined.ToArray();
}

double[][] combinedTestProbabilities = new double[testSetIDX.Length][];
for (int i = 0; i < testSetIDX.Length; i++)
{
    List<double> combined = testProbabilities[i]
        .Concat(svmTestProbabilities[i])
        .Concat(nbTestProbabilities[i])
        .ToList();
    combined.Add(logitTestPreds[i]);
    combined.Add(svmTestPreds[i]);
    combined.Add(nbTestPreds[i]);

    combinedTestProbabilities[i] = combined.ToArray();
}
Console.WriteLine("\n* input shape: ({0}, {1})\n", combinedTestProbabilities.Length,
combinedTestProbabilities[0].Length);
```

As you can see from this code, we are concatenating the predicted probabilities from all three models that we built so far. Using this probability output data as input, we are going to build a new meta-model, using the Naive Bayes learning algorithm. The following code is how we trained this meta-model:

```
// Build meta-model using NaiveBayes Learning Algorithm
var metaModelTeacher = new NaiveBayesLearning<NormalDistribution>();
var metamodel = metaModelTeacher.Learn(
    combinedTrainProbabilities,
    output.Where((x, i) => trainSetIDX.Contains(i)).ToArray()
);
```

From this code, you can see that we are still using `NormalDistribution`, as the input is a set of continuous values. Then, we train this new Naive Bayes classifier with the combined probability output of the base learning models that we trained before. Similar to the previous steps, we get the prediction output from this meta-model by using the `Probabilities` method and store these results into separate variables. The code to get the prediction output for the train and test sets using this new meta-model is as follows:

```
// Get in-sample & out-of-sample predictions and prediction probabilities for each
class
double[][] metaTrainProbabilities = new double[trainSetIDX.Length][];
int[] metamodelTrainPreds = new int[trainSetIDX.Length];
for (int i = 0; i < trainSetIDX.Length; i++)
{
    metaTrainProbabilities[i] = metamodel.Probabilities(combinedTrainProbabilities[i]);
    metamodelTrainPreds[i] = metamodel.Decide(combinedTrainProbabilities[i]);
}

double[][] metaTestProbabilities = new double[testSetIDX.Length][];
int[] metamodelTestPreds = new int[testSetIDX.Length];
for (int i = 0; i < testSetIDX.Length; i++)
{
    metaTestProbabilities[i] = metamodel.Probabilities(combinedTestProbabilities[i]);
    metamodelTestPreds[i] = metamodel.Decide(combinedTestProbabilities[i]);
}
```

Now that we have all the models built, let's start looking at the performances of these models. In the following section, we will evaluate the performance of base models as well as the meta-model we just built.

Evaluating recommendation/rank-ordering models

Evaluating recommendation models that rank-order the outcomes is quite different from evaluating classification models. Aside from whether the model prediction is right or wrong, we also care about in which rank the correct outcome comes in the recommendation models. In other words, a model that predicted the correct outcome to be the second from the top is a better model than a model that predicted the correct outcome to be fourth or fifth from the top. For example, when you search for something on a search engine, getting the most appropriate document on the top of the first page is great, but it is still OK to have that document as the second or third link on the first page, as long as it does not appear at the bottom of the first page or the next page. We are going to discuss some ways to evaluate such recommendation and ranking models in the following sections.

Prediction accuracy

The first and the simplest metric to look at is accuracy. For the first logistic regression model we built, we can use the following code to get the accuracy:

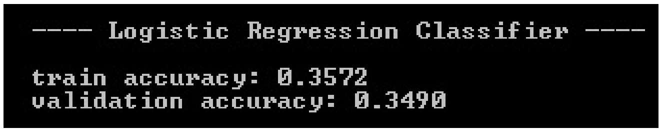
```
Console.WriteLine(String.Format("train accuracy: {0:0.0000}", 1-
logitResult.Training.Value));
Console.WriteLine(String.Format("validation accuracy: {0:0.0000}", 1-
logitResult.Validation.Value));
```

For the following models, SVM and Naive Bayes classifiers, we can use the following code to compute the accuracy for the train and test set predictions:

```
Console.WriteLine(
    String.Format(
        "train accuracy: {0:0.0000}",
        1 - new ZeroOneLoss(output.Where((x, i) =>
trainSetIDX.Contains(i)).ToArray()).Loss(nbTrainPreds)
    )
);
Console.WriteLine(
    String.Format(
        "validation accuracy: {0:0.0000}",
        1 - new ZeroOneLoss(output.Where((x, i) =>
testSetIDX.Contains(i)).ToArray()).Loss(nbTestPreds)
    )
);
```

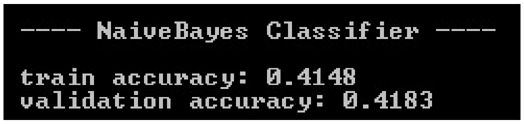
We used the `SplitSetValidation` class for the first logistic regression model, so it computes the accuracy while the model is being fit. However, for the subsequent models, we trained SVM and Naive Bayes models individually, so we need to use the `ZeroOneLoss` class to compute accuracies.

When you run this code, you will see the accuracy output for the logistic regression model as follows:



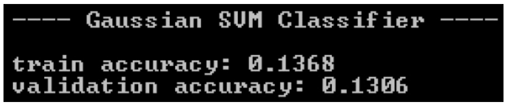
```
---- Logistic Regression Classifier ----

train accuracy: 0.3572
validation accuracy: 0.3490
```

For the Naive Bayes model, the accuracy results look as follows:



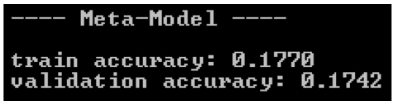
```
---- NaiveBayes Classifier ----

train accuracy: 0.4148
validation accuracy: 0.4183
```

And for the SVM model, the output looks as follows:



```
---- Gaussian SVM Classifier ----

train accuracy: 0.1368
validation accuracy: 0.1306
```

Lastly, the accuracy results for the meta-model look as follows:



```
---- Meta-Model ----

train accuracy: 0.1770
validation accuracy: 0.1742
```

From these results, we can see that the Naive Bayes classifier performed the best by predicting the correct genre for about 42% of the time. The logistic regression model comes in as the second best model with the second highest accuracy and the SVM model comes in as the worst model in terms of prediction accuracy. Interestingly, the meta-model that we built with the output from the other three models did not perform so well. It did better than the SVM model, but performed worse than the Naive Bayes and logistic regression classifiers.

Confusion matrices

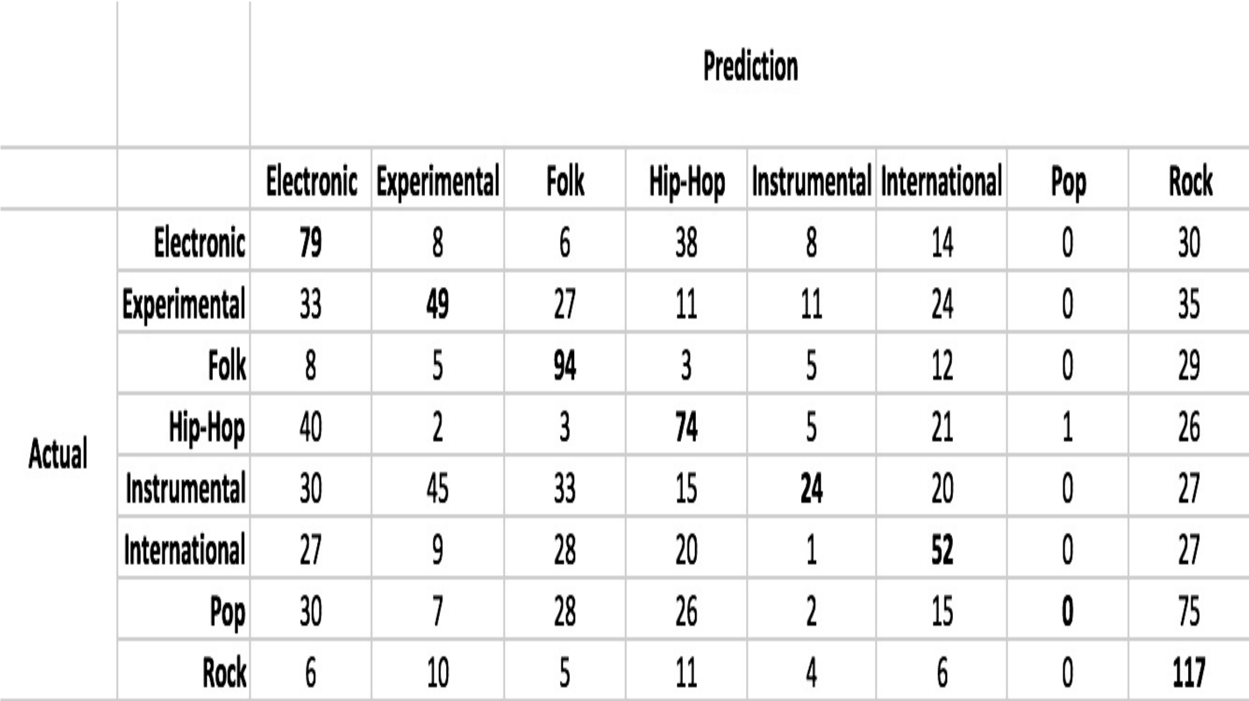
The next thing we are going to look at is confusion matrices. In the case of binary classification in `Chapter 2`, *Spam Email Filtering*, we explored a case where the confusion matrix was a 2 x 2 matrix. However, in this project, our models have `8` outcomes and the shape of the confusion matrix will be 8 x 8. Let's first look at how we can build such a confusion matrix:

```
// Build confusion matrix
string[] confMatrix = BuildConfusionMatrix(
    output.Where((x, i) => testSetIDX.Contains(i)).ToArray(), logitTestPreds, 8
);

System.IO.File.WriteAllLines(Path.Combine(dataDirPath, "logit-conf-matrix.csv"),
confMatrix);
```

The code for the helper function, `BuildConfusionMatrix`, looks as follows:

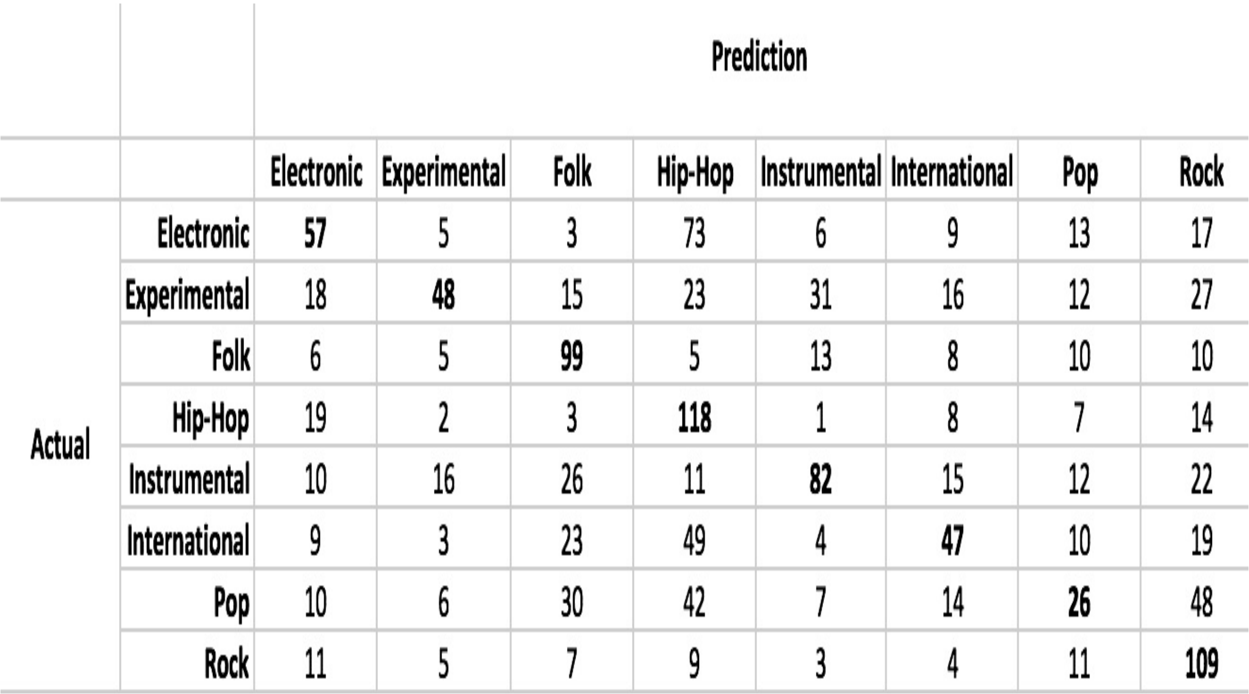
```
private static string[] BuildConfusionMatrix(int[] actual, int[] preds, int numClass)
{
    int[][] matrix = new int[numClass][];
    for(int i = 0; i < numClass; i++)
    {
        matrix[i] = new int[numClass];
    }

    for(int i = 0; i < actual.Length; i++)
    {
        matrix[actual[i]][preds[i]] += 1;
    }

    string[] lines = new string[numClass];
    for(int i = 0; i < matrix.Length; i++)
    {
        lines[i] = string.Join(",", matrix[i]);
    }

    return lines;
}
```

Once you run this code, you are going to get an 8 x 8 matrix, where the rows are the actual and observed genres and the columns are the predicted genres from the models. The following is the confusion matrix for our logistic regression model:



		Prediction							
		Electronic	Experimental	Folk	Hip-Hop	Instrumental	International	Pop	Rock
Actual	Electronic	**79**	8	6	38	8	14	0	30
	Experimental	33	**49**	27	11	11	24	0	35
	Folk	8	5	**94**	3	5	12	0	29
	Hip-Hop	40	2	3	**74**	5	21	1	26
	Instrumental	30	45	33	15	**24**	20	0	27
	International	27	9	28	20	1	**52**	0	27
	Pop	30	7	28	26	2	15	**0**	75
	Rock	6	10	5	11	4	6	0	**117**

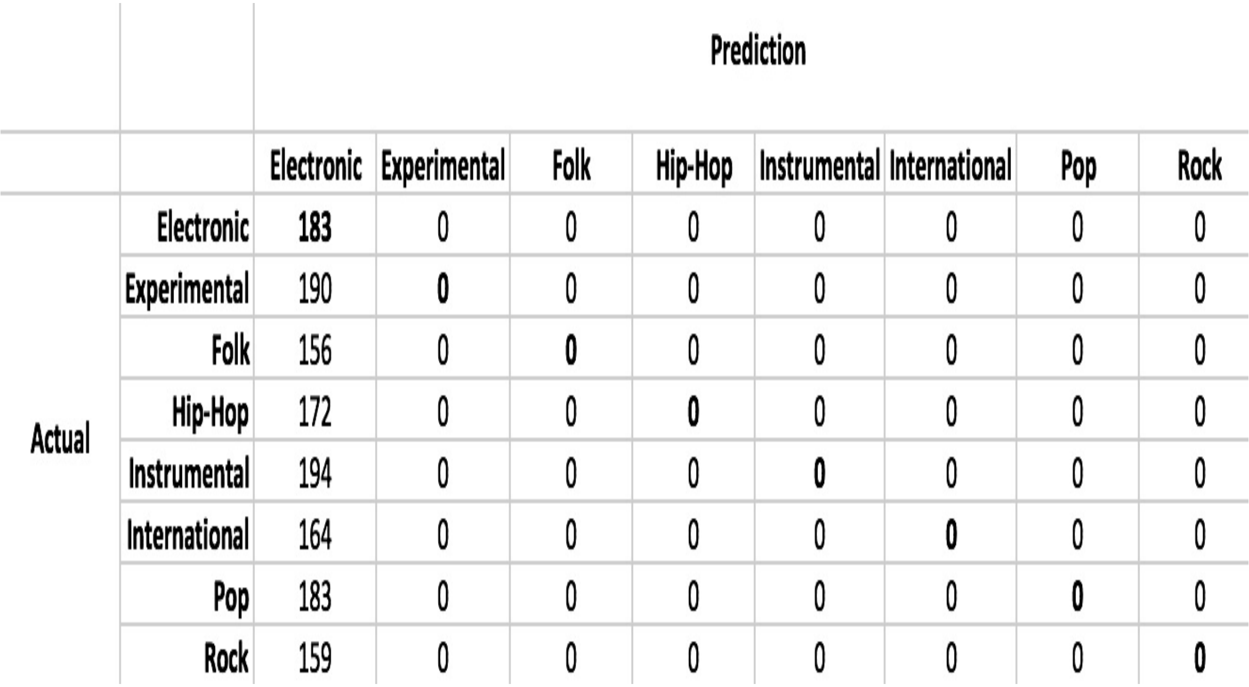
The numbers in bold represent the number of records that the model predicted correctly. For example, this logistic regression model predicted **79** songs correctly as **Electronic** and **33** songs were predicted as **Electronic**, where they were actually **Experimental**. One thing noticeable here is that this logistic regression model did not do so well for predicting Pop songs. It only had one prediction for Pop, but that prediction was wrong and the song was actually a **Hip-Hop** song. Let's now look at the confusion matrix of the Naive Bayes classifier's predictions:



		Prediction							
		Electronic	Experimental	Folk	Hip-Hop	Instrumental	International	Pop	Rock
Actual	Electronic	**57**	5	3	73	6	9	13	17
	Experimental	18	**48**	15	23	31	16	12	27
	Folk	6	5	**99**	5	13	8	10	10
	Hip-Hop	19	2	3	**118**	1	8	7	14
	Instrumental	10	16	26	11	**82**	15	12	22
	International	9	3	23	49	4	**47**	10	19
	Pop	10	6	30	42	7	14	**26**	48
	Rock	11	5	7	9	3	4	11	**109**

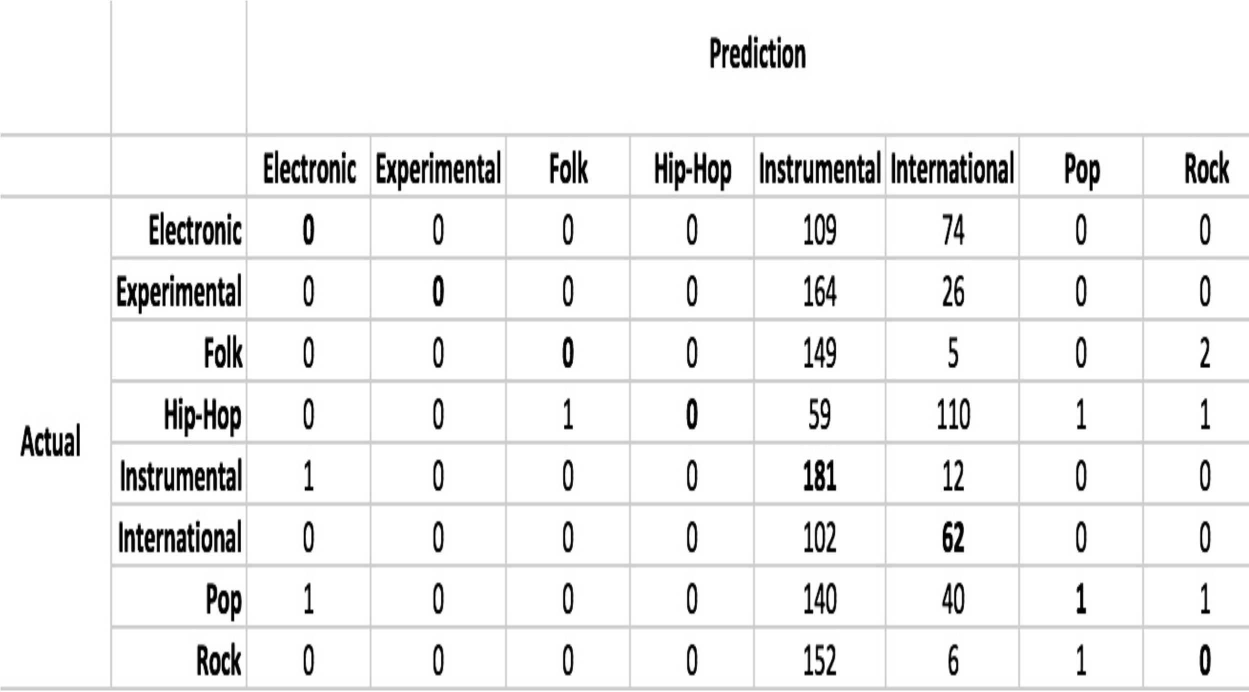
As expected from the accuracy results, the confusion matrix looks better than that for logistic regression. A higher proportion of predictions in each category were right, when compared to the logistic regression classifier. The Naive Bayes classifier seemed to do much better for **Pop** songs as well.

The following is the confusion matrix for the SVM classifier:



		Prediction							
		Electronic	Experimental	Folk	Hip-Hop	Instrumental	International	Pop	Rock
Actual	Electronic	**183**	0	0	0	0	0	0	0
	Experimental	190	**0**	0	0	0	0	0	0
	Folk	156	0	**0**	0	0	0	0	0
	Hip-Hop	172	0	0	**0**	0	0	0	0
	Instrumental	194	0	0	0	**0**	0	0	0
	International	164	0	0	0	0	**0**	0	0
	Pop	183	0	0	0	0	0	**0**	0
	Rock	159	0	0	0	0	0	0	**0**

As expected, the prediction results are not good. The SVM model predicted 100% of the records as **Electronic**. Lastly, let's look at how the meta-model did:



		Prediction							
		Electronic	Experimental	Folk	Hip-Hop	Instrumental	International	Pop	Rock
Actual	Electronic	**0**	0	0	0	109	74	0	0
	Experimental	0	**0**	0	0	164	26	0	0
	Folk	0	0	**0**	0	149	5	0	2
	Hip-Hop	0	0	1	**0**	59	110	1	1
	Instrumental	1	0	0	0	**181**	12	0	0
	International	0	0	0	0	102	**62**	0	0
	Pop	1	0	0	0	140	40	**1**	1
	Rock	0	0	0	0	152	6	1	**0**

This confusion matrix looks slightly better than that of the SVM model. However, the majority of the predictions were either **Instrumental** or **International** and only a handful of records were predicted as other genres.

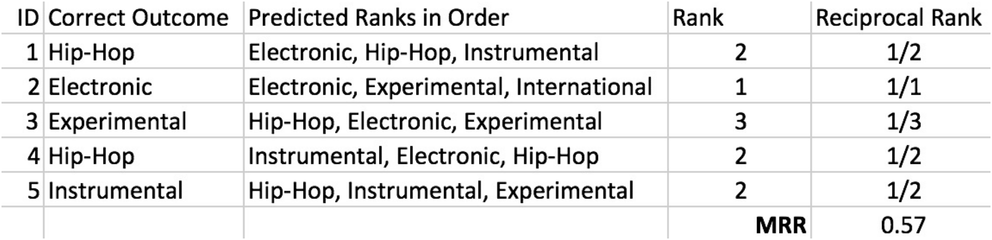
Looking at the confusion matrix is a good way to check misclassifications by models and find out the weaknesses and strengths of the models. These results are well aligned with the accuracy results, where the Naive Bayes classifier outperformed all the other models and the meta-model did not do so well, although it is not the worst among the four models that we have built.

Mean Reciprocal Rank

The next evaluation metric we are going to look at is MRR. MRR can be used where a model produces a list of outcomes and it measures the overall quality of the rankings. Let's first look at the equation:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$$

As you can see, it is an average of the sum of the inverse of the ranks. Consider the following example:



ID	Correct Outcome	Predicted Ranks in Order	Rank	Reciprocal Rank
1	Hip-Hop	Electronic, Hip-Hop, Instrumental	2	1/2
2	Electronic	Electronic, Experimental, International	1	1/1
3	Experimental	Hip-Hop, Electronic, Experimental	3	1/3
4	Hip-Hop	Instrumental, Electronic, Hip-Hop	2	1/2
5	Instrumental	Hip-Hop, Instrumental, Experimental	2	1/2
			MRR	0.57

In the first example, the correct genre was the second in rank, so the reciprocal rank is **1/2**. The second example's correct genre was the first in rank, so the reciprocal rank is **1/1**, which is **1**. Following this process, we can get the reciprocal ranks for all the records and the final MRR value is simply the average of those reciprocal ranks. This tells us the general quality of the rankings. In this example, the **MRR** is **0.57**, which is above 1/2. So, this MRR number suggests that, on average, the correct genres come up within the top two predicted genres by the model.

In order to compute the MRR for our models, we first need to transform the probability output into rankings and then compute the MRR from this transformed model output. The following code snippet shows how we computed the MRR for our models:

```
// Calculate evaluation metrics
int[][] logitTrainPredRanks = GetPredictionRanks(trainProbabilities);
int[][] logitTestPredRanks = GetPredictionRanks(testProbabilities);

double logitTrainMRRScore = ComputeMeanReciprocalRank(
    logitTrainPredRanks,
    output.Where((x, i) => trainSetIDX.Contains(i)).ToArray()
);
double logitTestMRRScore = ComputeMeanReciprocalRank(
    logitTestPredRanks,
    output.Where((x, i) => testSetIDX.Contains(i)).ToArray()
);

Console.WriteLine("\n---- Logistic Regression Classifier ----\n");
Console.WriteLine(String.Format("train MRR score: {0:0.0000}", logitTrainMRRScore));
Console.WriteLine(String.Format("validation MRR score: {0:0.0000}",
logitTestMRRScore));
```

This code uses two helper functions, `GetPredictionRanks` and `ComputeMeanReciprocalRank`. The `GetPredictionRanks` method transforms the probability output of a model into rankings and the `ComputeMeanReciprocalRank` method calculates the MRR from the rankings. The helper function, `GetPredictionRanks`, looks as follows:

```
private static int[][] GetPredictionRanks(double[][] predProbabilities)
{
    int[][] rankOrdered = new int[predProbabilities.Length][];

    for(int i = 0; i< predProbabilities.Length; i++)
    {
        rankOrdered[i] = Matrix.ArgSort<double>(predProbabilities[i]).Reversed();
    }

    return rankOrdered;
}
```

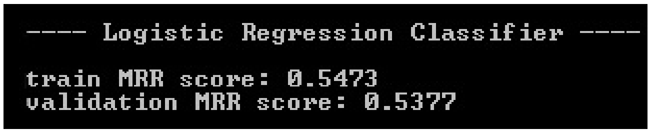
We are using the `Matrix.ArgSort` method from the `Accord.Math` package to rank-order the genres for each record. `Matrix.ArgSort` returns the indexes of the genres after sorting them by probability in ascending order. However, we want them to be sorted in descending order so that the most likely genre comes up as the first in rank. This is why we reverse the order of the sorted indexes using the `Reversed` method.

The helper function, `ComputeMeanReciprocalRank`, looks as follows:

```
private static double ComputeMeanReciprocalRank(int[][] rankOrderedPreds, int[]
actualClasses)
{
    int num = rankOrderedPreds.Length;
    double reciprocalSum = 0.0;

    for(int i = 0; i < num; i++)
    {
        int predRank = 0;
        for(int j = 0; j < rankOrderedPreds[i].Length; j++)
        {
            if(rankOrderedPreds[i][j] == actualClasses[i])
            {
                predRank = j + 1;
            }
        }
        reciprocalSum += 1.0 / predRank;
    }

    return reciprocalSum / num;
}
```

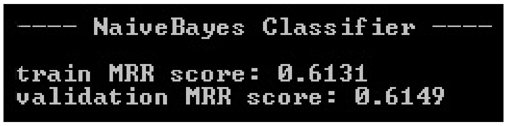
This is our implementation of the equation for the MRR calculation that we discussed previously. This method iterates through each record and gets the rank of the correct genre. Then, it reciprocates the rank, sums all of the reciprocals, and finally divides this sum by the number of records to get the MRR number.

Let's start looking at the MRR scores for the models that we have built so far. The following output shows the MRR scores for the `Logistic Regression Classifier`:



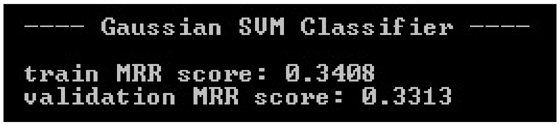
```
---- Logistic Regression Classifier ----

train MRR score: 0.5473
validation MRR score: 0.5377
```

The in-sample and out-of-sample MRR scores for the Naive Bayes classifier look as follows:



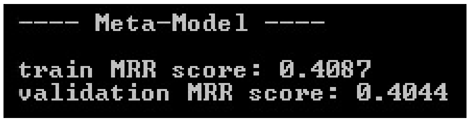
```
---- NaiveBayes Classifier ----

train MRR score: 0.6131
validation MRR score: 0.6149
```

And the results for the SVM classifier are as follows:



```
---- Gaussian SVM Classifier ----

train MRR score: 0.3408
validation MRR score: 0.3313
```

Lastly, the MRR scores for the meta-model look as follows:



```
---- Meta-Model ----

train MRR score: 0.4087
validation MRR score: 0.4044
```

From these outputs, we can see that the Naive Bayes classifier has the best MRR scores at around `0.61`, while the SVM classifier has the worst MRR scores at around `0.33`. The meta-model has MRR scores at around `0.4`. This is aligned with the results we have found from looking at the prediction accuracy and confusion matrix in the previous steps. From these MRR scores, we can see that the correct genres generally fall within the top two ranks for the Naive Bayes classifier. On the other hand, the correct genres typically come up as the third from the top for the SVM classifier and within the top three for the meta-model. As you can see from these cases, we can understand the overall quality of the rankings by looking at the MRR measures.

Summary

In this chapter, we built our first recommendation model to rank-order the likelihood of each of the outcomes. We started this chapter by defining the problems that we were going to solve and the modeling and the evaluation approaches that we were going to use. Then, we looked at the distributions of the variables in our sample set. First, we looked at how well the target variables were distributed among different classes or genres and noticed that it was a well-balanced sample set with no one genre taking up the majority of the samples in our dataset. Then, we looked at the distributions of the audio features. In this project, we focused mainly on MFCCs and their statistical distributions, such as kurtosis, skewness, min, and max. By looking at the quartiles and the scatter plots of these features, we confirmed that the feature distributions differed among the music genres.

During our model-building step, we experimented with three learning algorithms: logistic regression, SVM, and Naive Bayes. Since we were building multi-class classification models, we had to use different learning algorithms from previous chapters. We learned how to use the `MultinomialLogisticRegression` and `MulticlassSupportVectorMachine` classes in the Accord.NET framework, as well as when to use `NormalDistribution` for `NaiveBayesLearning`. We then discussed how we could build a meta-model that ensembled the prediction results from the base learning models to improve the predictive power of the ML models. Lastly, we discussed how evaluating ranking models differed from other classification models and looked at the accuracy, confusion matrix, and MRR metrics to evaluate our ML models.

In the next chapter, we are going to use a hand-written digit image dataset to build a classifier that classifies each image into the corresponding digit. We are going to discuss some techniques to reduce the dimensions of the feature set and how to apply them to the image dataset. We will also discuss how to build a neural network in C# using the Accord.NET framework, which is the backbone of deep learning.

Handwritten Digit Recognition

We have looked at how to build recommendation models using multi-class classification models. In this chapter, we are going to expand our knowledge and experience of building multi-class classification models with an image dataset. Image recognition is a well-known **machine learning** (**ML**) problem and is one of the topics that are actively being researched. One image recognition problem that has high applicability to our lives is recognizing handwritten letters and digits. A good example of the application of a handwritten image recognition system is the address recognition system that is used at post offices. Using such a technology, post offices can now automatically and more quickly identify addresses that are written by hand, and expedite and improve overall mailing services.

In this chapter, we are going to build machine learning models for handwritten digit recognition. We are going to start with a dataset that contains grayscale pixel-by-pixel information about over 40,000 handwritten digit images. We will look at the distributions of the values in each pixel and discuss how sparse this grayscale image dataset is. Then, we are going to discuss when and how to apply dimensionality reduction techniques, more specifically **Principal Component Analysis** (**PCA**), and how we can benefit from this technique for our image recognition project. We will be exploring different learning algorithms, such as logistic regression and Naive Bayes, and will also cover how to build an **Artificial Neural Network** (**ANN**), which forms the backbone of deep learning technologies, using the Accord.NET framework. Then, we will compare the prediction performances of these ML models by looking at various evaluation metrics, and discuss which model performed the best for the handwritten digit recognition project.

In this chapter, we will cover the following topics:

- Problem definition for the handwritten digit recognition project
- Data analysis for an image dataset
- Feature engineering and dimensionality reduction

- ML models for handwritten digit recognition
- Evaluating multi-class classification models

Problem definition

Image recognition technology can be applied to, and can be found easily in, our daily lives. At post offices, image recognition systems are used to programmatically understand addresses that are written by hand. Social network services, such as Facebook, use image recognition technology for automatic people tag suggestions, for instance, when you want to tag people in your photos. Also, as briefly mentioned in the very first chapter of this book, Microsoft's Kinect uses image recognition technology for its motion-sensing games. Of these real-life applications, we are going to experiment with building a handwritten digit recognition system. As you can imagine, such digit image recognition models and systems can be used for automated handwritten address recognition at post offices. Before we had this ability to teach machines to identify and understand handwritten digits, people had to go through and look at each letter to find out the destination and the origin of individual letters. However, now that we can train machines to understand handwritten addresses, mailing processes have become much easier and faster.

In order to build a handwritten digit recognition model, we are going to use the **MNIST** dataset, which has over 60,000 handwritten digit images. The **MNIST** dataset contains 28 x 28 images that are in grayscale. You can find more information at this link: `http://yann.lecun.com/exdb/mnist/`. For this project, we will be using a cleaned and processed MNIST dataset that can be found at this link: `https://www.kaggle.com/c/digit-recognizer/data`. With this data, we will first look at how the digits are distributed across the dataset, and how sparse the feature set is. Then, we are going to use PCA for dimensionality reduction and to visualize the differences in the distributions of features among different classes. With this PCA-transformed data, we are going to train a few ML models to compare their prediction performances. On top of logistic regression and Naive Bayes classification algorithms, we are going to experiment with the ANN, as it is known to work well for image datasets. We will look at accuracy, precision versus recall, and **area under the curve** (**AUC**), to compare the prediction performances among different machine learning models.

To summarize our problem definition for the handwritten digit recognition project:

- What is the problem? We need a handwritten digit recognition model that can classify each handwritten image into a corresponding digit class, so that it can be used for applications such as the address recognition system.
- Why is it a problem? Without such a model, it takes an enormous amount of human labor to identify and organize letters by addresses. If we have a technology that can recognize handwritten digits that are written on letters, it can significantly reduce the amount of human labor required for the same task.
- What are some of the approaches to solving this problem? We are going to use publicly available data that contains numerous examples of handwritten digit images. With this data, we are going to build machine learning models that can classify each image into one of 10 digits.
- What are the success criteria? We want a machine learning model that accurately classifies each image with the corresponding digit. Since this model will eventually be used for address recognition, we want high precision rates, even if we have to sacrifice recall rates.

Data analysis for the image dataset

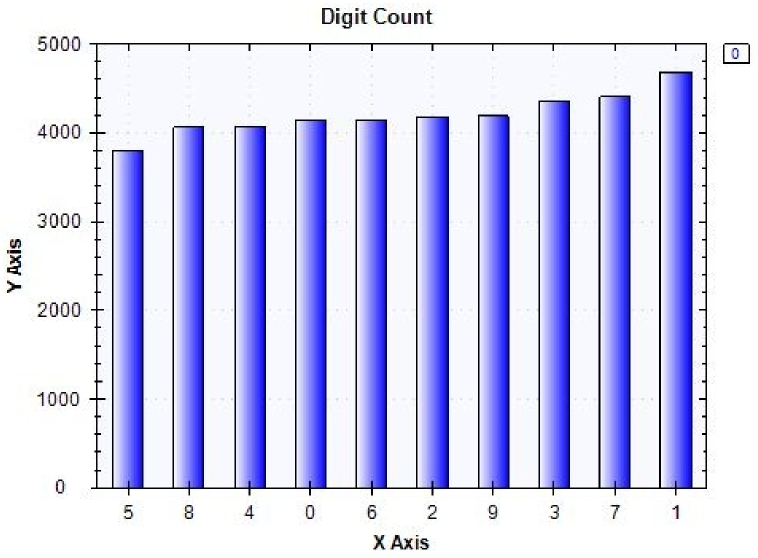
Let's start looking into this image dataset. As mentioned in the previous section, we will be using the data from this link: `https://www.kaggle.com/c/digit-recognizer/data`. You can download the `train.csv` data from the link and store it in a place from where you can load it into your C# environment.

Target variable distribution

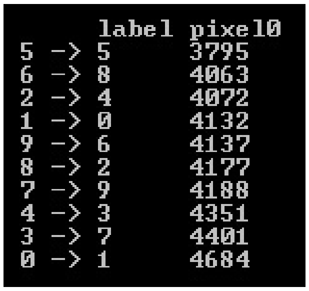
The first thing we are going to look at is the distribution of the target variables. Our target variable is encoded in the `label` column, which can take values between 0 and 9, and represents the digit that the image belongs to. The following code snippet shows how we aggregated the data by the target variable and counted the number of examples for each digit:

```
var digitCount =
featuresDF.AggregateRowsBy<string, int>(
new string[] { "label" },
new string[] { "pixel0" },
x => x.ValueCount
).SortRows("pixel0");

digitCount.Print();

var barChart = DataBarBox.Show(
digitCount.GetColumn<string>("label").Values.ToArray(),
digitCount["pixel0"].Values.ToArray()
).SetTitle(
"Digit Count"
);
```

As in other chapters, we used the `AggregateRowsBy` method in Deedle's data frame to aggregate data by the target variable, `label`, count the number of records in each label, and sort by the counts. Similar to previous chapters, we are using the `DataBarBox` class to display a bar plot of target variable distributions in the dataset. The following is the bar plot that you will see when you run this code:



In the console output, you will see the following:



```
        label pixel0
5  -> 5      3795
6  -> 8      4063
2  -> 4      4072
1  -> 0      4132
9  -> 6      4137
8  -> 2      4177
7  -> 9      4188
4  -> 3      4351
3  -> 7      4401
0  -> 1      4684
```

As you can see from the bar plot and this console output, the digit `1`, occurs the most in the dataset, and the digit `5`, occurs the least. However, there is no one class that takes the majority of the examples in the dataset, and the target variables are pretty well balanced and spread across different classes.

Handwritten digit images

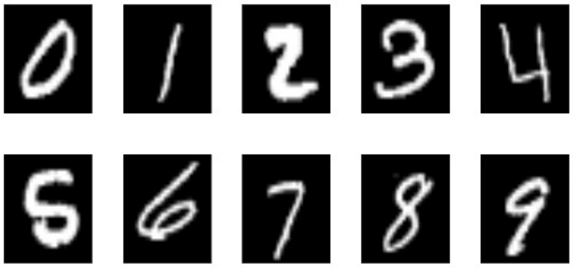
Before we start looking into the feature set, let's first look at actual images of handwritten digits. In each record of our dataset, we have the grayscale values for 784 pixels for each of the 28 x 28 images. In order to build an image from this flattened dataset, we need to first convert each array of 784-pixel values into a two-dimensional array. The following code shows the helper function we wrote to create an image from a flattened array:

```
private static void CreateImage(int[] rows, string digit)
{
    int width = 28;
    int height = 28;
    int stride = width * 4;
    int[,] pixelData = new int[width, height];

    for (int i = 0; i < width; ++i)
    {
        for (int j = 0; j < height; ++j)
        {
            byte[] bgra = new byte[] { (byte)rows[28 * i + j], (byte)rows[28 * i + j],
(byte)rows[28 * i + j], 255 };
            pixelData[i, j] = BitConverter.ToInt32(bgra, 0);
        }
    }

    Bitmap bitmap;
    unsafe
    {
        fixed (int* ptr = &pixelData[0, 0])
        {
            bitmap = new Bitmap(width, height, stride, PixelFormat.Format32bppRgb, new
IntPtr(ptr));
        }
    }
    bitmap.Save(
        String.Format(@"\\Mac\Home\Documents\c-sharp-machine-learning\ch.8\input-data\
{0}.jpg", digit)
    );
}
```

As you can see from this code, it first initializes a two-dimensional integer array, `pixelData`, which is going to store the pixel data. Since we know each image is a 28 x 28 image, we are going to take the first 28 pixels in the flattened data as the first row in the two-dimensional integer array, the second set of 28 pixels as the second row, and so forth. Inside the `for` loop, we are converting the value of each pixel into a **Blue-Green-Red-Alpha** (**BGRA**) byte array, named `bgra`. As we know the images are in grayscale, we can use the same value for blue, green, and red components. Once we have converted the flattened pixel data into a 28 x 28 two-dimensional integer array, we can now build images of the handwritten digit images. We are using the `Bitmap` class to reconstruct these handwritten digit images. The following code shows how we used this helper function to build images for each digit:

```
ISet<string> exportedLabels = new HashSet<string>();
for(int i = 0; i < featuresDF.RowCount; i++)
{
    exportedLabels.Add(featuresDF.Rows[i].GetAs<string>("label"));

    CreateImage(
        featuresDF.Rows[i].ValuesAll.Select(x => (int)x).Where((x, idx) => idx >
0).ToArray(),
        featuresDF.Rows[i].GetAs<string>("label")
    );

    if(exportedLabels.Count() >= 10)
    {
        break;
    }
}
```

When you run this code, you will see the following images being stored on your local drive:



You can use the same code to generate more images, which will help you better understand what raw images of handwritten digits look like.

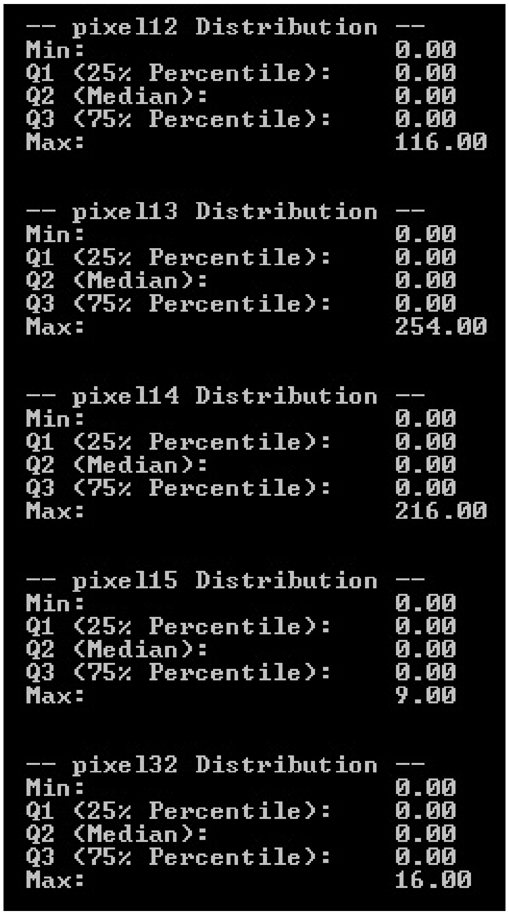
Image features - pixels

Let's now look at image features. In our dataset, we have integer values for each pixel in each image that represent a grayscale value. It will be helpful to understand the ranges of values each pixel can take, and whether we can find any noticeable differences in the distributions of that pixel data among different handwritten digit classes.

We will first take a look at individual distributions of pixel data. The following code snippet shows how you can calculate the quartiles for each pixel in our dataset:

```
List<string> featureCols = new List<string>();
foreach (string col in featuresDF.ColumnKeys)
{
    if (featureCols.Count >= 20)
    {
        break;
    }

    if (col.StartsWith("pixel"))
    {
        if (featuresDF[col].Max() > 0)
        {
            featureCols.Add(col);

            Console.WriteLine(String.Format("\n\n-- {0} Distribution -- ", col));
            double[] quantiles = Accord.Statistics.Measures.Quantiles(
                featuresDF[col].ValuesAll.ToArray(),
                new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
            );
            Console.WriteLine(
                "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
                quantiles[0], quantiles[1], quantiles[2], quantiles[3], quantiles[4]
            );
        }

    }
}
```

Similar to the case previous chapters, we used the `Quantiles` method in `Accord.Statistics.Measures` to get the quartiles for each pixel. As you might recall from previous chapters, quartiles are the values that separate the data into four sections. In other words, the first quartile (`Q1`) represents the 25% percentile that is the middle point between the minimum value and the median value. The second quartile (`Q2`) represents the median value, and the third quartile (`Q3`) represents the 75% percentile that is the middle point between the median and the maximum. In this code example, we are only computing quartiles for the first 20 pixels that have values other than 0, as you can see in lines 4-7, and in line 11. When you run this code, you will get an output that looks like the following:



```
-- pixel12 Distribution --
Min:                      0.00
Q1 (25% Percentile):      0.00
Q2 (Median):              0.00
Q3 (75% Percentile):      0.00
Max:                      116.00

-- pixel13 Distribution --
Min:                      0.00
Q1 (25% Percentile):      0.00
Q2 (Median):              0.00
Q3 (75% Percentile):      0.00
Max:                      254.00

-- pixel14 Distribution --
Min:                      0.00
Q1 (25% Percentile):      0.00
Q2 (Median):              0.00
Q3 (75% Percentile):      0.00
Max:                      216.00

-- pixel15 Distribution --
Min:                      0.00
Q1 (25% Percentile):      0.00
Q2 (Median):              0.00
Q3 (75% Percentile):      0.00
Max:                      9.00

-- pixel32 Distribution --
Min:                      0.00
Q1 (25% Percentile):      0.00
Q2 (Median):              0.00
Q3 (75% Percentile):      0.00
Max:                      16.00
```

Here, we are only showing the first five distributions. As you can see from this output, the majority of the pixel values are 0. If you look at the images that we reconstructed in the previous section, the majority of the pixels in the image are black and only a subset of the pixels are used to show digits. These pixels in black are encoded as `0` in our pixel data, and thus it is expected that many pixels have 0 values for the corresponding image.

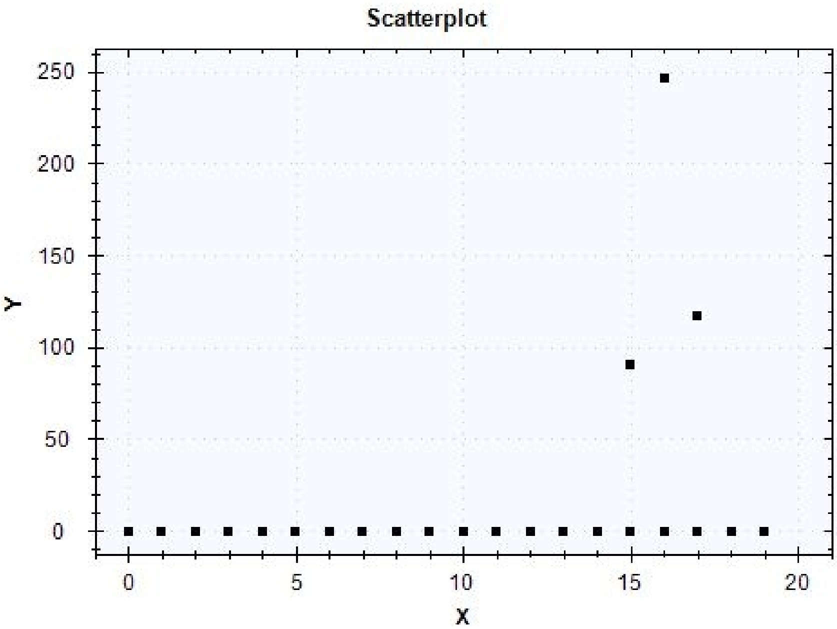
Let's build some scatter plots, so that we can understand this data better visually. The following code builds scatter plots of distributions of the first 20 non-zero pixel features for each handwritten digit:

```
string[] featureColumns = featureCols.ToArray();
```

```
foreach (string label in digitCount.GetColumn<string>("label").Values)
{
    var subfeaturesDF = featuresDF.Rows[
        featuresDF.GetColumn<string>("label").Where(x => x.Value == label).Keys
    ].Columns[featureColumns];

    ScatterplotBox.Show(
        BuildXYPairs(
            subfeaturesDF.Columns[featureColumns].ToArray2D<double>(),
            subfeaturesDF.RowCount,
            subfeaturesDF.ColumnCount
        )
    ).SetTitle(String.Format("Digit: {0} - 20 sample Pixels", label));
}
```

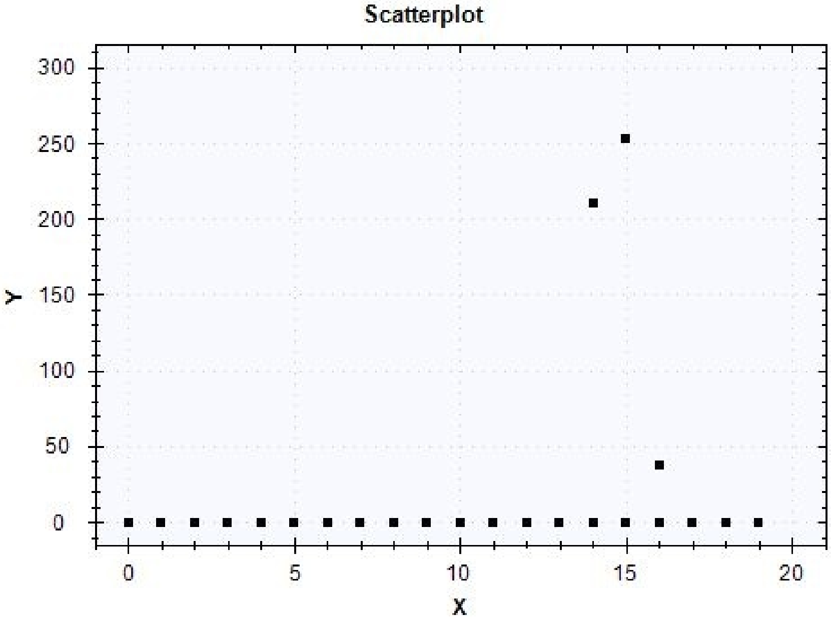
If you look closely at this code, we first build a `featureColumns` string array from the `featureCols`, `List` object. The `List` object, `featureCols`, is a list of the first 20 pixels that have values other than 0, and this was built from the previous code when we were computing quartiles. We are using the same helper function, `BuildXYPairs`, that we used in the previous chapter to transform the data frame into an array of x-y pairs, where the `x` values are the indexes of each pixel and the `y` values are the actual pixel value. Using this helper function, we use the `ScatterplotBox` class to display a scatter plot that shows the pixel distribution for each of the 20 sample pixels.

The following is a scatter plot for the 0 digit:



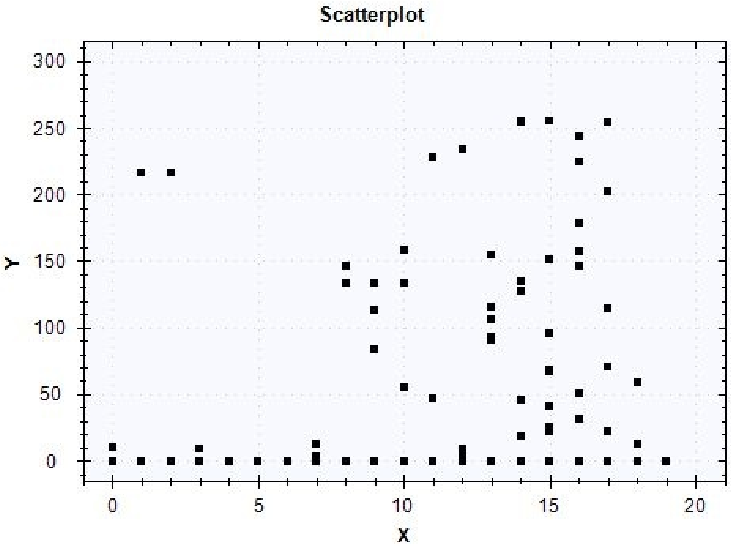
The majority of the first 20 pixels have 0 values for all the images in the 0 digit class. Of those 20 pixels that we show in this scatter plot, there are only three pixels that have values other than 0. Let's look at the distributions of these pixels for a different digit class.

The following scatter plot is for the 1 digit class:



Similar to the case of the 0 digit class, of those 20 pixels that we show here, the majority have 0 values and only three pixels have values other than 0. Compared to the previous scatter plot for of the 0 digit class, the distributions of the pixel data are slightly different for the 1 digit class.

The following is for the 2 digit class:

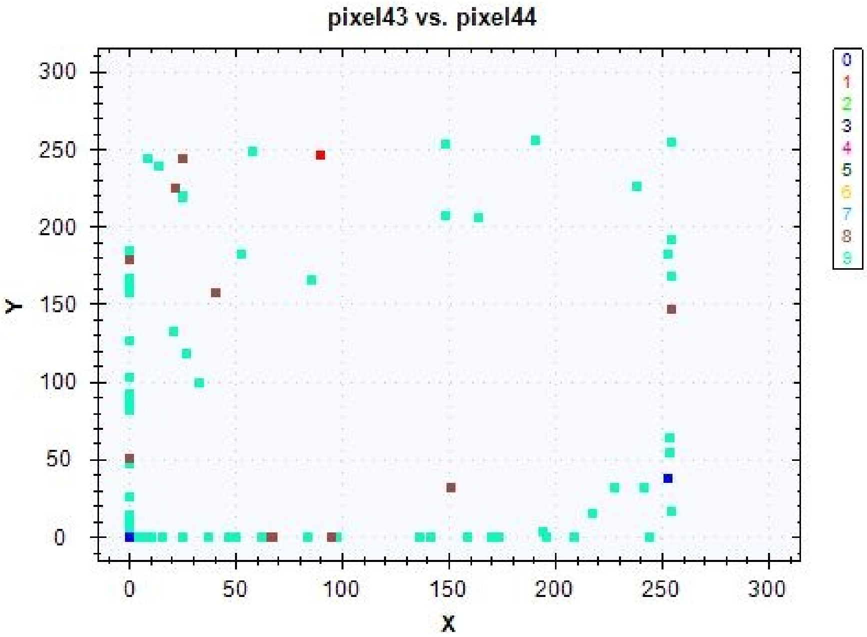


This scatter plot shows quite different distributions for the 20 pixels that we show here. The majority of those 20 pixels have values ranging between 0 and 255, and only a few have 0 values for all the images. This kind of difference in the distributions of the feature set will help our ML models learn how to correctly classify handwritten digits.

Lastly, we are going to look at one more scatter plot, where we will see how the target variables are distributed across two different pixels. We used the following code to generate a sample two-dimensional scatter plot:

```
double[][] twoPixels = featuresDF.Columns[
    new string[] { featureColumns[15], featureColumns[16] }
].Rows.Select(
    x => Array.ConvertAll<object, double>(x.Value.ValuesAll.ToArray(), o =>
Convert.ToDouble(o))
).ValuesAll.ToArray();

ScatterplotBox.Show(
    String.Format("{0} vs. {1}", featureColumns[15], featureColumns[16]),
    twoPixels,
    featuresDF.GetColumn<int>("label").Values.ToArray()
);
```

For illustration purposes, we chose the fifteenth and sixteenth indexed features, which turn out to be `pixel43` and `pixel44`. When you run this code, you will see the following scatter plot:



We can see some distinctions among different classes, but since the majority of the pixel values for both `pixel43` and `pixel44` are 0, it is quite difficult to draw a clear distinction among different target classes by looking at this scatter plot. In the next section, we are going to look at how to use PCA and its principal components to create another version of this scatter plot that can help us identify a clearer distinction among different target classes when we visualize the data.

The full code for this data analysis step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.8/DataAnalyzer.cs`.

Feature engineering and dimensionality reduction

So far, we have looked at the distributions of the target variables and pixel data. In this section, we are going to start discussing building train and test sets for our ML modeling step, and then how we can use PCA for dimensionality reduction and to visualize data using the principal components.

Splitting the sample set into train versus test sets

The first task we are going to do in this step is to randomly split our dataset into train and test sets. Let's first look at the code:

```
double trainSetProportiona = 0.7;

var rnd = new Random();
var trainIdx = featuresDF.RowKeys.Where((x, i) => rnd.NextDouble() <=
trainSetProportiona);
var testIdx = featuresDF.RowKeys.Where((x, i) => !trainIdx.Contains(i));

var trainset = featuresDF.Rows[trainIdx];
var testset = featuresDF.Rows[testIdx];

var trainLabels = trainset.GetColumn<int>("label").Values.ToArray();

string[] nonZeroPixelCols = trainset.ColumnKeys.Where(x => trainset[x].Max() > 0 &&
!x.Equals("label")).ToArray();

double[][] data = trainset.Columns[nonZeroPixelCols].Rows.Select(
    x => Array.ConvertAll<object, double>(x.Value.ValuesAll.ToArray(), o =>
Convert.ToDouble(o))
).ValuesAll.ToArray();
```

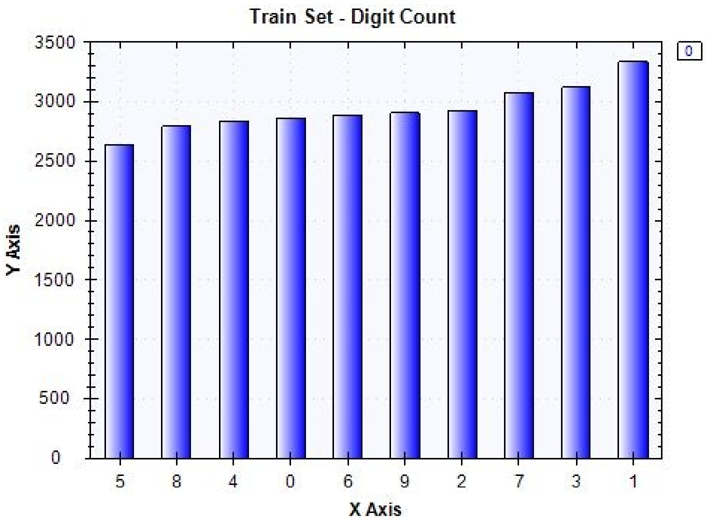
As you can see from the preceding code, we are taking roughly about 70% of our data for training, and the rest for testing. Here, we are using the `Random` class to generate random numbers to split the sample set into train and test sets using the indexes of the records. Once we have built train and test sets, we are removing columns or pixels that have 0 values for all the images (line 12). This is because if a feature doesn't vary among different target classes, it doesn't have any information about those target classes for ML models to learn.

Now that we have train and test sets, let's check on the distributions of target classes in both train and test sets. The following code can be used for the aggregation:

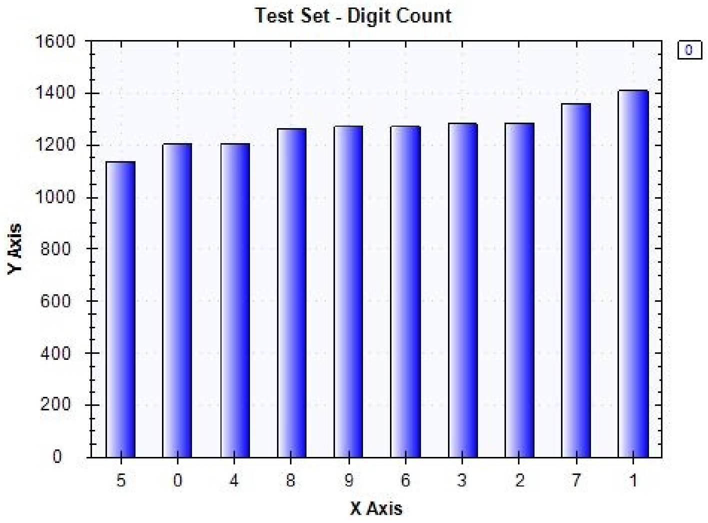
```
var digitCount = trainset.AggregateRowsBy<string, int>(
    new string[] { "label" },
    new string[] { "pixel0" },
    x => x.ValueCount
).SortRows("pixel0");

digitCount.Print();
```

```
var barChart = DataBarBox.Show(
    digitCount.GetColumn<string>("label").Values.ToArray(),
    digitCount["pixel0"].Values.ToArray()
).SetTitle(
    "Train Set - Digit Count"
);

digitCount = testset.AggregateRowsBy<string, int>(
    new string[] { "label" },
    new string[] { "pixel0" },
    x => x.ValueCount
).SortRows("pixel0");

digitCount.Print();

barChart = DataBarBox.Show(
    digitCount.GetColumn<string>("label").Values.ToArray(),
    digitCount["pixel0"].Values.ToArray()
).SetTitle(
    "Test Set - Digit Count"
);
```

When you run this code, you will see the following plot for the target variable distribution in the train set:



And, the following is what we see for the test set:



These distributions look similar to what we saw in the data analysis step, when we analyzed the target variable distribution in the overall dataset. Let's now start discussing how we can apply PCA to our train set.

Dimensionality reduction by PCA

We saw that many of our feature or pixel values are 0, when we were analyzing our data. In such cases, applying PCA can be helpful for reducing the dimensions of the data, while minimizing the loss of information from the reduced dimensions. Simply put, PCA is used to explain a dataset and its structure through linear combinations of the original features. So, each principal component is a linear combination of the features. Let's start looking at how we can run PCA in C#, using the Accord.NET framework.

The following is how you can initialize and train a PCA:

```
var pca = new PrincipalComponentAnalysis(
    PrincipalComponentMethod.Standardize
);
pca.Learn(data);
```

Once a `PrincipalComponentAnalysis` is trained with the data, it contains all the information about the linear combinations for each principal component and can be applied to transform other data. We used `PrincipalComponentMethod.Standardize` to standardize our data before applying PCA. This is because PCA is sensitive to the scale of each feature. So, we want to standardize our dataset before applying PCA.

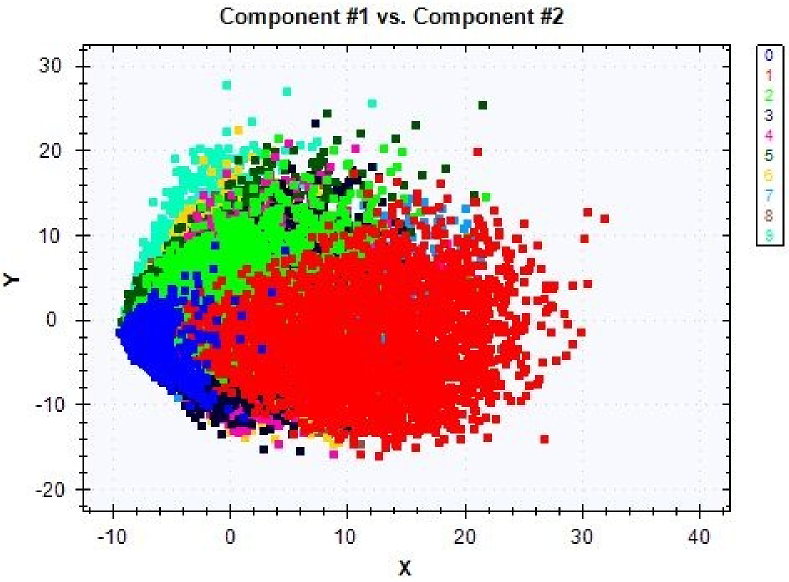
In order to PCA-transform other data, you can use the `Transform` method, as shown in the following code snippet:

```
double[][] transformed = pca.Transform(data);
```

Now that we have learned how we can apply PCA to our dataset, let's look at the first two principal components and see if we can find any noticeable patterns in the target variable distributions. The following code shows how we can build a scatter plot of the first two components with target classes color-coded:

```
double[][] first2Components = transformed.Select(x => x.Where((y, i) => i <
2).ToArray()).ToArray();

ScatterplotBox.Show("Component #1 vs. Component #2", first2Components, trainLabels);
```

Once you run this code, you will see the following scatter plot:



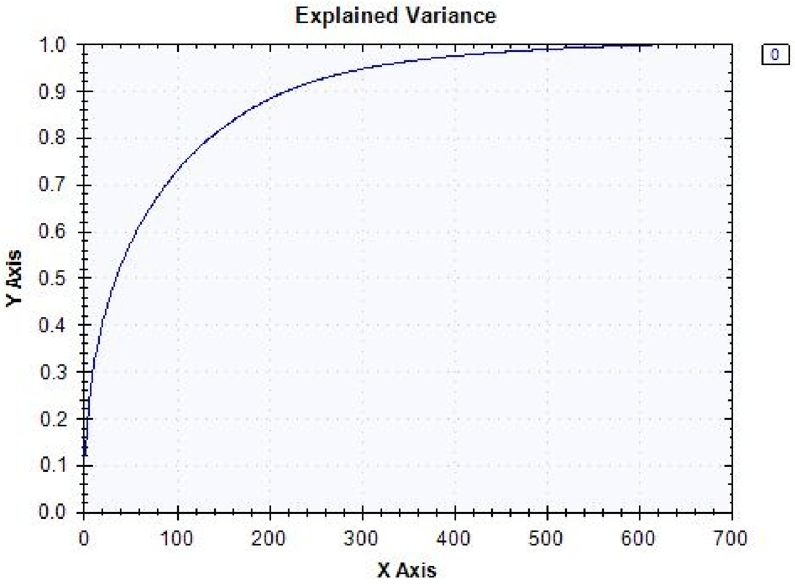
When you compare this chart with the one between `pixel43` and `pixel44` that we looked at during the data analysis step, this looks quite different. From this scatter plot of the first two principal components, we can see that the target classes are more discernible. Although it is not perfectly separable from these two components, we can see that if we combine more components into our analysis and modeling, it will get easier to separate one target class from another.

Another important aspect of PCA that we should look at is the amount of variance explained by each principal component. Let's take a look at the following code:

```
DataSeriesBox.Show(
    pca.Components.Select((x, i) => (double)i),
    pca.Components.Select(x => x.CumulativeProportion)
).SetTitle("Explained Variance");

System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "explained-variance.csv"),
    pca.Components.Select((x, i) => String.Format("{0},{1:0.0000}", i,
x.CumulativeProportion))
);
```

We can retrieve the cumulative proportion of the variance in our data explained by each PCA component by using the `CumulativeProportion` property. In order to get the individual proportion explained by each PCA component, you can use the `Proportion` property of each PCA component. Then, we will use the `DataSeriesBox` class to plot a line chart to display the cumulative proportions of the variance explained by each component.

When you run this code, it will produce the following plot:



As you can see from this plot, about 90% of the variance in the dataset can be explained by the first 200 components. With 600 components, we can explain almost 100% of the variance in our dataset. Compared to the total of 784 pixels we had as our features in the raw dataset, this is a big reduction in the dimension of our data. Depending on how much variance you want to capture for your ML models, you can use this chart to decide the number of components that is most suitable for your modeling process.

Finally, we need to export the train and test sets, so that we can use them for the following model building step. You can use the following code to export the PCA-transformed train and test sets:

```
Console.WriteLine("exporting train set...");
var trainTransformed = pca.Transform(
    trainset.Columns[nonZeroPixelCols].Rows.Select(
        x => Array.ConvertAll<object, double>(x.Value.ValuesAll.ToArray(), o =>
Convert.ToDouble(o))
    ).ValuesAll.ToArray()
);

System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "pca-train.csv"),
    trainTransformed.Select((x, i) => String.Format("{0},{1}", String.Join(",", x),
trainset["label"].GetAt(i)))
);

Console.WriteLine("exporting test set...");
var testTransformed = pca.Transform(
    testset.Columns[nonZeroPixelCols].Rows.Select(
        x => Array.ConvertAll<object, double>(x.Value.ValuesAll.ToArray(), o =>
Convert.ToDouble(o))
    ).ValuesAll.ToArray()
);
System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "pca-test.csv"),
    testTransformed.Select((x, i) => String.Format("{0},{1}", String.Join(",", x),
testset["label"].GetAt(i)))
);
```

The full code for this feature engineering and dimensionality reduction step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.8/FeatureEngineering.cs`.

ML models for handwritten digit recognition

Now that we have everything ready for building ML models, let's start building those models. In this section, we will cover how to sub-select the features based on the PCA results and then discuss how we can build logistic regression and Naive Bayes classifiers for the handwritten digit recognition model. We are going to introduce a new learning model, the neural network, and explain how to build one for this project, using the Accord.NET framework.

Loading data

The first step in building a ML model for handwritten digit recognition is to load the data that we built from the previous section. You can use the following code to load the train and test sets that we created previously:

```
// Load the data into a data frame
string trainDataPath = Path.Combine(dataDirPath, "pca-train.csv");
Console.WriteLine("Loading {0}\n\n", trainDataPath);
var trainDF = Frame.ReadCsv(
    trainDataPath,
    hasHeaders: false,
    inferTypes: true
);

string testDataPath = Path.Combine(dataDirPath, "pca-test.csv");
Console.WriteLine("Loading {0}\n\n", testDataPath);
var testDF = Frame.ReadCsv(
    testDataPath,
    hasHeaders: false,
    inferTypes: true
);

string[] colnames = trainDF.ColumnKeys.Select(
    (x, i) => i < trainDF.ColumnKeys.Count() - 1 ? String.Format("component-{0}", i +
1) : "label"
).ToArray();

trainDF.RenameColumns(colnames);
testDF.RenameColumns(colnames);
```

For our experimentation with different models in this chapter, we will be using the principal components that cumulatively explain about 70% of the variance in our dataset. Take a look at the following code to see how we filtered for the components of our interest:

```
// Capturing 70% of the variance
string[] featureCols = colnames.Where((x, i) => i <= 90).ToArray();


double[][] trainInput = BuildJaggedArray(
    trainDF.Columns[featureCols].ToArray2D<double>(), trainDF.RowCount,
featureCols.Length
);
int[] trainOutput = trainDF.GetColumn<int>("label").ValuesAll.ToArray();

double[][] testInput = BuildJaggedArray(
    testDF.Columns[featureCols].ToArray2D<double>(), testDF.RowCount,
featureCols.Length
);
int[] testOutput = testDF.GetColumn<int>("label").ValuesAll.ToArray();
```

As you can see in the first line of this code, we are taking the first 91 components (up to the ninetieth index) as the features for our models. If you recall from the previous step or look at the plot for the cumulative variance proportion explained by the components, you will see that the first 91 components capture about 70% of the variance in our dataset. Then, we create a two-dimensional array of doubles that we will use for training and testing our ML models. The following code shows the helper function, `BuildJaggedArray`, that we wrote to convert a data frame into a two-dimensional array:

```
private static double[][] BuildJaggedArray(double[,] ary2d, int rowCount, int colCount)
{
    double[][] matrix = new double[rowCount][];
    for(int i = 0; i < rowCount; i++)
    {
        matrix[i] = new double[colCount];
        for(int j = 0; j < colCount; j++)
        {
            matrix[i][j] = double.IsNaN(ary2d[i, j]) ? 0.0 : ary2d[i, j];
        }
    }
    return matrix;
}
```

Logistic regression classifier

The first learning algorithm we are going to experiment with for handwritten digit recognition is logistic regression. We wrote a method, named `BuildLogitModel`, which takes in the inputs and outputs to the model, trains a logistic regression classifier, and then evaluates the performance. The following code shows how this method is written:

```
private static void BuildLogitModel(double[][] trainInput, int[] trainOutput, double[]
[] testInput, int[] testOutput)
{
    var logit = new MultinomialLogisticLearning<GradientDescent>()
    {
        MiniBatchSize = 500
    };
    var logitModel = logit.Learn(trainInput, trainOutput);

    int[] inSamplePreds = logitModel.Decide(trainInput);
    int[] outSamplePreds = logitModel.Decide(testInput);

    // Accuracy
    double inSampleAccuracy = 1 - new ZeroOneLoss(trainOutput).Loss(inSamplePreds);
    double outSampleAccuracy = 1 - new ZeroOneLoss(testOutput).Loss(outSamplePreds);
    Console.WriteLine("* In-Sample Accuracy: {0:0.0000}", inSampleAccuracy);
    Console.WriteLine("* Out-of-Sample Accuracy: {0:0.0000}", outSampleAccuracy);

    // Build confusion matrix
    int[][] confMatrix = BuildConfusionMatrix(
        testOutput, outSamplePreds, 10
    );
    System.IO.File.WriteAllLines(
        Path.Combine(
            @"<path-to-dir>",
            "logit-conf-matrix.csv"
        ),
        confMatrix.Select(x => String.Join(",", x))
    );

    // Precision Recall
    PrintPrecisionRecall(confMatrix);
    DrawROCCurve(testOutput, outSamplePreds, 10, "Logit");
}
```

Similar to the previous chapter, we are using the `MultinomialLogisticLearning` class to train a logistic regression classifier. Once this model is trained, we start evaluating by various evaluation metrics, which we will discuss in more detail in the following section.

Naive Bayes classifier

The second model we are going to experiment with is a Naive Bayes classifier. Similar to the previous case involving the logistic regression classifier, we wrote a helper function, `BuildNBModel`, that takes in the inputs and outputs, trains a Naive Bayes classifier, and then evaluates the trained model. The code looks as follows:

```
private static void BuildNBModel(double[][] trainInput, int[] trainOutput, double[][]
testInput, int[] testOutput)
{
    var teacher = new NaiveBayesLearning<NormalDistribution>();
    var nbModel = teacher.Learn(trainInput, trainOutput);

    int[] inSamplePreds = nbModel.Decide(trainInput);
    int[] outSamplePreds = nbModel.Decide(testInput);

    // Accuracy
    double inSampleAccuracy = 1 - new ZeroOneLoss(trainOutput).Loss(inSamplePreds);
    double outSampleAccuracy = 1 - new ZeroOneLoss(testOutput).Loss(outSamplePreds);
    Console.WriteLine("* In-Sample Accuracy: {0:0.0000}", inSampleAccuracy);
    Console.WriteLine("* Out-of-Sample Accuracy: {0:0.0000}", outSampleAccuracy);

    // Build confusion matrix
    int[][] confMatrix = BuildConfusionMatrix(
        testOutput, outSamplePreds, 10
    );
    System.IO.File.WriteAllLines(
        Path.Combine(
            @"<path-to-dir>",
            "nb-conf-matrix.csv"
        ),
        confMatrix.Select(x => String.Join(",", x))
    );

    // Precision Recall
    PrintPrecisionRecall(confMatrix);
    DrawROCCurve(testOutput, outSamplePreds, 10, "NB");
}
```

As you might recall from the previous chapter, we are using the `NaiveBayesLearning` class to train a Naive Bayes classifier. We are using `NormalDistribution`, as all the features for our ML models are the principal components from the previous PCA step, and the values of these components are continuous values.

Neural network classifier

The last learning algorithm that we are going to experiment with is the ANN. As you might know already, the neural network model is the backbone of all of the deep learning technologies. The neural network model is known to perform well for image datasets, so we will compare the performance of this model against the other models to see how much performance gain we get by using the neural network over the other classification models. In order to build neural network models in C# using the Accord.NET framework, you will need to install the `Accord.Neuro` package first. You can install the `Accord.Neuro` package by using the following command in the **NuGet Package Manager Console**:

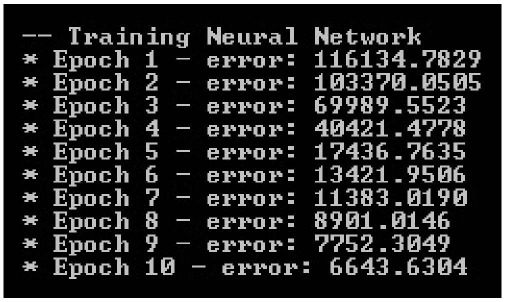
```
Install-Package Accord.Neuro
```

Let's now take a look at how we can build a neural network model in C#, using the Accord.NET framework. The code looks like the following:

```
private static void BuildNNModel(double[][] trainInput, int[] trainOutput, double[][]
testInput, int[] testOutput)
{
    double[][] outputs = Accord.Math.Jagged.OneHot(trainOutput);

    var function = new BipolarSigmoidFunction(2);
    var network = new ActivationNetwork(
        new BipolarSigmoidFunction(2),
        91,
        20,
        10
    );

    var teacher = new LevenbergMarquardtLearning(network);

    Console.WriteLine("\n-- Training Neural Network");
    int numEpoch = 10;
    double error = Double.PositiveInfinity;
    for (int i = 0; i < numEpoch; i++)
    {
        error = teacher.RunEpoch(trainInput, outputs);
        Console.WriteLine("* Epoch {0} - error: {1:0.0000}", i + 1, error);
    }
    Console.WriteLine("");

    List<int> inSamplePredsList = new List<int>();
    for (int i = 0; i < trainInput.Length; i++)
    {
        double[] output = network.Compute(trainInput[i]);
        int pred = output.ToList().IndexOf(output.Max());
        inSamplePredsList.Add(pred);
    }
    List<int> outSamplePredsList = new List<int>();
    for (int i = 0; i < testInput.Length; i++)
    {
        double[] output = network.Compute(testInput[i]);
        int pred = output.ToList().IndexOf(output.Max());
        outSamplePredsList.Add(pred);
    }
}
```

Let's take a closer look at this code. We first transform the training labels from a one-dimensional array into a two-dimensional array, where the columns are the target classes and the values are 1 if the given record belongs to the given target class, and 0 if it does not. We are using the `Accord.Math.Jagged.OneHot` method to perform one-hot encoding for the training labels. Then, we build a neural network by using the `ActivationNetwork` class. The `ActivationNetwork` class takes three parameters: the activation function, the input count, and the information about the layers. For the activation function, we are using a sigmoid function, `BipolarSigmoidFunction`. The input count is straightforward, as it is the number of features that we are going to use to train this model, which is 91. For this model, we only used one hidden layer with 20 neurons. For a deeper neural network, you can use more than one hidden layer and can also experiment with different numbers of neurons in each hidden layer. Lastly, the last parameter of the `ActivationNetwork` constructor represents the output count. Since the target variable is the digit class, it can take values between 0 and 9, and thus the number of output neurons we need is 10. Once this network is built, we can use the `LevenbergMarquardtLearning` learning algorithm to train the network.

Once we have set up the network and the learning algorithm, we can actually start training a neural network model. As you might know already, a neural network model needs to be run through the dataset multiple times (epochs) during its learning phase for better predictability. You can use the `RunEpoch` method to train and update the neural network model in each epoch. To save some time, we are only running 10 epochs to train our neural network model. However, we recommend you try increasing this value, as it can improve the performance of your neural network model. The following shows how the error measure decreases as we train and update the neural network model in each epoch:



```
-- Training Neural Network
* Epoch 1 - error: 116134.7829
* Epoch 2 - error: 103370.0505
* Epoch 3 - error: 69989.5523
* Epoch 4 - error: 40421.4778
* Epoch 5 - error: 17436.7635
* Epoch 6 - error: 13421.9506
* Epoch 7 - error: 11383.0190
* Epoch 8 - error: 8901.0146
* Epoch 9 - error: 7752.3049
* Epoch 10 - error: 6643.6304
```

As you can see from this output, the error measure decreases significantly in each epoch. One thing to note here is that the amount of reduction in the error measure decreases in each additional epoch. When you are building a neural network model with large numbers of epochs, you can monitor the amount of gain in each run and decide to stop when there is no more significant performance gain.

The full code that we used for the model building step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.8/Modeling.cs`.

Evaluating multi-class classification models

In this section, we are going to evaluate the three models that we built in the previous section. We are going to revisit the validation metrics that we used previously for the classification models, and compare the performances of each model against the others.

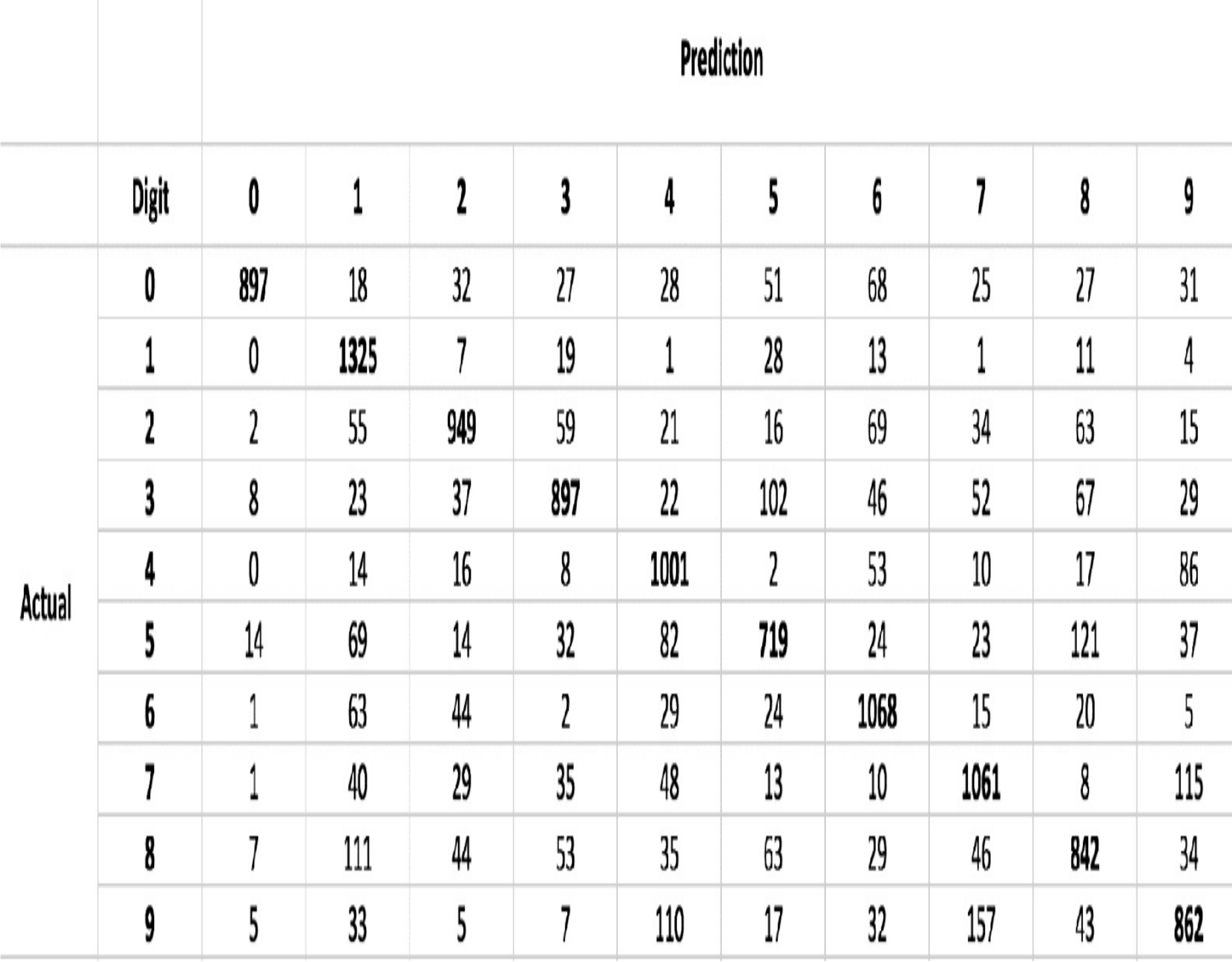
Confusion matrices

First, let's look at confusion matrices. The following code shows how you can build a confusion matrix with the predicted output and the actual output:

```
private static int[][] BuildConfusionMatrix(int[] actual, int[] preds, int numClass)
{
int[][] matrix = new int[numClass][];
for (int i = 0; i < numClass; i++)
{
matrix[i] = new int[numClass];
}

for (int i = 0; i < actual.Length; i++)
{
matrix[actual[i]][preds[i]] += 1;
}

return matrix;
}
```

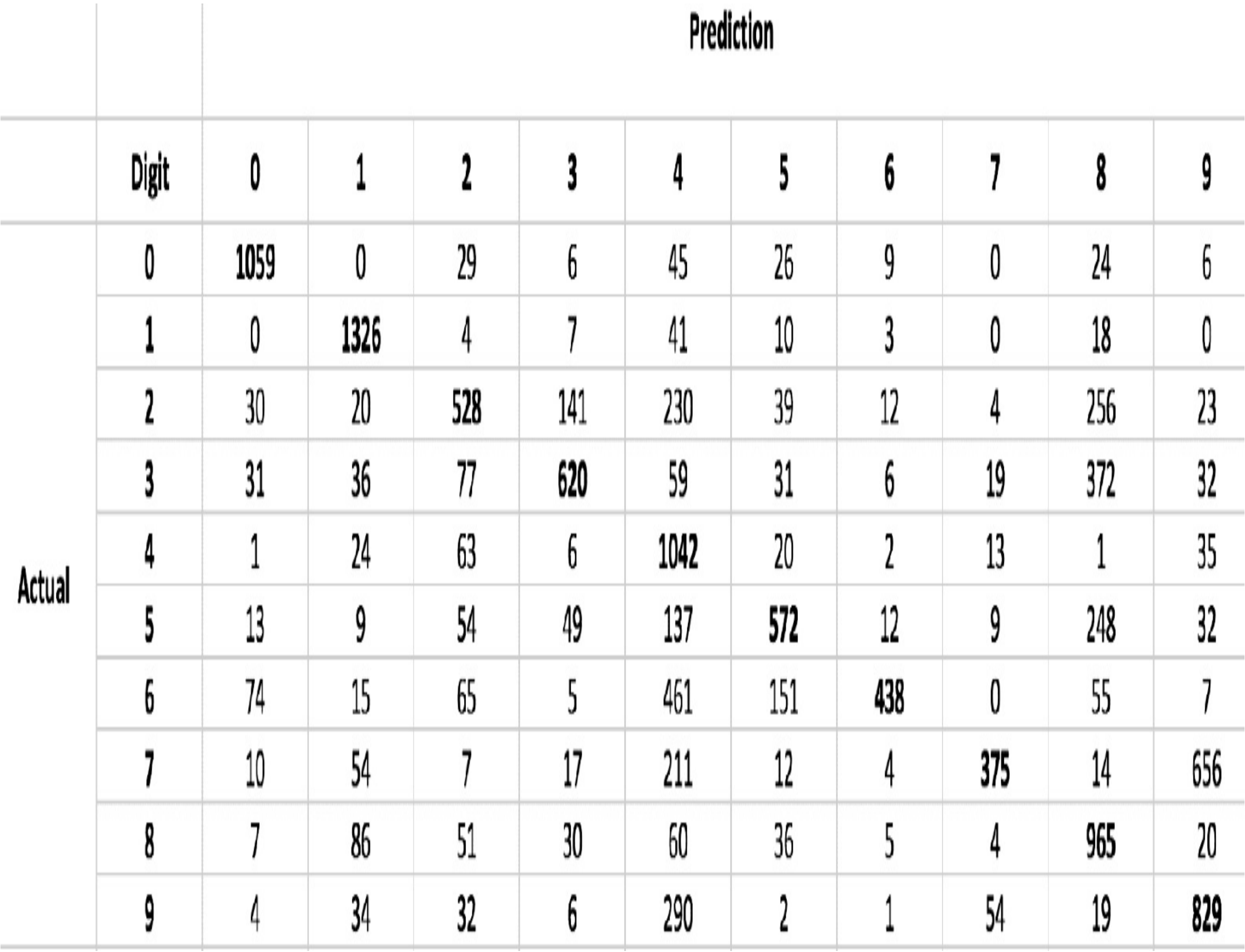
This method is similar to the one we wrote in the previous chapter, except that it is returning a two-dimensional array, instead of a string array. We are going to use this two-dimensional array output for calculating precision and recall rates in the next section.

The confusion matrix for the logistic regression classifier looks like the following:



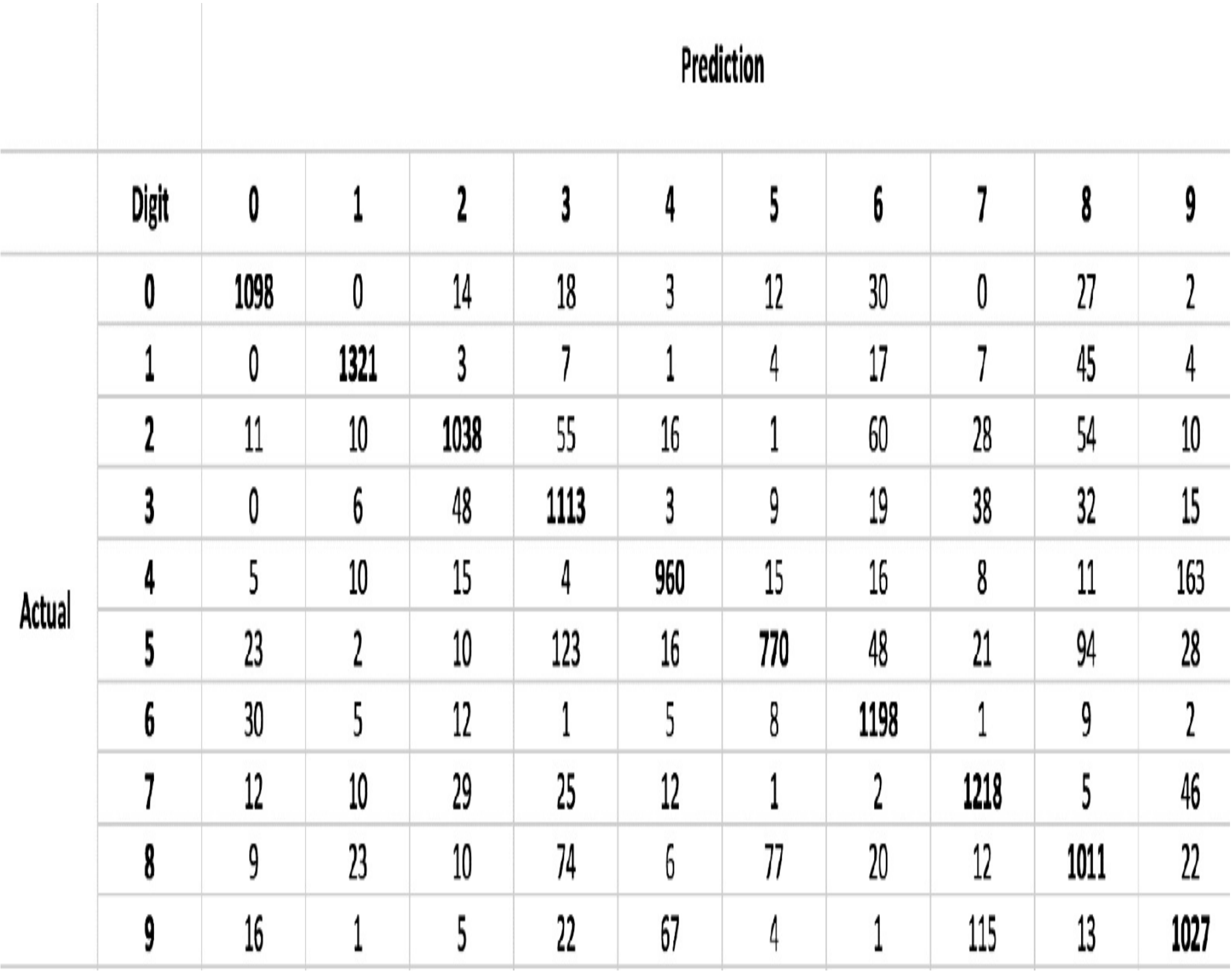
		Prediction									
	Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Actual	**0**	**897**	18	32	27	28	51	68	25	27	31
	1	0	**1325**	7	19	1	28	13	1	11	4
	2	2	55	**949**	59	21	16	69	34	63	15
	3	8	23	37	**897**	22	102	46	52	67	29
	4	0	14	16	8	**1001**	2	53	10	17	86
	5	14	69	14	32	82	**719**	24	23	121	37
	6	1	63	44	2	29	24	**1068**	15	20	5
	7	1	40	29	35	48	13	10	**1061**	8	115
	8	7	111	44	53	35	63	29	46	**842**	34
	9	5	33	5	7	110	17	32	157	43	**862**

For the Naive Bayes classifier, you will get a confusion matrix that looks similar to the following table:



		Prediction									
	Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Actual	**0**	**1059**	0	29	6	45	26	9	0	24	6
	1	0	**1326**	4	7	41	10	3	0	18	0
	2	30	20	**528**	141	230	39	12	4	256	23
	3	31	36	77	**620**	59	31	6	19	372	32
	4	1	24	63	6	**1042**	20	2	13	1	35
	5	13	9	54	49	137	**572**	12	9	248	32
	6	74	15	65	5	461	151	**438**	0	55	7
	7	10	54	7	17	211	12	4	**375**	14	656
	8	7	86	51	30	60	36	5	4	**965**	20
	9	4	34	32	6	290	2	1	54	19	**829**

Lastly, for the neural network model, the confusion matrix looks like the following:



		Prediction									
	Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Actual	**0**	**1098**	0	14	18	3	12	30	0	27	2
	1	0	**1321**	3	7	1	4	17	7	45	4
	2	11	10	**1038**	55	16	1	60	28	54	10
	3	0	6	48	**1113**	3	9	19	38	32	15
	4	5	10	15	4	**960**	15	16	8	11	163
	5	23	2	10	123	16	**770**	48	21	94	28
	6	30	5	12	1	5	8	**1198**	1	9	2
	7	12	10	29	25	12	1	2	**1218**	5	46
	8	9	23	10	74	6	77	20	12	**1011**	22
	9	16	1	5	22	67	4	1	115	13	**1027**

From these confusion matrices, the neural network model outperforms the other two models, and the logistic regression model seems to come in second.

Accuracy and precision/recall

The second metric that we are going to look at is the accuracy measure. We are use `ZeroOneLoss` to compute the loss, and then subtract it from `1` to get the accuracy number. The code to compute the accuracy measure is as follows:

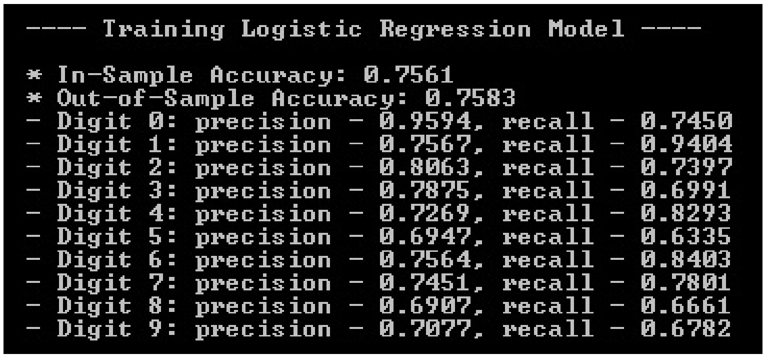
```
// Accuracy
double inSampleAccuracy = 1 - new ZeroOneLoss(trainOutput).Loss(inSamplePreds);
double outSampleAccuracy = 1 - new ZeroOneLoss(testOutput).Loss(outSamplePreds);
Console.WriteLine("* In-Sample Accuracy: {0:0.0000}", inSampleAccuracy);
Console.WriteLine("* Out-of-Sample Accuracy: {0:0.0000}", outSampleAccuracy);
```

The third and fourth metrics that we are going to look at are the precision and recall rates. Unlike before, we have 10 classes for the target prediction. So, we are going to have to calculate precision and recall rates separately for each of the target classes. The code looks like the following:

```
private static void PrintPrecisionRecall(int[][] confMatrix)
{
    for (int i = 0; i < confMatrix.Length; i++)
    {
        int totalActual = confMatrix[i].Sum();
        int correctPredCount = confMatrix[i][i];

        int totalPred = 0;
        for(int j = 0; j < confMatrix.Length; j++)
        {
            totalPred += confMatrix[j][i];
        }

        double precision = correctPredCount / (float)totalPred;
        double recall = correctPredCount / (float)totalActual;

        Console.WriteLine("- Digit {0}: precision - {1:0.0000}, recall - {2:0.0000}",
i, precision, recall);
    }

}
```

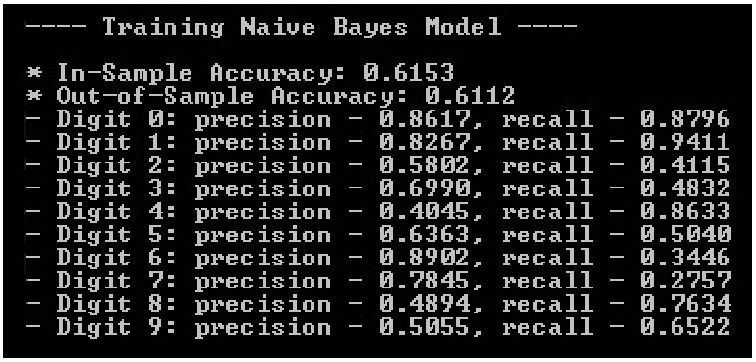
As you can see from this code, the input to this `PrintPrecisionRecall` method is the confusion matrix that we built from the previous section. In this method, it iterates through each of the target classes and computes the precision and recall rates.

The following is the output that we get when we compute accuracy, precision, and recall for the logistic regression model:



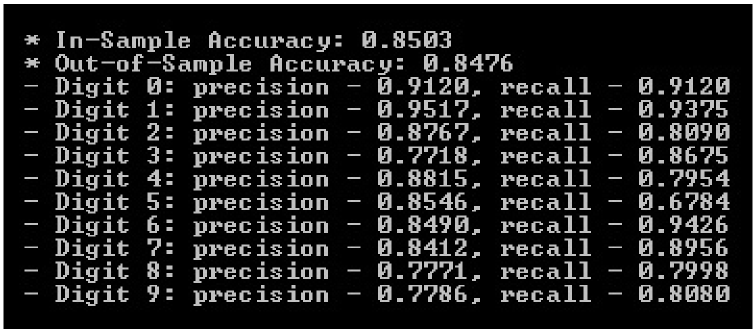
```
---- Training Logistic Regression Model ----

* In-Sample Accuracy: 0.7561
* Out-of-Sample Accuracy: 0.7583
- Digit 0: precision - 0.9594, recall - 0.7450
- Digit 1: precision - 0.7567, recall - 0.9404
- Digit 2: precision - 0.8063, recall - 0.7397
- Digit 3: precision - 0.7875, recall - 0.6991
- Digit 4: precision - 0.7269, recall - 0.8293
- Digit 5: precision - 0.6947, recall - 0.6335
- Digit 6: precision - 0.7564, recall - 0.8403
- Digit 7: precision - 0.7451, recall - 0.7801
- Digit 8: precision - 0.6907, recall - 0.6661
- Digit 9: precision - 0.7077, recall - 0.6782
```

For the Naive Bayes model, we get the following results for the metrics:



```
---- Training Naive Bayes Model ----

* In-Sample Accuracy: 0.6153
* Out-of-Sample Accuracy: 0.6112
- Digit 0: precision - 0.8617, recall - 0.8796
- Digit 1: precision - 0.8267, recall - 0.9411
- Digit 2: precision - 0.5802, recall - 0.4115
- Digit 3: precision - 0.6990, recall - 0.4832
- Digit 4: precision - 0.4045, recall - 0.8633
- Digit 5: precision - 0.6363, recall - 0.5040
- Digit 6: precision - 0.8902, recall - 0.3446
- Digit 7: precision - 0.7845, recall - 0.2757
- Digit 8: precision - 0.4894, recall - 0.7634
- Digit 9: precision - 0.5055, recall - 0.6522
```

Lastly, for the neural network model, the performance results look as follows:



```
* In-Sample Accuracy: 0.8503
* Out-of-Sample Accuracy: 0.8476
- Digit 0: precision - 0.9120, recall - 0.9120
- Digit 1: precision - 0.9517, recall - 0.9375
- Digit 2: precision - 0.8767, recall - 0.8090
- Digit 3: precision - 0.7718, recall - 0.8675
- Digit 4: precision - 0.8815, recall - 0.7954
- Digit 5: precision - 0.8546, recall - 0.6784
- Digit 6: precision - 0.8490, recall - 0.9426
- Digit 7: precision - 0.8412, recall - 0.8956
- Digit 8: precision - 0.7771, recall - 0.7998
- Digit 9: precision - 0.7786, recall - 0.8080
```

As you might notice from these results, the neural network model outperformed the other two models. Both the overall accuracy and the precision/recall rates are the highest for the neural network model, when compared to the logistic regression and Naive Bayes models. The logistic regression model seems to come in as the second best model among the three that we built.

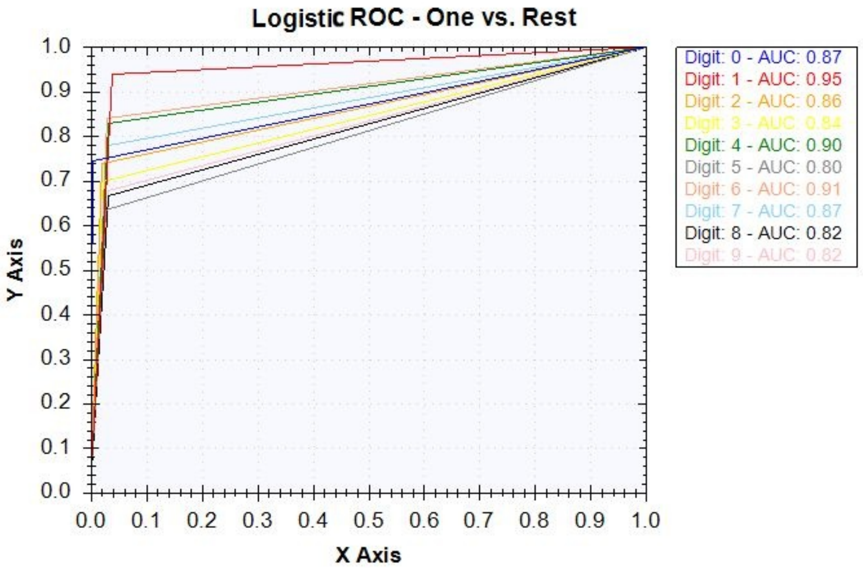
One versus Rest AUC

The last evaluation measure that we are going to look at is the **Receiver Operating Characteristic** (**ROC**) curve and the AUC. One thing we need to do differently in this chapter, when we are building a ROC curve and an AUC, is that we need to build one for each of the target classes. Let's take a look at the code first:

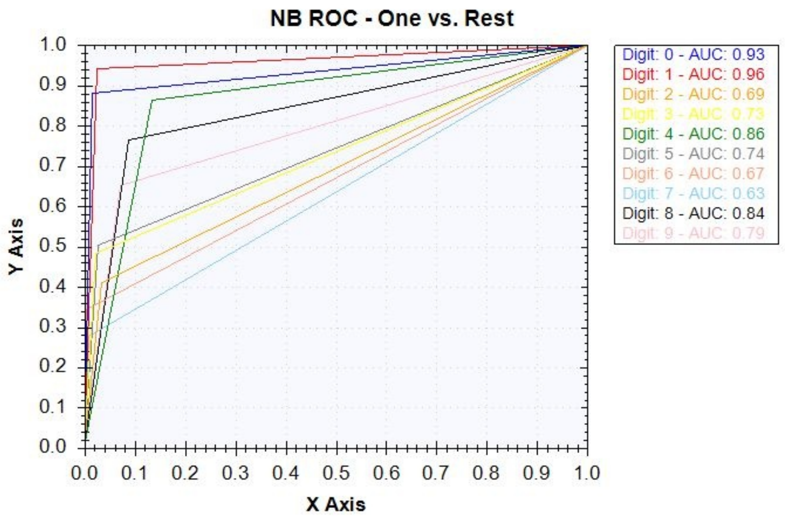
```
private static void DrawROCCurve(int[] actual, int[] preds, int numClass, string
modelName)
{
    ScatterplotView spv = new ScatterplotView();
    spv.Dock = DockStyle.Fill;
    spv.LinesVisible = true;

    Color[] colors = new Color[] {
        Color.Blue, Color.Red, Color.Orange, Color.Yellow, Color.Green,
        Color.Gray, Color.LightSalmon, Color.LightSkyBlue, Color.Black, Color.Pink
    };

    for (int i = 0; i < numClass; i++)
    {
        // Build ROC for Train Set
        bool[] expected = actual.Select(x => x == i ? true : false).ToArray();
        int[] predicted = preds.Select(x => x == i ? 1 : 0).ToArray();

        var trainRoc = new ReceiverOperatingCharacteristic(expected, predicted);
        trainRoc.Compute(1000);

        // Get Train AUC
        double auc = trainRoc.Area;
        double[] xVals = trainRoc.Points.Select(x => 1 - x.Specificity).ToArray();
        double[] yVals = trainRoc.Points.Select(x => x.Sensitivity).ToArray();

        // Draw ROC Curve
        spv.Graph.GraphPane.AddCurve(
            String.Format(
                "Digit: {0} - AUC: {1:0.00}",
                i, auc
            ),
            xVals, yVals, colors[i], SymbolType.None
        );
        spv.Graph.GraphPane.AxisChange();
    }

    spv.Graph.GraphPane.Title.Text = String.Format(
        "{0} ROC - One vs. Rest",
        modelName
    );

    Form f1 = new Form();
    f1.Width = 700;
    f1.Height = 500;
    f1.Controls.Add(spv);
    f1.ShowDialog();
|}
```

As you can see from this `DrawROCCurve` method that we wrote, we iterate through each target class in a `for` loop, and reformat the predicted and actual labels by encoding `1` if each label matches with the target class, and 0 if it does not. After we have done this encoding, we can then use the `ReceiverOperatingCharacteristic` class to compute the AUC and build the ROC curve.

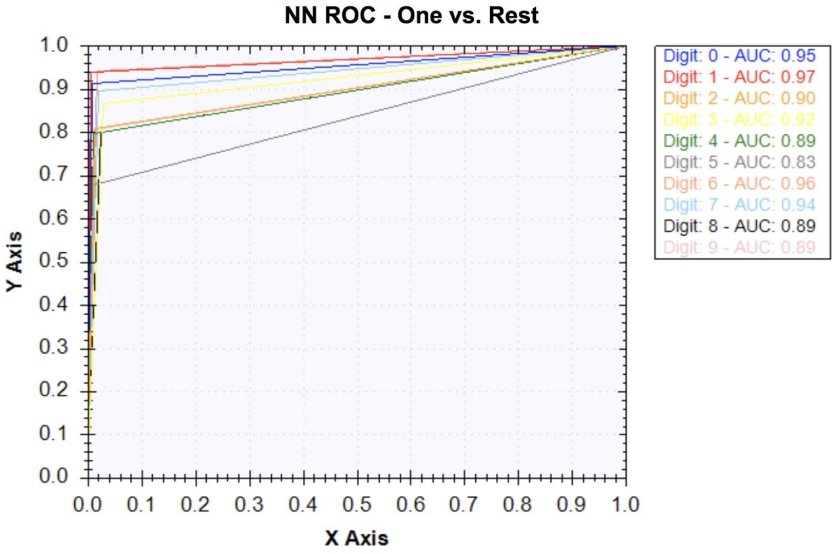
The following is the ROC curve for the logistic regression model:



For the Naive Bayes model, the ROC curve looks as follows:



Lastly, the ROC curve for the neural network model looks like the following:



As expected from the previous metrics that we have looked at, the results look the best for the neural network model, and the logistic regression model comes in as second-best. For the Naive Bayes model, there are some digits that it didn't compute well. For example, the Naive Bayes model struggles to classify the digits 6 and 7 well. However, the AUC numbers for all of the target classes are close to 1 for the neural network, which suggests that the model is trained well to identify digits for handwritten images.

From looking at the confusion matrix, the accuracy, precision and recall rates, and the ROC curves, we can conclude that the neural network model works the best among the three classifiers that we trained in this chapter. This reaffirms the fact that neural networks work well on image datasets and image recognition problems.

Summary

In this chapter, we built our first image recognition model that can identify handwritten digits in grayscale images. We started this chapter by discussing how this type of model can be widely used in real-life applications, and how we are planning to build a handwritten digit recognition model. Then, we started looking into the dataset. We first looked at the distributions of target classes to see if the sample set is a well-balanced set. When we were analyzing the pixel data, we noticed that the majority of the pixel values were 0, and we could intuitively make sense of it by reconstructing the images from the pixel data. During the feature engineering step, we discussed how we can use PCA for dimensionality reduction.

With these PCA-transformed features, we then started building various machine learning models. On top of the logistic regression and Naive Bayes models that we are already familiar with, we introduced a new ML model, neural network. We learned how to initialize the `ActivationNetwork` model with `BipolarSigmoidFunction` as an activation function. We then started training the neural network with the `LevenbergMarquardtLearning` learning algorithm over 10 epochs. We saw how error measures decrease in each additional epoch, and discussed how the amount of gain in the error rate is in diminishing returns for additional epochs. In the model evaluation step, we combined multiple validation metrics for classification models. For the machine learning models we built in this chapter, we looked at the confusion matrix, prediction accuracy, precision and recall rates, and the ROC curves and AUC. We noticed how the neural network model outperformed the other two models, which reaffirmed that neural network models work well for image data.

In the next chapter, we are going to switch gears and start building models for anomaly detection. We are going to work on a cyber attack detection project using PCA. With the Network Intrusion dataset, we will discuss how to use PCA to detect cyber attacks, and run multiple experiments to find the optimal threshold at which to notify us about potential cyber attacks.

Cyber Attack Detection

So far, we have been mainly developing **machine learning** (**ML**) models with well-balanced sample sets, where the target classes are distributed equally or almost equally across the sample records in the dataset. However, there are cases where a dataset has imbalanced class distributions. Class imbalance is especially common in anomaly and fraud detections. These kinds of class imbalance problems causes issues when training ML models, as most ML algorithms work best when the target classes are roughly equally distributed. In order to tackle this imbalanced class problem, we cannot approach it the same way we have been developing models for various classification and regression problems. We will need to approach it differently.

In this chapter, we are going to discuss how we can build an anomaly detection model. We will be using a network intrusion dataset, **KDD Cup 1999 Data**, which has a large amount of network connection data where some of the connections are normal network connections, and some others are cyber attacks. We will first look at the structure of the data, types of cyber attacks present in the dataset, and distributions of various network features. Then, we will apply some of the feature-engineering techniques we have discussed in previous chapters, as the feature set contains both categorical and continuous variables. We are also going to apply the dimensionality reduction technique, **Principal Component Analysis** (**PCA**), that we discussed in the previous chapter. In addition to what we covered about PCA in the previous chapter, we are going to use PCA to build models for anomaly detection. With the models built using PCA, we are going to further discuss some of the ways to evaluate anomaly detection models, and what will work best for the cyber attack detection project.

In this chapter, we will cover the following topics:

- Problem definition for the cyber attack detection project
- Data analysis for the internet traffic dataset
- Feature engineering and PCA
- Principal component classifier for anomaly detection

- Evaluating anomaly detection models

Problem definition

Datasets with imbalanced class distributions cause problems for most ML algorithms, as they typically perform well for well-balanced datasets. There are various ways to handle class imbalance problems in ML. Resampling the dataset to balance the target classes is one way. You can upsample the positive training samples, where you randomly select and duplicate the positive training samples, so that roughly 50% of the dataset belongs to a positive class. You can also downsample the negative training samples so that the number of negative training examples matches with the number of positive training examples. In cases of extreme class imbalance, you can approach it as an anomaly detection problem, where the positive events are considered anomalies or outliers. Anomaly detection techniques have many applications in real-world problems. They are often used for network intrusion detection, credit card fraud detection, or even medical diagnosis.

In this chapter, we are going to work on building an anomaly detection model for cyber attacks. In order to build a cyber attack detection model, we are going to use the **KDD Cup 1999 Data**, which has a large amount of artificial and hand-injected cyber attack data, along with normal network connection data. This data can be found at the following link: `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`. With this data, we are going to first look at the distributions of the cyber attack types and then the distributions of the network features. Since this is a simulated and artificial dataset, the majority of this dataset is made up of cyber attacks, which are abnormal and unrealistic in the real world. In order to simulate real-world examples of cyber attacks we are going to randomly sub-select the cyber attack events from the sample set and build a new training set that contains more normal network connections than malicious connections. With this sub-sampled dataset, we are going to build an anomaly detection model using PCA. Then, we are going to evaluate this model by looking at the cyber attack detection rate at various target false alarm rates.

To summarize our problem definition for the cyber attack detection project:

- What is the problem? We need a cyber attack detection model that can identify potential malicious connections from large amounts of network connections so that we can avoid cyber attacks.
- Why is it a problem? The number of cyber attacks increases every year and without being properly prepared for such attacks, our systems will become more vulnerable from various cyber attacks. With a cyber attack detection model, we can avoid becoming the victims of cyber attacks.
- What are some of the approaches to solving this problem? We are going to use publicly available data that has a large amount of artificial and simulated cyber attack data. We are going to sub-sample this data to replicate a real-life situation where there are more normal network connections than abnormal and malicious connections. Then, we are going to use PCA and its principal components to detect anomalies.
- What are the success criteria? We want a high cyber attack detection rate, even if we need to sacrifice it for higher false alarm rate. This is because we are more concerned about allowing cyber attacks than false positive alerts.

Data analysis for internet traffic data

Let's start by taking a look into the internet traffic data. As mentioned previously, we are going to use the KDD Cup 1999 Data, which you can download from the following link: `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`. We will be using the `kddcup.data_10_percent.gz` data for this cyber attack detection project.

Data clean-up

The first thing we need to do is clean up the data for future steps. If you open the data that you just downloaded, you will notice that there is no header in the dataset. However, for future data analysis and model building, it is always beneficial to have headers associated with each column. Based on the column description that can be found at `http://kdd.ics.uci.edu/databases/kddcup99/kddcup.names`, we are going to attach headers to the raw dataset. The code to attach column names to the data frame looks as follows:

```
// Read in the Cyber Attack dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-data-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "kddcup.data_10_percent");
Console.WriteLine("Loading {0}\n\n", dataPath);
var featuresDF = Frame.ReadCsv(
    dataPath,
    hasHeaders: false,
    inferTypes: true
);

string[] colnames =
{
    "duration", "protocol_type", "service", "flag", "src_bytes",
    "dst_bytes", "land", "wrong_fragment", "urgent", "hot",
    "num_failed_logins", "logged_in", "num_compromised", "root_shell",
    "su_attempted", "num_root", "num_file_creations", "num_shells",
    "num_access_files", "num_outbound_cmds", "is_host_login", "is_guest_login",
    "count", "srv_count", "serror_rate", "srv_serror_rate", "rerror_rate",
    "srv_rerror_rate", "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
    "dst_host_count", "dst_host_srv_count", "dst_host_same_srv_rate",
    "dst_host_diff_srv_rate", "dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate", "dst_host_serror_rate",
    "dst_host_srv_serror_rate", "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
    "attack_type"
};
featuresDF.RenameColumns(colnames);
```

As you can see from this code, we are loading this raw dataset without headers, by supplying the `hasHeaders: false` flag to the `ReadCsv` method of Deedle's data frame. By supplying this flag, we are telling Deedle not to take the first row of the dataset as the header. Once this data is loaded into a data frame, we are using the `RenameColumns` method to attach the names of the columns to the data frame.

The next clean-up task we are going to take is to group the cyber attack types together by corresponding categories. You can find the mapping between the attack type and the category at the following link: http://kdd.ics.uci.edu/databases/kddcup99/training_attack_types. Using this mapping, we are going to create a new column in the data frame that contains information about the attack category. Let's look at the code first:

```
// keeping "normal" for now for plotting purposes
IDictionary<string, string> attackCategories = new Dictionary<string, string>
{
    {"back", "dos"},
    {"land", "dos"},
    {"neptune", "dos"},
    {"pod", "dos"},
    {"smurf", "dos"},
    {"teardrop", "dos"},
    {"ipsweep", "probe"},
    {"nmap", "probe"},
    {"portsweep", "probe"},
    {"satan", "probe"},
    {"ftp_write", "r2l"},
    {"guess_passwd", "r2l"},
    {"imap", "r2l"},
    {"multihop", "r2l"},
    {"phf", "r2l"},
    {"spy", "r2l"},
    {"warezclient", "r2l"},
    {"warezmaster", "r2l"},
    {"buffer_overflow", "u2r"},
    {"loadmodule", "u2r"},
    {"perl", "u2r"},
    {"rootkit", "u2r"},
    {"normal", "normal"}
};

featuresDF.AddColumn(
    "attack_category",
    featuresDF.GetColumn<string>("attack_type")
        .Select(x => attackCategories[x.Value.Replace(".", "")])
);
```

If you look closely at this code, we created a `Dictionary` object that has mapping between an attack type and its category. For example, the attack type, `"back"`, is one of the **Denial-of-Service** (**DOS**) attacks and the attack type, `"rootkit"`, is one of the **User-to-Root** (**U2R**) attacks. Using this mapping, we created a new column, `"attack_category"`, and added it to the `featuresDF`. Now that we have cleaned the raw dataset with column names and attack categories, we need to export it and store it into our local drive for future use. You can use the following code to export this data:

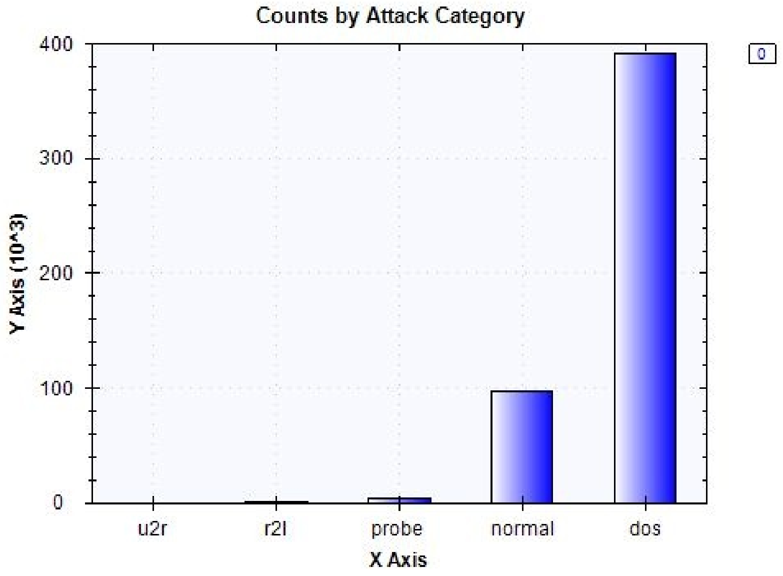
```
featuresDF.SaveCsv(Path.Combine(dataDirPath, "data.csv"));
```

Target variable distribution

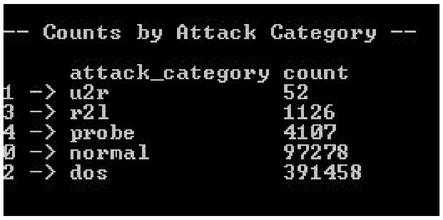
Now that we have clean data to work with, we will start digging into the data. Let's first look at the distributions of cyber attack categories. The code to get the distribution of target variables looks as follows:

```
// 1. Target Variable Distribution
Console.WriteLine("\n\n-- Counts by Attack Category --\n");
var attackCount = featuresDF.AggregateRowsBy<string, int>(
    new string[] { "attack_category" },
    new string[] { "duration" },
    x => x.ValueCount
).SortRows("duration");
attackCount.RenameColumns(new string[] { "attack_category", "count" });

attackCount.Print();

DataBarBox.Show(
    attackCount.GetColumn<string>("attack_category").Values.ToArray(),
    attackCount["count"].Values.ToArray()
).SetTitle(
    "Counts by Attack Category"
);
```

Similar to the previous chapters, we are using the `AggregateRowsBy` method in Deedle's data frame to group by the target variable, `attack_category`, and count the number of occurrences per category in the dataset. Then, we use the `DataBarBox` class to display a bar chart of this distribution. Once you run this code, the following bar chart will be displayed:



And the output that shows us the number of occurrences of each cyber attack category looks as follows:



```
-- Counts by Attack Category --

     attack_category count
1 -> u2r             52
3 -> r2l             1126
4 -> probe           4107
0 -> normal          97278
2 -> dos             391458
```

There is one thing that is noticeable here. There are more DOS attack samples than normal samples in the dataset. As mentioned previously, the KDD Cup 1999 dataset that we are using for this project is artificial and simulated data, and thus, it does not reflect a real-life situation, where the number of normal internet connections will outnumber the number of all the other cyber attacks combined. We will have to keep this in mind when building models in the following sections.

Categorical variable distribution

The features that we have in this dataset are a mixture of categorical and continuous variables. For example, the feature named `duration`, which represents the length of the connection, is a continuous variable. However, the feature, named `protocol_type`, which represents the type of the protocol, such as `tcp`, `udp`, and so forth, is a categorical variable. For a complete set of feature descriptions, you can go to this link: `http://kdd.ics.uci.edu/databases/kddcup99/task.html`.

In this section, we are going to take a look at the distribution differences in the categorical variables between the normal connections and the malicious connections. The following code shows how we separate the sample set into two subgroups, one for normal connections and another for abnormal connections:

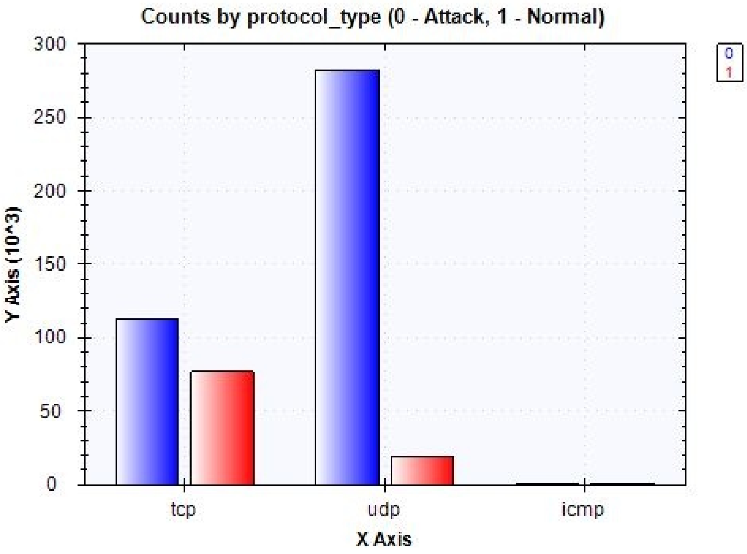
```
var attackSubset = featuresDF.Rows[
    featuresDF.GetColumn<string>("attack_category").Where(
        x => !x.Value.Equals("normal")
    ).Keys
];
var normalSubset = featuresDF.Rows[
    featuresDF.GetColumn<string>("attack_category").Where(
        x => x.Value.Equals("normal")
    ).Keys
];
```

Now that we have these two subsets, let's start comparing the distributions of categorical variables between normal and malicious connections. Let's first take a look at the code:

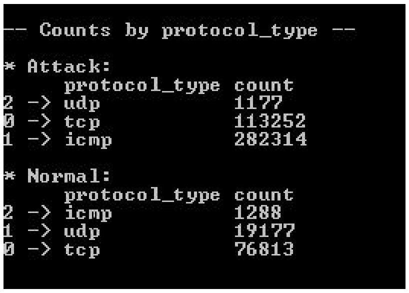
```
// 2. Categorical Variable Distribution
string[] categoricalVars =
{
    "protocol_type", "service", "flag", "land"
};
foreach (string variable in categoricalVars)
{
    Console.WriteLine("\n\n-- Counts by {0} --\n", variable);
    Console.WriteLine("* Attack:");
    var attackCountDF = attackSubset.AggregateRowsBy<string, int>(
        new string[] { variable },
        new string[] { "duration" },
        x => x.ValueCount
    );
    attackCountDF.RenameColumns(new string[] { variable, "count" });

    attackCountDF.SortRows("count").Print();

    Console.WriteLine("* Normal:");
    var countDF = normalSubset.AggregateRowsBy<string, int>(
        new string[] { variable },
        new string[] { "duration" },
        x => x.ValueCount
    );
    countDF.RenameColumns(new string[] { variable, "count" });

    countDF.SortRows("count").Print();

    DataBarBox.Show(
        countDF.GetColumn<string>(variable).Values.ToArray(),
        new double[][]
        {
            attackCountDF["count"].Values.ToArray(),
            countDF["count"].Values.ToArray()
        }
    ).SetTitle(
        String.Format("Counts by {0} (0 - Attack, 1 - Normal)", variable)
    );
}
```

In this code, we are iterating through an array of categorical variables: `protocol_type`, `service`, `flag`, and `land`. We will defer the feature descriptions to the description page that can be found at the following link: `http://kdd.ics.uci.edu/databases/kddcup99/task.html`. For each categorical variable, we used the `AggregateRowsBy` method to group by each type of the variable and count the number of occurrences for each type. We do this aggregation once for the normal group and then once more for the attack group. Then, we use the `DataBarBox` class to display bar charts to visually show the differences in the distributions. Let's take a look at a few plots and outputs.

The following bar chart is for the `protocol_type` feature:



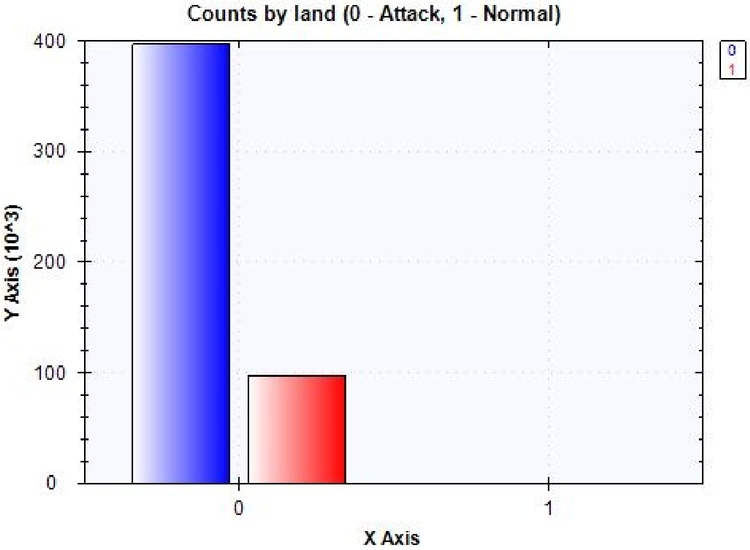
The actual counts per type between the two groups look as follows:



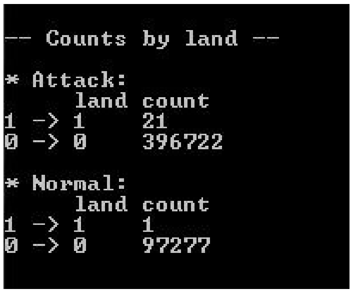
```
-- Counts by protocol_type --

* Attack:
     protocol_type count
2 -> udp           1177
0 -> tcp           113252
1 -> icmp          282314

* Normal:
     protocol_type count
2 -> icmp          1288
1 -> udp           19177
0 -> tcp           76813
```

As you can see from these outputs, there are some noticeable distinctions between the distributions of normal and cyber attack groups. For example, the majority of attacks happen on `icmp` and `tcp` protocols, while the majority of normal connections are on `tcp` and `udp`.

The following bar chart is for the `land` feature:



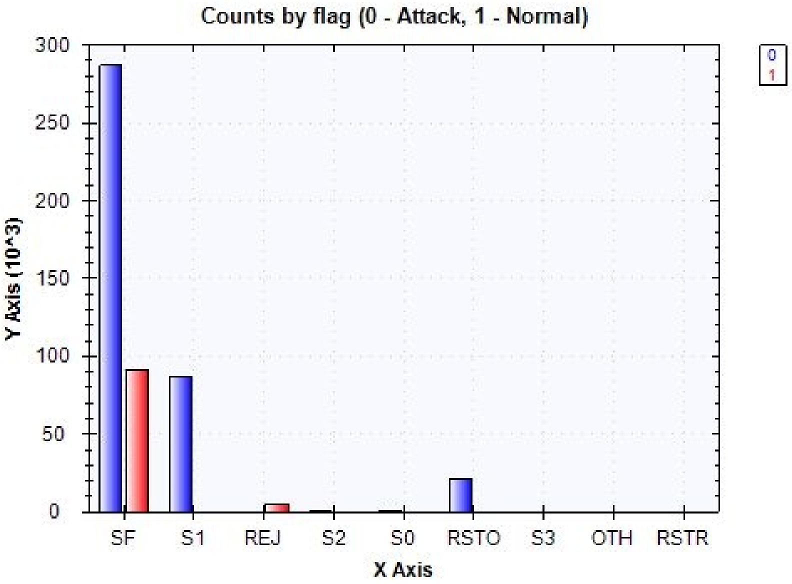
The actual counts for each type in this feature look as follows:



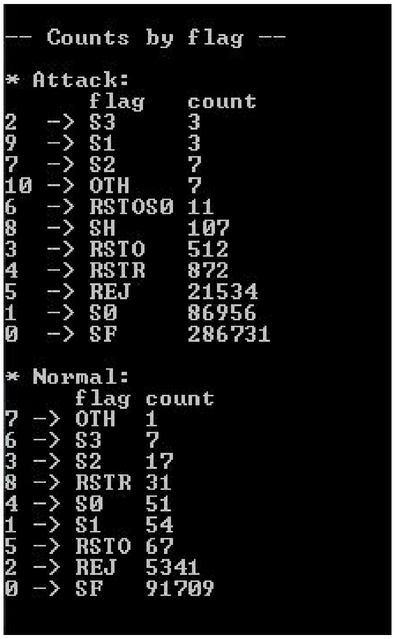
```
-- Counts by land --

* Attack:
     land count
1 -> 1    21
0 -> 0    396722

* Normal:
     land count
1 -> 1    1
0 -> 0    97277
```

It is quite hard to tell if we can deduce any meaningful insights from these outputs. Almost all samples in the dataset have a value of `0` for both the attack and normal groups. Let's take a look at one more feature.

The following bar chart shows the distributions of the feature `flag` in attack and normal groups:



And the actual counts look as follows:



```
-- Counts by flag --

* Attack:
      flag   count
2  -> S3     3
9  -> S1     3
7  -> S2     7
10 -> OTH    7
6  -> RSTOS0 11
8  -> SH     107
3  -> RSTO   512
4  -> RSTR   872
5  -> REJ    21534
1  -> S0     86956
0  -> SF     286731

* Normal:
     flag count
7 -> OTH  1
6 -> S3   7
3 -> S2   17
8 -> RSTR 31
4 -> S0   51
1 -> S1   54
5 -> RSTO 67
2 -> REJ  5341
0 -> SF   91709
```

There are some noticeable distinctions in this feature, even though the most frequently appearing flag type for both attack and normal groups is `SF`. It seems the flag types `SF` and `REJ` take up the majority of the normal group. On the other hand, the flag types `SF`, `S0`, and `REJ` take up the majority of the attack group.

Continuous variable distribution

So far, we have looked at the distributions of categorical variables. Let's now look at the distributions of continuous variables in our feature set. Similar to the previous chapters, we are going to look at the quartiles for each continuous variable. The code to compute quartiles for each continuous feature looks as follows:

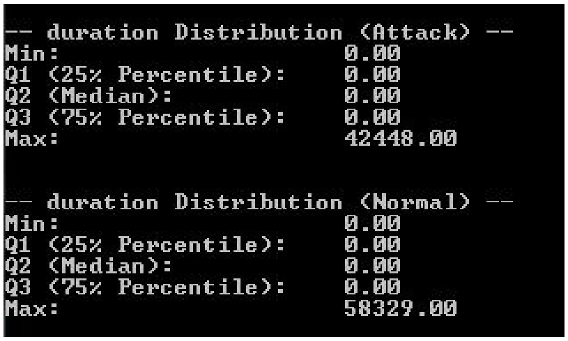
```
foreach (string variable in continuousVars)
{
    Console.WriteLine(String.Format("\n\n-- {0} Distribution (Attack) -- ", variable));
    double[] attachQuartiles = Accord.Statistics.Measures.Quantiles(
        attackSubset[variable].DropMissing().ValuesAll.ToArray(),
        new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
    );
    Console.WriteLine(
        "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
        attachQuartiles[0], attachQuartiles[1], attachQuartiles[2], attachQuartiles[3],
attachQuartiles[4]
    );

    Console.WriteLine(String.Format("\n\n-- {0} Distribution (Normal) -- ", variable));
    double[] normalQuantiles = Accord.Statistics.Measures.Quantiles(
        normalSubset[variable].DropMissing().ValuesAll.ToArray(),
        new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
    );
    Console.WriteLine(
        "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
        normalQuantiles[0], normalQuantiles[1], normalQuantiles[2], normalQuantiles[3],
normalQuantiles[4]
    );
}
```

And the variable, `continuousVars`, is defined as the following array of strings:

```
// 3. Continuous Variable Distribution
string[] continuousVars =
{
    "duration", "src_bytes", "dst_bytes", "wrong_fragment", "urgent", "hot",
    "num_failed_logins", "num_compromised", "root_shell", "su_attempted",
    "num_root", "num_file_creations", "num_shells", "num_access_files",
    "num_outbound_cmds", "count", "srv_count", "serror_rate", "srv_serror_rate",
    "rerror_rate", "srv_rerror_rate", "same_srv_rate", "diff_srv_rate",
    "srv_diff_host_rate", "dst_host_count", "dst_host_srv_count",
    "dst_host_same_srv_rate", "dst_host_diff_srv_rate", "dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate", "dst_host_serror_rate", "dst_host_srv_serror_rate",
    "dst_host_rerror_rate", "dst_host_srv_rerror_rate"
};
```

Similar to what we did for categorical variable analysis, we start iterating through the continuous variables in the preceding code. The string array, `continuousVars`, contains a list of all the continuous features we have in our dataset, and we iterate through this array to start computing the quartiles of each distribution. As in the previous chapters, we are using the `Accord.Statistics.Measures.Quantiles` method to compute quartiles, which are min, 25% percentile, median, 75% percentile, and max numbers. We do this twice, once for the attack group and another time for the normal group, so that we can see if there are any noticeable differences in the distributions. Let's take a look at a few features and their distributions.

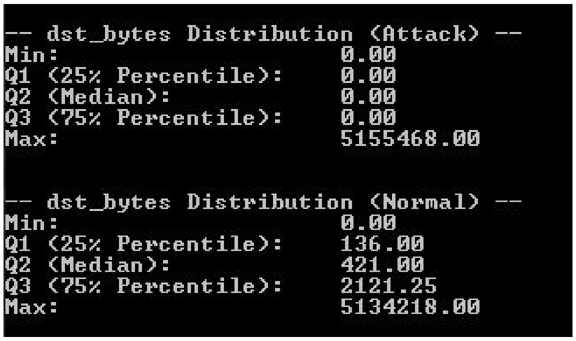
First, the following output is for the distribution of a feature called `duration`:



```
-- duration Distribution (Attack) --
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.00
Max:                        42448.00

-- duration Distribution (Normal) --
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.00
Max:                        58329.00
```

From this output, we can see that the majority of the values for this feature are `0` for both attack and normal groups. As there is not so much variance in this variable, our model might not learn much information from this feature. Let's take a look at another feature.

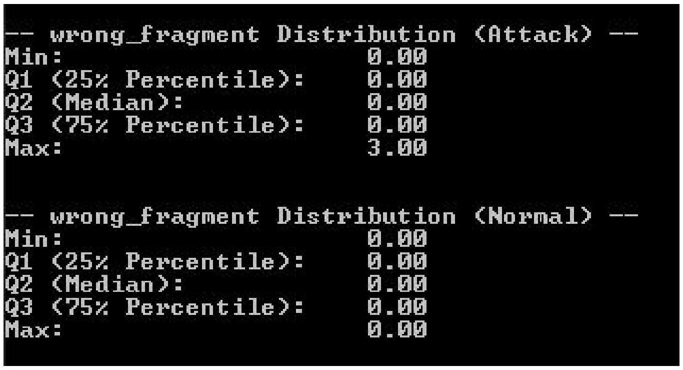
The following output is for the distribution of a feature called `dst_bytes`, which represents the number of data bytes from destination to source:



```
-- dst_bytes Distribution (Attack) --
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.00
Max:                        5155468.00

-- dst_bytes Distribution (Normal) --
Min:                        0.00
Q1 (25% Percentile):        136.00
Q2 (Median):                421.00
Q3 (75% Percentile):        2121.25
Max:                        5134218.00
```

Here, we see some noticeable distinctions in the distributions between the attack and normal groups. Almost all the cyber attacks have a value of 0, while the values are distributed across a wide range for the normal network connections.

Lastly, the following output is for a feature called `wrong_fragment`:



```
-- wrong_fragment Distribution (Attack) --
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.00
Max:                        3.00

-- wrong_fragment Distribution (Normal) --
Min:                        0.00
Q1 (25% Percentile):        0.00
Q2 (Median):                0.00
Q3 (75% Percentile):        0.00
Max:                        0.00
```

Similar to the case of the `duration` feature, the majority of the values are `0` for both the attack and normal group, which suggests that our model might not learn many insights from this feature. You can run the previous code to look at the distribution differences between the two groups for all the other features.

The full code to run this data analysis step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.9/DataAnalyzer.cs`.

Feature engineering and PCA

So far, we have analyzed the distributions of the target variable `attack_category`, as well as the categorical and continuous variables in the cyber attack dataset. In this section, we are going to focus on encoding the target variable and categorical features, and creating PCA features for our future model-building step.

Target and categorical variables encoding

First, we will have to encode different classes in the target variable, `attack_category`. If you recall from the previous data analysis step, there are five different categories: normal, `dos`, `probe`, `r2l`, and `u2r`. We are going to encode each of these string values with a corresponding integer representation. Then, we are going to encode each of the categorical variables with one-hot encoding, where we encode with 1 if the given value appears in the example, and 0 if not. Let's first load the cleaned-up data that we created in the previous data analysis step, using the following code:

```
// Read in the Cyber Attack dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-data-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "data.csv");
Console.WriteLine("Loading {0}\n\n", dataPath);
var rawDF = Frame.ReadCsv(
    dataPath,
    hasHeaders: true,
    inferTypes: true
);
```

As you can see from this code, we set `hasHeaders: true`, as the cleaned-up data now has correct headers associated with each of the columns. The following code shows how we went about encoding the target and categorical variables:

```
// Encode Categorical Variables
string[] categoricalVars =
{
    "protocol_type", "service", "flag", "land"
};
// Encode Target Variables
IDictionary<string, int> targetVarEncoding = new Dictionary<string, int>
{
    {"normal", 0},
    {"dos", 1},
    {"probe", 2},
    {"r2l", 3},
    {"u2r", 4}
};

var featuresDF = Frame.CreateEmpty<int, string>();

foreach (string col in rawDF.ColumnKeys)
```

```
{
    if(col.Equals("attack_type"))
    {
        continue;
    }
    else if (col.Equals("attack_category"))
    {
        featuresDF.AddColumn(
            col,
            rawDF.GetColumn<string>(col).Select(x => targetVarEncoding[x.Value])
        );
    }
    else if (categoricalVars.Contains(col))
    {
        var categoryDF = EncodeOneHot(rawDF.GetColumn<string>(col), col);

        foreach (string newCol in categoryDF.ColumnKeys)
        {
            featuresDF.AddColumn(newCol, categoryDF.GetColumn<int>(newCol));
        }
    }
    else
    {
        featuresDF.AddColumn(
            col,
            rawDF[col].Select((x, i) => double.IsNaN(x.Value) ? 0.0 : x.Value)
        );
    }
}
```

Let's take a deeper look at this code. We first created a string array variable, `categoricalVars`, which contains the column names of all the categorical variables, and a dictionary variable, `targetVarEncoding`, which maps each target class to an integer value. For example, we are encoding the `normal` class as `0`, the `dos` attack class as `1`, and so forth. Then, we iterate through all the columns in the `rawDF` data frame and start adding encoded data to the new and empty `featuresDF`. One thing to note here is that we use a helper function, `EncodeOneHot`, for encoding each of the categorical variables. Let's take a look at the following code:

```
private static Frame<int, string> EncodeOneHot(Series<int, string> rows, string
originalColName)
{

    var categoriesByRows = rows.GetAllValues().Select((x, i) =>
    {
        // Encode the categories appeared in each row with 1
        var sb = new SeriesBuilder<string, int>();
        sb.Add(String.Format("{0}_{1}", originalColName, x.Value), 1);

        return KeyValue.Create(i, sb.Series);
    });

    // Create a data frame from the rows we just created
    // And encode missing values with 0
    var categoriesDF = Frame.FromRows(categoriesByRows).FillMissing(0);

    return categoriesDF;
}
```

If you recall Chapter 2, *Spam Email Filtering* and Chapter 3, *Twitter Sentiment Analysis*, this code should look familiar. In this code, we iterate through each row, create a new variable that is a combination of the original column name and the value, and finally create a new Deedle data frame, `categoriesDF`. Once this step is done, this data frame output gets appended to the `featuresDF` in the previous code.

Now that we are done with encoding the target and categorical variables, we will need to export and store this new data frame, `featuresDF`. We are using the following code to store this data:

```
Console.WriteLine("* Exporting feature set...");
featuresDF.SaveCsv(Path.Combine(dataDirPath, "features.csv"));
```

Fitting PCA

With the encoded data we just created in the previous section, let's start building PCA features that we are going to use for the anomaly detection in the following model-building step.

The first thing we need to do is separate our sample set into two separate sets—one with normal connection data and another with malicious connections. While we create these subsets, we need to create more realistic distributions between the two groups. If you recall from the previous data analysis step, we noticed that there are more malicious connections than normal connections, which is unrealistic, due to the fact that the KDD CUP 1999 Dataset is an artificial and hand-injected dataset. Aside from the purpose of creating a dataset with a more realistic number of normal and malicious connections, we need to create the two subsets so that we can apply PCA to the normal group only, and then apply it to the abnormal group.

This is because we want to learn and build principal components only from the normal connections group, and be able to flag any outliers as potential cyber attacks. We will discuss more in detail about how we are going to build an anomaly detection model using principal components.

Let's take a look at the following code for splitting our sample set into two groups—one for the normal group and another for the cyber attack group:

```
// Build PCA with only normal data
var rnd = new Random();

int[] normalIdx = featuresDF["attack_category"]
    .Where(x => x.Value == 0)
    .Keys
    .OrderBy(x => rnd.Next())
    .Take(90000).ToArray();
int[] attackIdx = featuresDF["attack_category"]
    .Where(x => x.Value > 0)
    .Keys
    .OrderBy(x => rnd.Next())
    .Take(10000).ToArray();
int[] totalIdx = normalIdx.Concat(attackIdx).ToArray();
```

As you can see from this code, we are building arrays of indexes for the normal and cyber attack groups by filtering for whether the `attack_category` is `0` (normal) or greater than 0 (cyber attacks). Then, we randomly select 90,000 samples from the normal connections and 10,000 samples from the malicious connections. Now that we have the indexes for the normal and abnormal groups, we are going to use the following code to build the actual data for fitting PCA:

```
var normalSet = featuresDF.Rows[normalIdx];

string[] nonZeroValueCols = normalSet.ColumnKeys.Where(
    x => !x.Equals("attack_category") && normalSet[x].Max() != normalSet[x].Min()
).ToArray();

double[][] normalData = BuildJaggedArray(
    normalSet.Columns[nonZeroValueCols].ToArray2D<double>(),
    normalSet.RowCount,
    nonZeroValueCols.Length
);
double[][] wholeData = BuildJaggedArray(
    featuresDF.Rows[totalIdx].Columns[nonZeroValueCols].ToArray2D<double>(),
    totalIdx.Length,
    nonZeroValueCols.Length
);
int[] labels = featuresDF
    .Rows[totalIdx]
    .GetColumn<int>("attack_category")
    .ValuesAll.ToArray();
```

As you can see from this code, the `normalData` variable contains all the normal connection samples and the `wholeData` variable contains both the normal and cyber attack connection samples. We will be using `normalData` to fit PCA, and then apply this learned PCA to the `wholeData`, as you can see from the following code:

```
var pca = new PrincipalComponentAnalysis(
    PrincipalComponentMethod.Standardize
);
pca.Learn(normalData);

double[][] transformed = pca.Transform(wholeData);
```

As in `Chapter 8`, *Handwritten Digit Recognition*, we are using the `PrincipalComponentAnalysis` class in the Accord.NET framework to fit PCA. Once we have trained PCA with the normal connections data, we apply it to the `wholeData` that contains both normal and cyber attack connections by using the `Transform` method of the `pca` object.

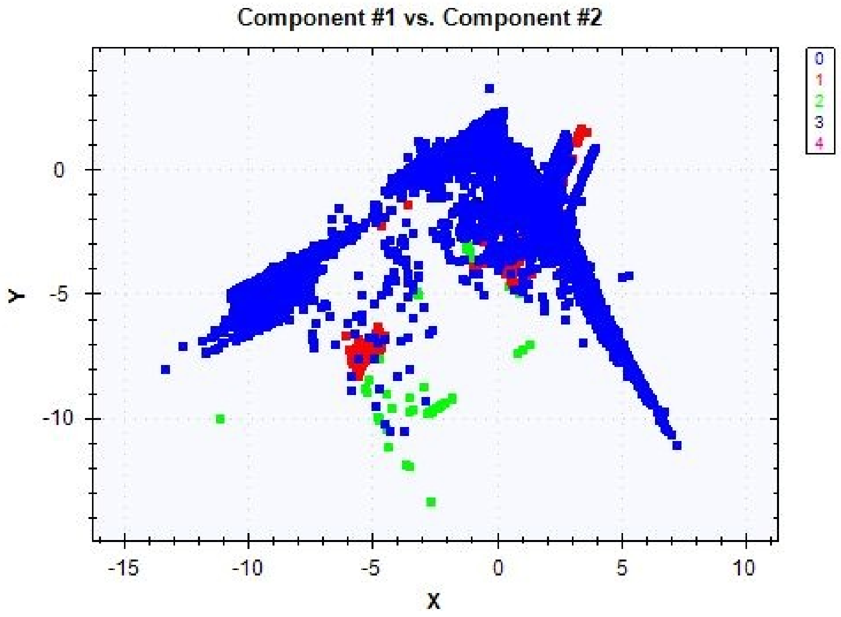
PCA features

We have now built principal components using just the normal connections group. Let's briefly inspect how well our target classes are separated on different combinations of principal components. Take a look at the following code:

```
double[][] first2Components = transformed.Select(
    x => x.Where((y, i) => i < 2).ToArray()
).ToArray();
ScatterplotBox.Show("Component #1 vs. Component #2", first2Components, labels);

double[][] next2Components = transformed.Select(
    x => x.Where((y, i) => i < 3 && i >= 1).ToArray()
).ToArray();
ScatterplotBox.Show("Component #2 vs. Component #3", next2Components, labels);

next2Components = transformed.Select(
    x => x.Where((y, i) => i < 4 && i >= 2).ToArray()
).ToArray();
ScatterplotBox.Show("Component #3 vs. Component #4", next2Components, labels);

next2Components = transformed.Select(
    x => x.Where((y, i) => i < 5 && i >= 3).ToArray()
).ToArray();
ScatterplotBox.Show("Component #4 vs. Component #5", next2Components, labels);

next2Components = transformed.Select(
    x => x.Where((y, i) => i < 6 && i >= 4).ToArray()
).ToArray();
ScatterplotBox.Show("Component #5 vs. Component #6", next2Components, labels);
```

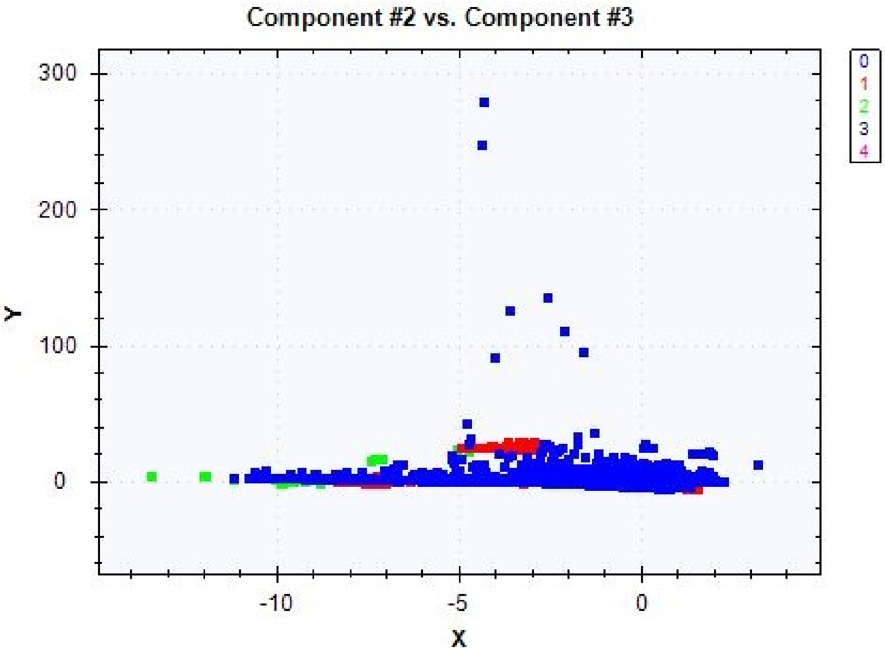
As you can see from this code, we are building scatter plots between two principal components at a time, for the first six components. When you run this code, you will see plots similar to the following ones.

The first plot is between the first and second principal components, and looks as follows:

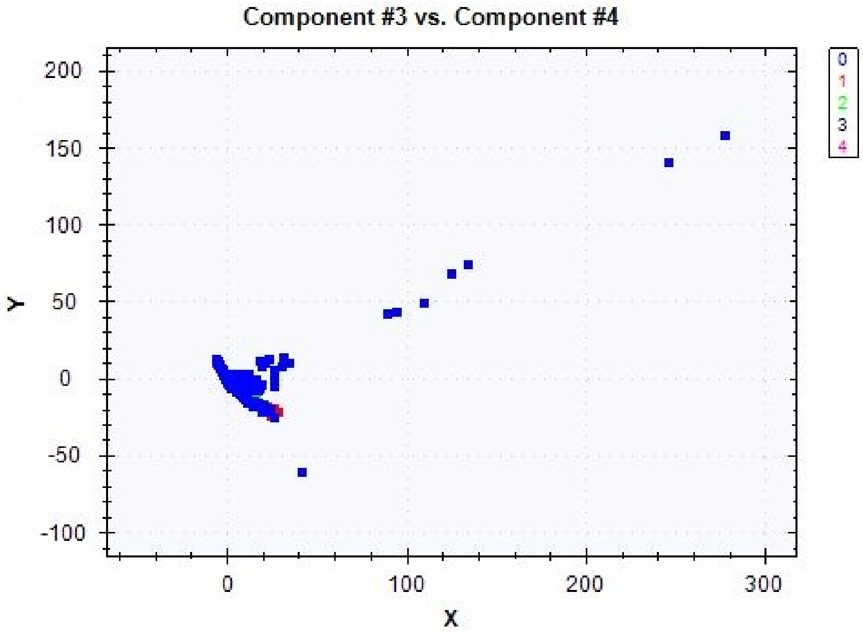


The blue dots represent the normal connections, and the other dots with different colors represent the cyber attacks. We can see some distinctions in the distributions among different classes, but the pattern does not seem so strong.

The following plot is between the second and third components:



Lastly, the following plot is between the third and fourth components:



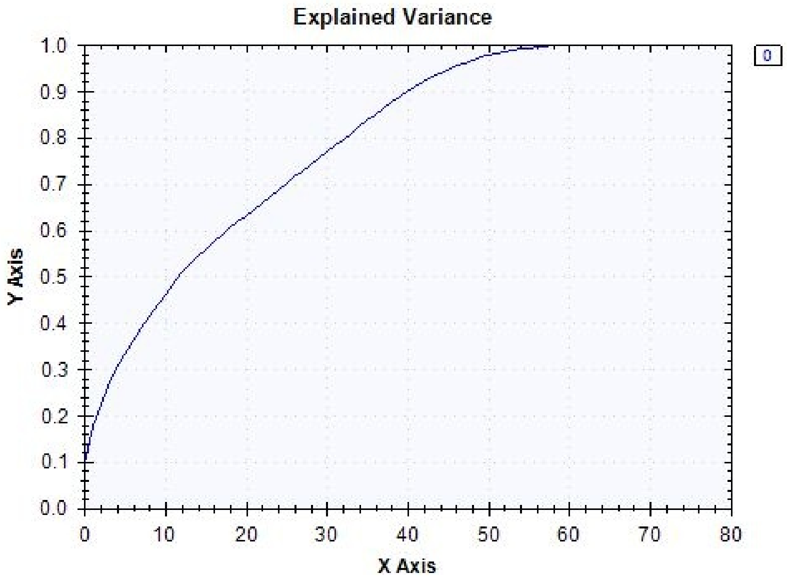
We cannot really see many distinctions among different classes in the last plot. Although the pattern does not seem very strong, previous scatter plots show some differences in the distributions. It is especially more difficult to visually see the distinctions on two-dimensional plots. If we take this to higher-dimensional space, which our anomaly detection model is going to be looking at, the differences in the patterns will become more noticeable.

Let's now look at the amount of variance explained from the principal components. The following code shows how we can get the cumulative proportion of variances explained, and display it in a line chart:

```
double[] explainedVariance = pca.Components
    .Select(x => x.CumulativeProportion)
    .Where(x => x < 1)
    .ToArray();

DataSeriesBox.Show(
    explainedVariance.Select((x, i) => (double)i),
    explainedVariance
).SetTitle("Explained Variance");
System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "explained-variance.csv"),
    explainedVariance.Select((x, i) => String.Format("{0},{1:0.0000}", i, x))
);
```

If you look at this code more closely, the `Components` property in the `pca` object contains information about the proportion of variance explained. We can iterate through each component and get the cumulative proportion by using the

`CumulativeProportion` property. Once we have extracted these values, we then use the `DataSeriesBox` class to display a line chart that shows the cumulative proportion of variance explained. The output looks like the following:



Now, we have successfully created PCA features and have full PCA-transformed data. You can use the following code to export this data:

```
Console.WriteLine("* Exporting pca-transformed feature set...");
System.IO.File.WriteAllLines(
    Path.Combine(
        dataDirPath,
        "pca-transformed-features.csv"
    ),
    transformed.Select(x => String.Join(",", x))
);
System.IO.File.WriteAllLines(
    Path.Combine(
        dataDirPath,
        "pca-transformed-labels.csv"
    ),
    labels.Select(x => x.ToString())
);
```

The full code for the feature engineering step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.9/FeatureEngineering.cs`.

Principal component classifier for anomaly detection

We have compiled everything and are now ready to start building an anomaly detection model for the cyber attack detection project. As mentioned previously, we are going to use the data of the distributions of principal components from the normal connections group, and take it as the normal ranges of principal components. For any records that deviate from these normal ranges of the principal component values, we are going to flag them as abnormal and potential cyber attacks.

Preparation for training

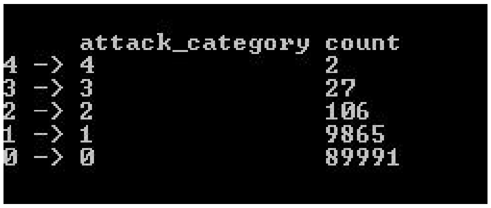
First, let's load the features data that we created from the feature engineering step. You can use the following code to load the PCA-transformed data and the labels data:

```
// Read in the Cyber Attack dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "pca-transformed-features.csv");
Console.WriteLine("Loading {0}\n\n", dataPath);
var featuresDF = Frame.ReadCsv(
    dataPath,
    hasHeaders: false,
    inferTypes: true
);
featuresDF.RenameColumns(
    featuresDF.ColumnKeys.Select((x, i) => String.Format("component-{0}", i + 1))
);

int[] labels = File.ReadLines(
    Path.Combine(dataDirPath, "pca-transformed-labels.csv")
).Select(x => int.Parse(x)).ToArray();
featuresDF.AddColumn("attack_category", labels);
```

Let's quickly look at the distributions of our target classes. The code to count by each target class is as follows:

```
var count = featuresDF.AggregateRowsBy<string, int>(
    new string[] { "attack_category" },
    new string[] { "component-1" },
    x => x.ValueCount
).SortRows("component-1");
count.RenameColumns(new string[] { "attack_category", "count" });
count.Print();
```

Once you run this code, you will see the following output:



```
     attack_category count
4 -> 4               2
3 -> 3               27
2 -> 2               106
1 -> 1               9865
0 -> 0               89991
```

As expected, the majority of the samples belong to the 0 class, which is the normal group, and the rest combined are the minority (about 10%) in our sample set. This is a more realistic view of cyber attacks. Cyber attacks happen way less frequently than normal connections.

For illustration purposes, we are going to use the first `27` principal components that explain about 70% of the overall variance in the dataset. You can experiment with different numbers of principal components and see how model performances change. The following code shows how we created a training set using the first `27` principal components:

```
// First 13 components explain about 50% of the variance
// First 19 components explain about 60% of the variance
// First 27 components explain about 70% of the variance
// First 34 components explain about 80% of the variance
int numComponents = 27;
string[] cols = featuresDF.ColumnKeys.Where((x, i) => i < numComponents).ToArray();

// First, compute distances from the center/mean among normal events
var normalDF = featuresDF.Rows[
    featuresDF["attack_category"].Where(x => x.Value == 0).Keys
].Columns[cols];

double[][] normalData = BuildJaggedArray(
    normalDF.ToArray2D<double>(), normalDF.RowCount, cols.Length
);
```

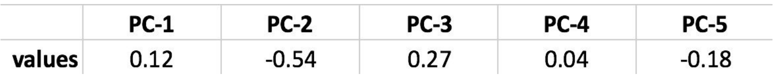
If you look at this code closely, you will notice that we are creating `normalDF` and `normalData` variables with normal connection samples only. As mentioned previously, we want to learn only from the normal data, so that we can flag any outliers and extreme deviations from the normal ranges of principal components. We are going to use these variables to build a principal component classifier for the cyber attack detection in the following section.

Building a principal component classifier

In order to build a principal component classifier, which will flag those events that deviate from the normal connections, we need to calculate the distance between a record and the distributions of normal connections. We are going to use a distance metric, the **Mahalanobis distance**, which measures the distance between a point and a distribution. For the standardized principal components, like those here, the equation to compute the **Mahalanobis distance** is as follows:

$$Distance = \sqrt{\sum_{i=1}^{n} \frac{C_i^2}{var_i}}$$

C_i in this equation represents the value of each principal component, and var_i represents the variance of each principal component. Let's take a look at the following example:



	PC-1	PC-2	PC-3	PC-4	PC-5
values	0.12	-0.54	0.27	0.04	-0.18

Assume you have 5 principal components with values as shown in this image and assume the variance for each principal is 1 for simplicity and demonstration purposes, then you can compute the **Mahalanobis distance** as the following:

$$\sqrt{\frac{0.12^2}{1} + \frac{-0.54^2}{1} + \frac{0.27^2}{1} + \frac{0.04^2}{1} + \frac{-0.18^2}{1}}$$

And the computed **Mahalanobis distance** for this example is 0.64. For a more detailed description of this distance metric, it is recommended that you review the following Wikipedia page: `https://en.wikipedia.org/wiki/Mahalanobis_distance`, or the following research paper: `https://users.cs.fiu.edu/~chens/PDF/ICDM03_WS.pdf`.

We implemented the Mahalanobis distance equation as a helper function, `ComputeDistances`, and it looks as follows:

```
private static double[] ComputeDistances(double[][] data, double[] componentVariances)
{

    double[] distances = data.Select(
        (row, i) => Math.Sqrt(
            row.Select(
                (x, j) => Math.Pow(x, 2) / componentVariances[j]
            ).Sum()
        )
    ).ToArray();

    return distances;
}
```

As you can see from this code snippet, the `ComputeDistances` method takes in two arguments—`data` and `componentVariances`. The variable `data` is a two-dimensional array that we want to compute distances for, and the `componentVariances` variable is the variance of the principal components that are learned from the normal network connections data. In order to compute the variances of the principal components, we use the following helper function:

```
private static double[] ComputeVariances(double[][] data)
{
    double[] componentVariances = new double[data[0].Length];

    for (int j = 0; j < data[0].Length; j++)
    {
        componentVariances[j] = data
            .Select((x, i) => Math.Pow(data[i][j], 2))
            .Sum() / data.Length;
    }

    return componentVariances;
}
```

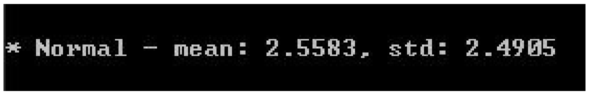
As you can see from this code snippet, it is computing the variances of individual columns, where each column of this two-dimensional array represents each principal component. In order to compute the distances of individual records from the distribution of normal network connections data, we can simply pass the two-dimensional array to the helper function `ComputeDistances`, as follows:

```
double[] distances = ComputeDistances(normalData);
```

Now that we have computed the distances of individual records, let's analyze how the ranges for the normal connections look. We used the following code to calculate the mean and standard deviation of the distances, and a histogram to visualize the overall distance distributions:

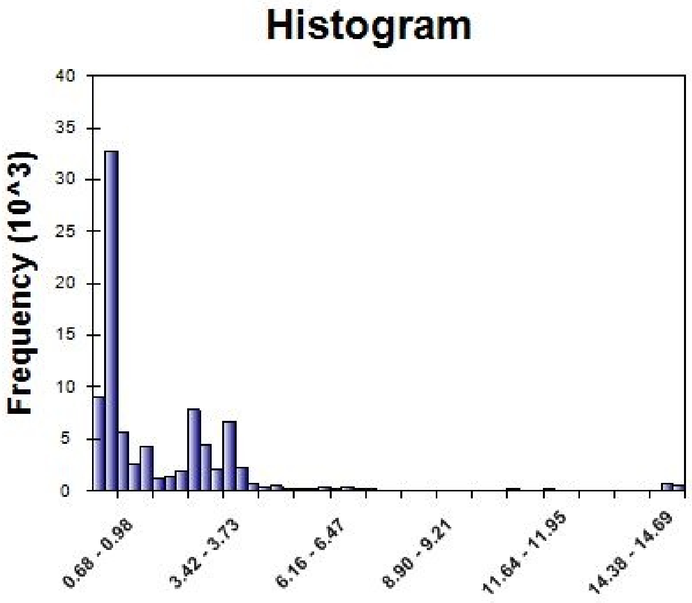
```
double meanDistance = distances.Average();
double stdDistance = Math.Sqrt(
    distances
    .Select(x => Math.Pow(x - meanDistance, 2))
    .Sum() / distances.Length
);

Console.WriteLine(
    "* Normal - mean: {0:0.0000}, std: {1:0.0000}",
    meanDistance, stdDistance
);

HistogramBox.Show(
    distances,
    title: "Distances"
)
.SetNumberOfBins(50);
```

When you run this code, you will see the following output for the mean and standard deviation of the distance metrics for normal connections:



```
* Normal - mean: 2.5583, std: 2.4905
```

And the histogram looks as follows:



As you can see from these outputs, the majority of the distances are very small, which suggests that the non-attack and normal connections are typically clustered together closely. With this information about the distance distributions within the normal connections group, let's start looking to see if we can build a detection model by flagging certain network connections that go beyond the normal range of distances.

The following code shows how we computed the distances of cyber attack connections from the distribution of normal network connections:

```
// Detection
var attackDF = featuresDF.Rows[
    featuresDF["attack_category"].Where(x => x.Value > 0).Keys
].Columns[cols];

double[][] attackData = BuildJaggedArray(
    attackDF.ToArray2D<double>(), attackDF.RowCount, cols.Length
);

double[] attackDistances = ComputeDistances(attackData, normalVariances);
```

As you can see from this code, we first created a variable, called `attackData`, which contains all the cyber attack connections from our training set. Then, we used the `ComputeDistances` method to calculate the distances of individual records in the cyber attack connections group.

Now, we are ready to start flagging suspicious network connections based on the distance metrics that we just calculated. Let's take a look at the following code first:

```
// 5-10% false alarm rate
for (int i = 4; i < 10; i++)
{
    double targetFalseAlarmRate = 0.01 * (i + 1);
    double threshold = Accord.Statistics.Measures.Quantile(
        distances,
        1 - targetFalseAlarmRate
    );

    int[] detected = attackDistances.Select(x => x > threshold ? 1 : 0).ToArray();

    EvaluateResults(attackLabels, detected, targetFalseAlarmRate);
}
```

As you can see from this code, we decide the threshold based on the distribution of distances within the normal connections group. For example, if our target is to have a 5% false alarm rate, we flag all the connections that have distances from the normal range greater than the 95% percentile of the distribution of distances within the normal connections group. More specifically, the 95% percentile of the normal connections' distance distribution in our case was 5.45. So, in this case, we will flag all the connections that have distances from the normal range greater than 5.45 as cyber attacks. We repeat this process for the false alarm rates from 5% to 10%. We will discuss the performance of this anomaly detection model in more detail in the following model-evaluation step.

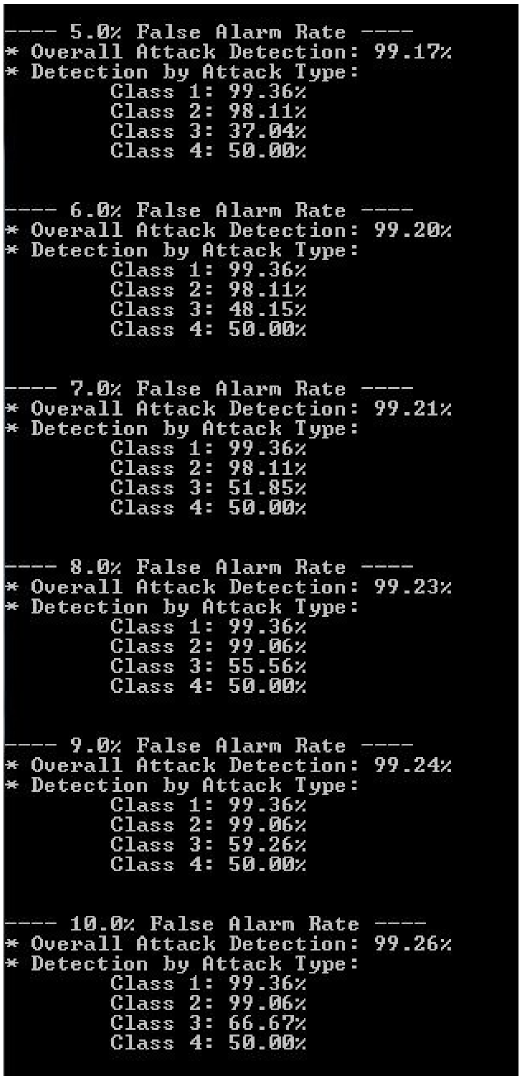
The full code for the model-building step can be found at this link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.9/Modeling.cs`.

Evaluating anomaly detection models

We built an anomaly detection model for cyber attacks in the previous model-building step. In the previous code, you might have noticed that we are using a function named `EvaluateResults`. It is a helper function that we wrote for evaluating model performances. Let's take a look at the following code:

```
private static void EvaluateResults(int[] attackLabels, int[] detected, double
targetFalseAlarmRate)
{
    double overallRecall = (double)detected.Sum() / attackLabels.Length;

    double[] truePositives = new double[4];
    double[] actualClassCounts = new double[4];

    for (int i = 0; i < attackLabels.Length; i++)
    {
        actualClassCounts[attackLabels[i] - 1] += 1.0;

        if (detected[i] > 0)
        {
            truePositives[attackLabels[i] - 1] += 1.0;
        }
    }

    double[] recalls = truePositives.Select((x, i) => x /
actualClassCounts[i]).ToArray();

    Console.WriteLine("\n\n---- {0:0.0}% False Alarm Rate ----", targetFalseAlarmRate *
100.0);
    Console.WriteLine("* Overall Attack Detection: {0:0.00}%", overallRecall * 100.0);
    Console.WriteLine(
        "* Detection by Attack Type:\n\t{0}",
        String.Join("\n\t", recalls.Select(
            (x, i) => String.Format("Class {0}: {1:0.00}%", (i + 1), x * 100.0))
        )
    );
}
```

As you can see from this code, we are interested in two metrics: overall cyber attack detection rate and per-class detection rate. The evaluation results look as follows:



```
---- 5.0% False Alarm Rate ----
* Overall Attack Detection: 99.17%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 98.11%
        Class 3: 37.04%
        Class 4: 50.00%


---- 6.0% False Alarm Rate ----
* Overall Attack Detection: 99.20%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 98.11%
        Class 3: 48.15%
        Class 4: 50.00%


---- 7.0% False Alarm Rate ----
* Overall Attack Detection: 99.21%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 98.11%
        Class 3: 51.85%
        Class 4: 50.00%


---- 8.0% False Alarm Rate ----
* Overall Attack Detection: 99.23%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 99.06%
        Class 3: 55.56%
        Class 4: 50.00%


---- 9.0% False Alarm Rate ----
* Overall Attack Detection: 99.24%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 99.06%
        Class 3: 59.26%
        Class 4: 50.00%


---- 10.0% False Alarm Rate ----
* Overall Attack Detection: 99.26%
* Detection by Attack Type:
        Class 1: 99.36%
        Class 2: 99.06%
        Class 3: 66.67%
        Class 4: 50.00%
```

The overall results look good with over 99% detection rates. At 5% false alarm rate, about 99.1% of the cyber attacks are detected. However, if we look closer at the per-class detection rates, we can see their weaknesses and strengths. At 5% false alarm rate, our model does very well for detecting classes 1 and 2, which are `dos` and `probe` attacks. On the other hand, our model does poorly in detecting classes 3 and 4, which are `r2l` and `u2r` attacks. As you can see from this output, as we increase the target false alarm rates, the overall and per-class detection rates increase as well. In a real-world situation, you will have to evaluate the trade-offs between a higher detection rate and a higher false alarm rate, and make a decision on the target false alarm rate that meets your business requirements.

Summary

In this chapter, we built our very first anomaly detection model that can detect cyber attacks. At the beginning of this chapter, we discussed how this type of anomaly detection model can be used and applied to real-life situations, and how developing an anomaly detection model is different from other ML models that we have built so far. Then, we started analyzing the distributions of target classes and various features to understand the dataset better. While we were analyzing this dataset, we also noticed how there are more cyber attack samples than normal connection samples, which is unrealistic in real life. In order to simulate real-life situations, where abnormal malicious connections occur much less frequently than normal connections, we randomly sub-selected the normal and malicious connection samples so that 90% of the training set were normal connections and only 10% were cyber attack examples.

With this sub-selected training set, we applied PCA to the normal connections data to find out the normal ranges of principal components. Using the **Mahalanobis distance** metric, we computed the distances between individual records from the distributions of normal connections. During the model-building step, we experimented with different thresholds based on the target false alarm rates. Using 5% to 10% false alarm rates, we built cyber attack detection models and evaluated their performance. In our model-evaluation step, we noticed that the overall detection rates were over 99%, while a closer look at per-attack detection rates exposed the weaknesses and strengths of the models. We also noticed that as we sacrifice and increase the false alarm rates, the overall cyber attack detection rates improved. When applying this anomaly detection technique, it becomes necessary to understand this tradeoff between the false alarm rate and detection rate, and make a decision based on pertinent business requirements.

In the next chapter, we are going to expand our knowledge and experience in building anomaly detection models. We are going to work on a credit card fraud detection project with a credit card dataset. On top of the PCA-based anomaly detection model, we are going to discuss how to use a one-class support vector machine for anomaly detection.

Credit Card Fraud Detection

In the previous chapter, we built our first anomaly detection model using **Principal Component Analysis** (**PCA**) and saw how we can detect cyber attacks using principal components. Similar to cyber attack or network intrusion problems, anomaly detection models are frequently used for fraud detection. Various organizations in many industries, such as financial services, insurance companies, and government agencies, often come across fraudulent cases. Especially in financial sectors, frauds are directly related to monetary losses and these fraudulent cases can come in many different guises, such as stolen credit cards, accounting forgeries, or fake checks. Because these events occur relatively rarely, it is difficult and tricky to detect these fraudulent cases.

In this chapter, we are going to discuss how we can build an anomaly detection model for credit card fraud detection. We are going to use an anonymized credit card dataset that contains a large portion of normal credit card transactions and relatively fewer fraudulent credit card transactions. We will first look at the structure of the dataset, the distribution of the target classes, and the distributions of various anonymized features. Then, we are going to start applying PCA and building standardized principal components that will be used as features for our fraud detection model. In the model building step, we are going to experiment with two different approaches to building fraud detection models—the **Principal Component Classifier** (**PCC**) that is similar to what we built in `Chapter 9`, *Cyber Attack Detection* and the one-class **Support Vector Machine** (**SVM**) that learns from normal credit card transactions and detects any anomalies. With these models built, we are going to evaluate their anomaly detection rates and compare their performances for credit card fraud detection.

In this chapter, we will cover the following topics:

- Problem definition for the credit card fraud detection project
- Data analysis for the anonymized credit card dataset
- Feature engineering and PCA
- The one-class SVM versus the PCC

- Evaluating anomaly detection models

Problem definition

Credit card fraud is relatively common among other fraudulent events, and can happen in our daily lives. There are various ways credit card fraud can happen. Credit cards can be lost or stolen and then used by a thief. Another way credit card fraud can occur is that your identity might have been exposed to malicious persons who then use your identity to open a new credit card account, or even take over your existing credit card accounts. Scammers can even use telephone phishing for credit card fraud. As there are many ways credit card fraud can happen, many credit card holders are exposed to the risk of this type of fraud, and having a proper way to prevent them from happening has become essential in our daily lives. Many credit card companies have employed various measures to prevent and detect these types of fraudulent activities, using various **machine learning** (**ML**) and anomaly-detection technologies.

In this chapter, we are going to work on building a credit card fraud detection model by using and expanding our knowledge about building anomaly detection models. We will be using an anonymized credit card dataset that can be found at the following link: `https://www.kaggle.com/mlg-ulb/creditcardfraud/data`. This dataset has about 285,000 credit card transactions, and only about 0.17% of those transactions are fraudulent transactions, which reflects a real-life situation very well. With this data, we are going to look at how the dataset is structured, and then start looking at the distributions of the target and feature variables. Then, we will be building features by using PCA, similar to what we did in `Chapter 9`, *Cyber Attack Detection*. For building credit card fraud detection models, we are going to experiment with both the PCC, similar to the one we built in `Chapter 9`, *Cyber Attack Detection*, and the one-class SVM, which learns from normal credit card transactions and decides whether a new transaction is fraudulent or not. Lastly, we are going to look at false alarm and fraud detection rates to evaluate and compare the performances of these models.

To summarize our problem definition for the credit card fraud detection project:

- What is the problem? We need an anomaly detection model for fraudulent credit card transactions that can identify, prevent, and stop potential fraudulent credit card activities.

- Why is it a problem? Every credit card holder is exposed to the risks of becoming the victim of credit card fraud, and without being properly prepared for such malicious attempts, the number of credit card fraud victims is going to increase. With a credit card fraud detection model, we can prevent and stop potential fraudulent credit card transactions from happening.
- What are some of the approaches to solving this problem? We are going to use anonymized credit card data that is publicly available, and has lots of normal credit card transactions and a small number of fraudulent transactions. We are going to apply PCA to this data and experiment with the PCC and the one-class SVM models for fraud detection.
- What are the success criteria? Since any credit card fraud event will result in monetary loss, we want a high fraud detection rate. Even if there are some false positives or false alarms, it is better to flag any suspicious credit card activities to prevent any fraudulent transactions from going through.

Data analysis for anonymized credit card data

Let's now start looking at the credit card dataset. As mentioned before, we are going to use the dataset that is available at the following link: `https://www.kaggle.com/mlg-ulb/creditcardfraud/data`. It is a dataset that contains about 285,000 records of credit card transactions, where some of them are fraudulent transactions and the majority of the records are normal credit card transactions. Due to confidentiality issues, the feature names in the dataset are anonymized. We will be using the `creditcard.csv` file, which can be downloaded from the link.

Target variable distribution

The first thing we are going to examine is the distribution of fraudulent and non-fraudulent credit card transactions in the dataset. In the dataset, the column named `Class` is the target variable that is encoded with `1` for fraudulent credit card transactions and `0` for non-fraudulent transactions. You can use the following code to first load the data into a Deedle data frame:

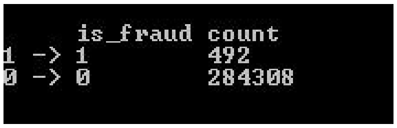
```
// Read in the Credit Card Fraud dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-your-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "creditcard.csv");
Console.WriteLine("Loading {0}\n\n", dataPath);
var df = Frame.ReadCsv(
    dataPath,
    hasHeaders: true,
    inferTypes: true
);
```

This dataset has headers that represent each of the features and the target class, so we are loading this data with the `hasHeaders: true` flag. Now that we have the data loaded, you can use the following code to analyze the distribution of the target classes:

```
// Target variable distribution
var targetVarCount = df.AggregateRowsBy<string, int>(
    new string[] { "Class" },
    new string[] { "V1" },
    x => x.ValueCount
).SortRows("V1");
targetVarCount.RenameColumns(new string[] { "is_fraud", "count" });

targetVarCount.Print();

DataBarBox.Show(
    targetVarCount.GetColumn<string>("is_fraud").Values.ToArray(),
    targetVarCount["count"].Values.ToArray()
).SetTitle(
    "Counts by Target Class"
);
```

As you might be familiar with this function already, we are using the `AggregateRowsBy` function in a Deedle data frame to group rows by the column `Class`, and then count the number of records in each target class. Since the column name, `Class`, is not a good representative of what our target class is and what it means, we renamed it with another name, `is_fraud`. As you can see from this code, you can use the `RenameColumns` function with an array of strings for new column names to rename the feature names. Lastly, we used the `DataBarBox` class in the Accord.NET framework to display a bar plot that visually shows the distributions of the target classes.

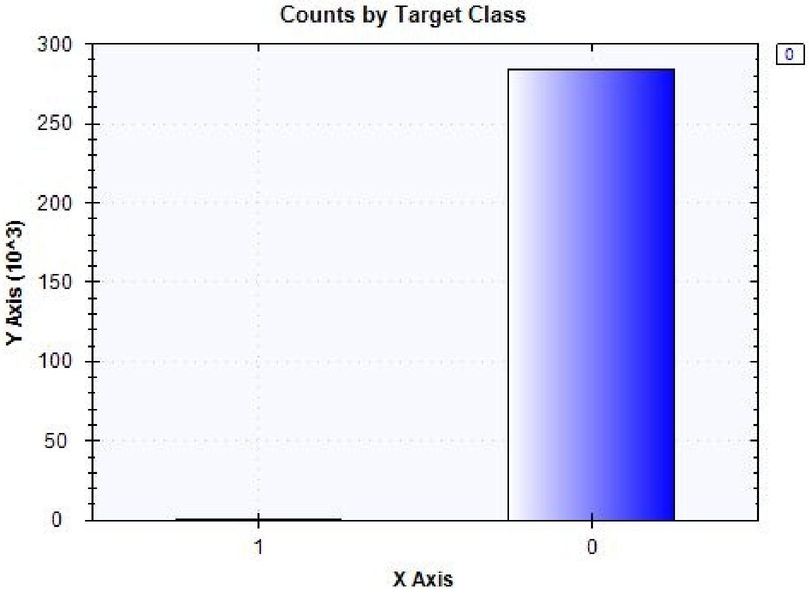
The following output shows the distribution of the target classes:



```
     is_fraud count
1 -> 1        492
0 -> 0        284308
```

As you can see from this output, there is a large gap between the number of fraudulent credit card transactions and non-fraudulent credit card transactions. We only have 492 records of frauds and over 284,000 records of non-frauds.

The following is a bar plot that the code generates for visually displaying the distribution of target classes:



As expected from the previous output, there is a large gap between the number of records that belong to the target class, **1**, which represents fraud, and the number of records that belong to the target class, **0**, which represents non-fraud and normal credit card transactions. This large gap is expected as credit card frauds happen relatively rarely, compared to the large number of normal everyday credit card transactions. This large class imbalance makes it difficult for most ML models to accurately learn how to identify frauds from non-frauds.

Feature distributions

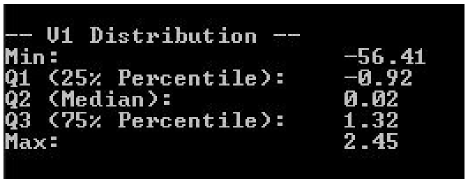
The features, except for the transactional amounts, we have in this data are anonymized due to confidentiality issues. Because we do not know what each feature represents and what each feature means, it will be difficult to deduce any intuitive insights from the feature analysis. However, it is still helpful to understand how each of the features is distributed, how the distribution of each feature differs from the others, and whether there is any noticeable pattern we can derive from the set of features.

Let's first take a look at the code. The following code shows how we can compute and visualize the distributions of the features:

```
// Feature distributions
foreach (string col in df.ColumnKeys)
{
    if (col.Equals("Class") || col.Equals("Time"))
    {
        continue;
    }

    double[] values = df[col].DropMissing().ValuesAll.ToArray();

    Console.WriteLine(String.Format("\n\n-- {0} Distribution -- ", col));
    double[] quartiles = Accord.Statistics.Measures.Quantiles(
        values,
        new double[] { 0, 0.25, 0.5, 0.75, 1.0 }
    );
    Console.WriteLine(
        "Min: \t\t\t{0:0.00}\nQ1 (25% Percentile): \t{1:0.00}\nQ2 (Median):
\t\t{2:0.00}\nQ3 (75% Percentile): \t{3:0.00}\nMax: \t\t\t{4:0.00}",
        quartiles[0], quartiles[1], quartiles[2], quartiles[3], quartiles[4]
    );

    HistogramBox.Show(
        values,
        title: col
    )
    .SetNumberOfBins(50);
}
```

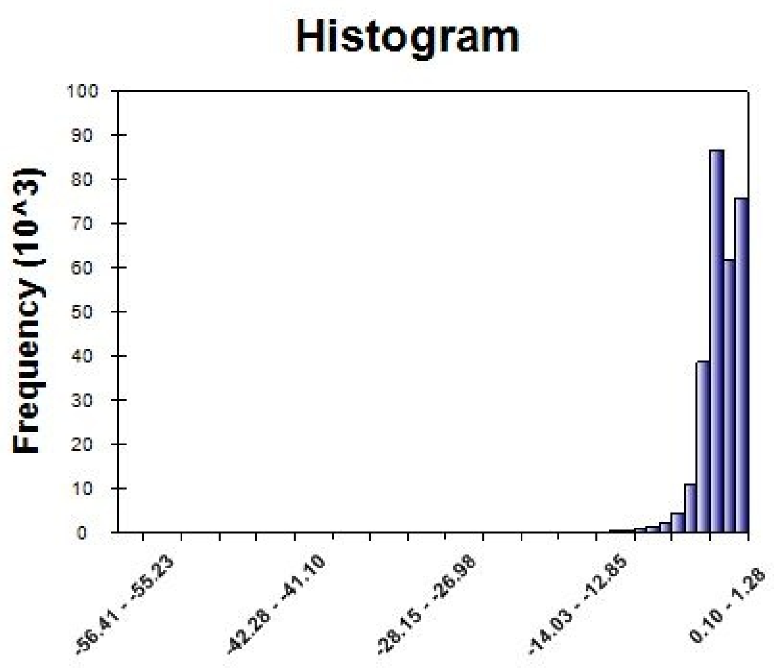
As you can see from this code, we are computing the quartiles. As you might recall, quartiles are the points that separate the data into four different sections. The first quartile is the middle point between the minimum and the median, the second quartile is the median, and the third quartile is the middle point between the median and the maximum. You can easily compute the quartiles by using the `Accord.Statistics.Measures.Quantiles` function. After we compute the quartiles, we build histogram plots for each feature to visualize the distributions, using the `HistogramBox` class in the Accord.NET framework. Let's take a look at some of the outputs from this code.

The first distribution we are going to look at is for the `V1` feature, and the quartiles for `V1` look like the following:



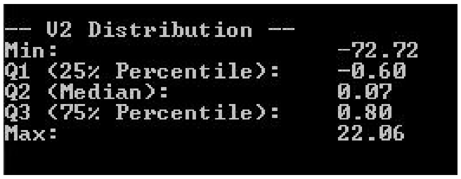
```
-- V1 Distribution --
Min:                        -56.41
Q1 (25% Percentile):        -0.92
Q2 (Median):                0.02
Q3 (75% Percentile):        1.32
Max:                        2.45
```

It seems the distribution of the `V1` feature is skewed towards the negative direction. Even though the median is about 0, the negative values range from -56.41 to 0, while the positive values range only from 0 to 2.45. The following is the histogram output from the previous code:



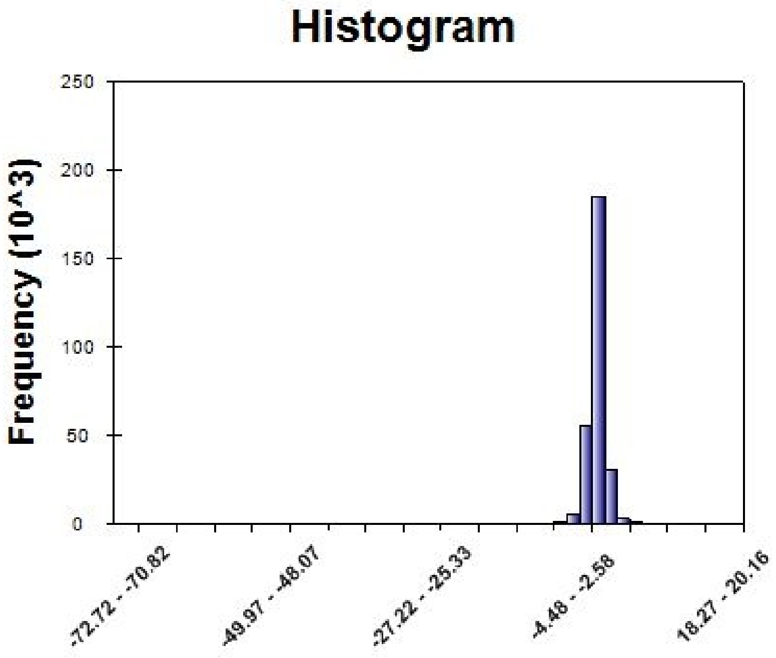
As expected, the histogram plot shows left skewness in the distribution of the feature, `V1`, while the majority of the values are around 0.

Next, let's look at the distribution of the second feature, `V2`, where the output looks as follows:



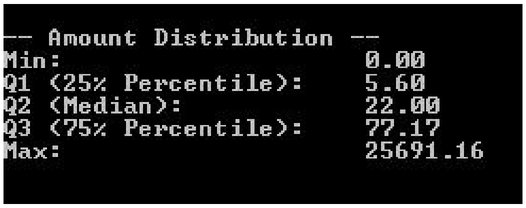
```
-- V2 Distribution --
Min:                        -72.72
Q1 (25% Percentile):        -0.60
Q2 (Median):                0.07
Q3 (75% Percentile):        0.80
Max:                        22.06
```

The histogram for `v2` looks like the following:



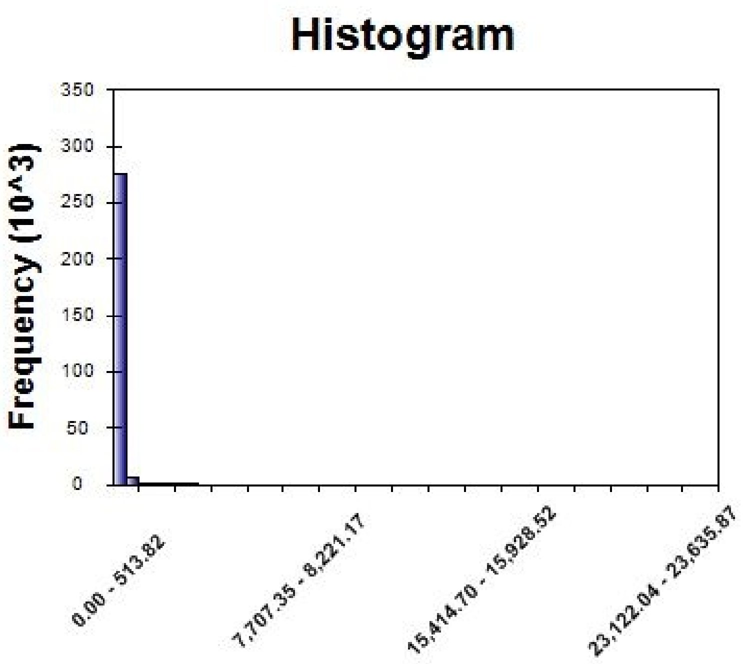
It seems the values are centered around 0, although there are some extreme values in the negative direction and in the positive direction. The skewness is less obvious, compared to the previous feature, `v1`.

Lastly, let's look at the distribution of the `amount` feature, which can tell us the range of transaction amounts. The following are the quartiles for the `amount` feature:



```
-- Amount Distribution --
Min:                        0.00
Q1 (25% Percentile):        5.60
Q2 (Median):                22.00
Q3 (75% Percentile):        77.17
Max:                        25691.16
```

It seems any credit card transaction can take any positive number that ranges between 0 and 25,691.16 as a transaction amount. The following is a histogram for the `amount` feature:



As expected, we can see there is a long tail to the right. This is somewhat expected, as the spending pattern for each individual differs from any other. Some people might typically buy moderately priced items, while some others might buy very expensive items.

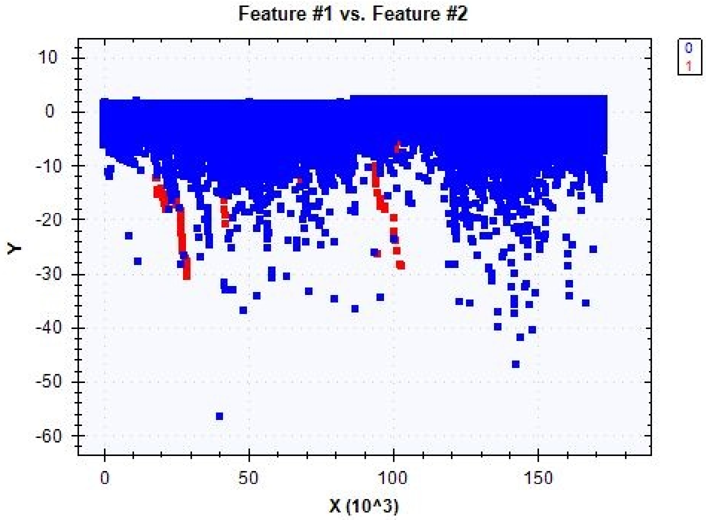
Lastly, let's take a brief look at how well the current feature set separates fraudulent credit card transactions from non-fraudulent transactions. Let's take a look at the following code first:

```
// Target Var Distributions on 2-dimensional feature space
double[][] data = BuildJaggedArray(
    df.ToArray2D<double>(), df.RowCount, df.ColumnCount
);
int[] labels = df.GetColumn<int>("Class").ValuesAll.ToArray();

double[][] first2Components = data.Select(
    x => x.Where((y, i) => i < 2
).ToArray()).ToArray();
ScatterplotBox.Show("Feature #1 vs. Feature #2", first2Components, labels);

double[][] next2Components = data.Select(
    x => x.Where((y, i) => i >= 1 && i <= 2).ToArray()
).ToArray();
ScatterplotBox.Show("Feature #2 vs. Feature #3", next2Components, labels);

next2Components = data.Select(
    x => x.Where((y, i) => i >= 2 && i <= 3).ToArray()
).ToArray();
```

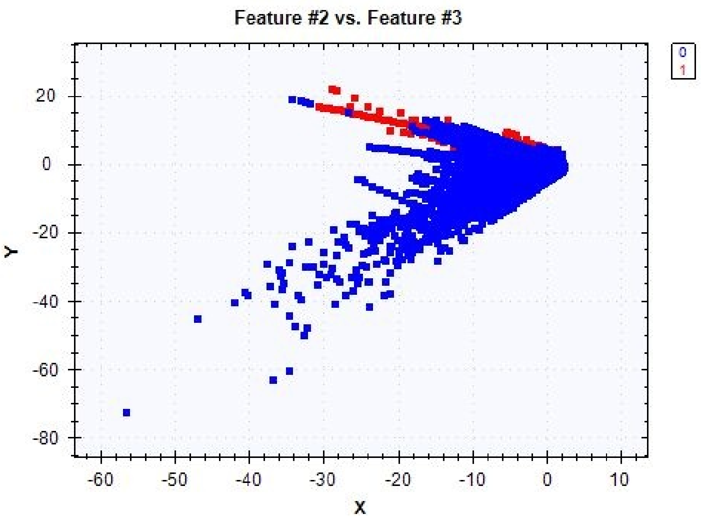
```
ScatterplotBox.Show("Feature #3 vs. Feature #4", next2Components, labels);
```

As you can see from this code, we first convert the Deedle data frame variable, `df`, to a two-dimensional array variable, `data`, to build scatter plots. Then, we take the first two features and display a scatter plot that shows the distribution of the target classes across these first two features. We repeat this process twice more for the second, third, and fourth features.

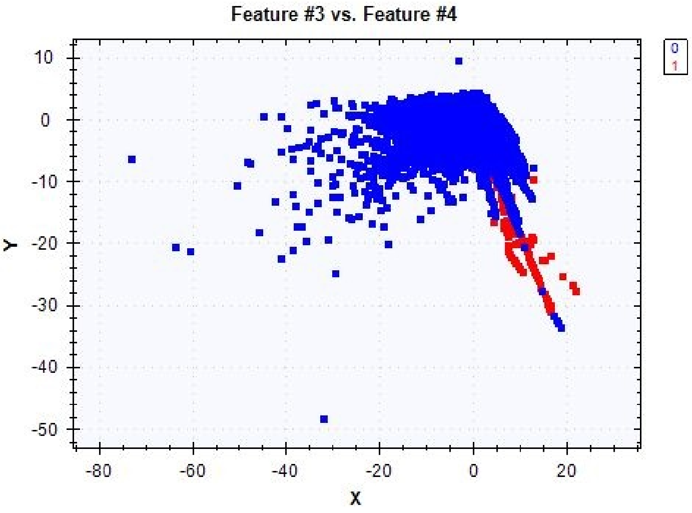
The following scatter plot is the distribution of target classes across the first and second features in our dataset:



From this scatter plot, it is quite difficult, if not impossible, to separate the frauds (encoded as 1) from the non-frauds (encoded as 0). Let's look at the scatter plot between the next two features:



Similar to the case of the first two features, there does not seem to be a clear line separating frauds from non-frauds. Lastly, the following is a scatter plot of the target classes between the third and fourth feature:



From looking at this scatter plot, it will be difficult to draw a clear line that separates the two target classes. The fraudulent transactions seem to reside more in the bottom-right side of this scatter plot, but the pattern is weak. In the following section, we will try to build features that better separate the two target classes.

The full code for this data analysis step can be found at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.10/DataAnalyzer.cs`.

Feature engineering and PCA

So far, we have analyzed what the distributions of the target and feature variables look like. In this chapter, we are going to focus on building features, using PCA.

Preparation for feature engineering

In order to fit the PCA, we will have to prepare our data first. Let's quickly look at the following code to load the credit card fraud data into Deedle's data frame:

```
// Read in the Credit Card Fraud dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "creditcard.csv");
Console.WriteLine("Loading {0}\n\n", dataPath);
var df = Frame.ReadCsv(
    dataPath,
    hasHeaders: true,
    inferTypes: true
);

Console.WriteLine("* Shape: {0}, {1}\n\n", df.RowCount, df.ColumnCount);
```

Now that we have loaded the data into a variable, named `df`, we are going to have to split the data into two sets, one for normal credit card transaction data and another for fraudulent transaction data, so that we can fit PCA with the normal transactions only. Take a look at the following code for how we can separate out the normal transactions from the raw dataset:

```
string[] featureCols = df.ColumnKeys.Where(
    x => !x.Equals("Time") && !x.Equals("Class")
).ToArray();

var noFraudData = df.Rows[
    df["Class"].Where(x => x.Value == 0.0).Keys
].Columns[featureCols];
double[][] data = BuildJaggedArray(
    noFraudData.ToArray2D<double>(), noFraudData.RowCount, featureCols.Length
);
```

If you recall from the previous data analysis step, the target variable, `Class`, is encoded as 1 for fraudulent transactions and 0 for non-fraudulent transactions. As you can see from the code, we created a data frame, `noFraudData`, with only normal credit card transaction records. Then, we converted this data frame into a two-dimensional double array that will be used to fit the PCA, using the helper function, `BuildJaggedArray`. The code for this helper function looks like the following:

```
private static double[][] BuildJaggedArray(double[,] ary2d, int rowCount, int colCount)
```

```
{
    double[][] matrix = new double[rowCount][];
    for (int i = 0; i < rowCount; i++)
    {
        matrix[i] = new double[colCount];
        for (int j = 0; j < colCount; j++)
        {
            matrix[i][j] = double.IsNaN(ary2d[i, j]) ? 0.0 : ary2d[i, j];
        }
    }
    return matrix;
}
```

This code should look familiar, as we have used it in a number of previous chapters.

The next thing we need to do is convert the entire data frame, including both non-fraudulent and fraudulent records, into a two-dimensional array. Using the trained PCA, we are going to transform this newly created two-dimensional array that will later be used for building credit card fraud detection models. Let's take a look at the following code:

```
double[][] wholeData = BuildJaggedArray(
    df.Columns[featureCols].ToArray2D<double>(), df.RowCount, featureCols.Length
);
int[] labels = df.GetColumn<int>("Class").ValuesAll.ToArray();
```

As you can see from this code snippet, we are simply converting the entire data frame, `df`, into a two-dimensional array, `wholeData`, by using the `BuildJaggedArray` function.

Fitting a PCA

We are now ready to fit a PCA using the non-fraudulent credit card data. Similar to what we did in `Chapter 9`, *Cyber Attack Detection*, we are going to use the following code to fit a PCA:

```
var pca = new PrincipalComponentAnalysis(
    PrincipalComponentMethod.Standardize
);
pca.Learn(data);
```

As you can see from this code, we are using the `PrincipalComponentAnalysis` class in the Accord.NET framework to train a PCA. One more thing to note here is how we used `PrincipalComponentMethod.Standardize`. Since PCA is sensitive to the scales of the features, we are standardizing the feature values first and then fitting a PCA. Using this trained PCA, we can transform the whole data that contains both fraudulent and non-fraudulent transactions. The code for applying PCA transformation to the dataset looks as follows:

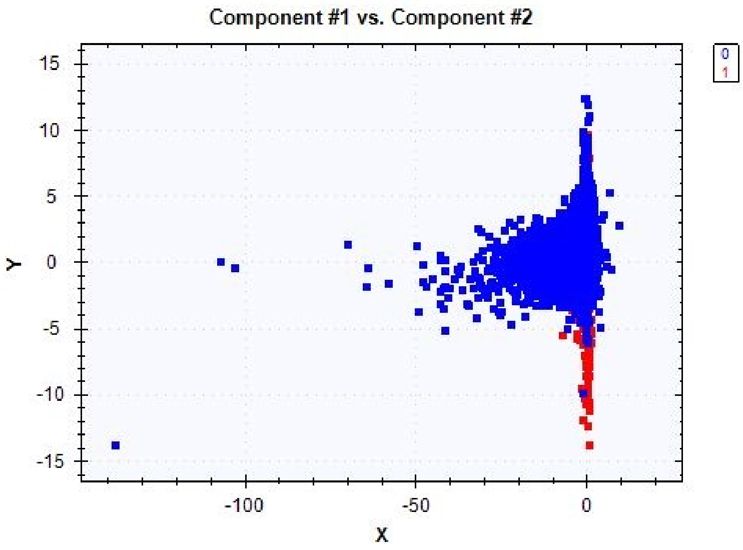
```
double[][] transformed = pca.Transform(wholeData);
```

Now, we have all the PCA features ready for the following model building step. Before we move one, let's see if we can find any noticeable patterns that can separate target classes with the new PCA features. Let's take a look at the following code first:

```
double[][] first2Components = transformed.Select(x => x.Where((y, i) => i <
2).ToArray()).ToArray();
ScatterplotBox.Show("Component #1 vs. Component #2", first2Components, labels);

double[][] next2Components = transformed.Select(
    x => x.Where((y, i) => i >= 1 && i <= 2).ToArray()
).ToArray();
ScatterplotBox.Show("Component #2 vs. Component #3", next2Components, labels);

next2Components = transformed.Select(
    x => x.Where((y, i) => i >= 2 && i <= 3).ToArray()
).ToArray();
ScatterplotBox.Show("Component #3 vs. Component #4", next2Components, labels);

next2Components = transformed.Select(
    x => x.Where((y, i) => i >= 3 && i <= 4).ToArray()
).ToArray();
ScatterplotBox.Show("Component #4 vs. Component #5", next2Components, labels);
```

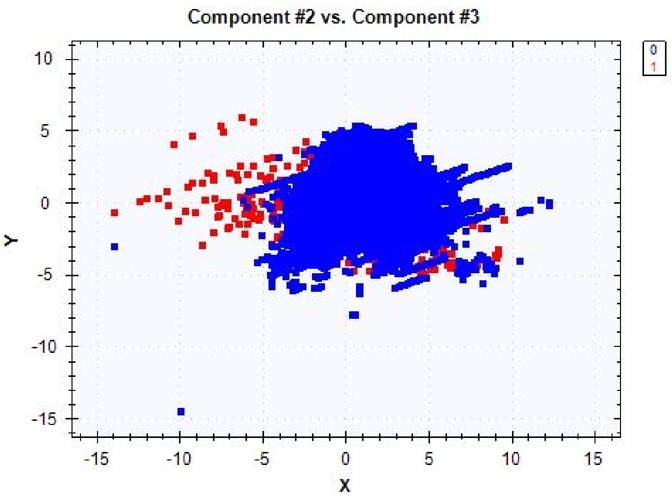
Similar to what we did in the data analysis step, we are taking two features and creating scatter plots of target classes across the selected features. From these plots, we can see if the principal components in the PCA-transformed data more effectively separate fraudulent credit card transactions from non-fraudulent transactions.

The following scatter plot applies between the first and second principal components:



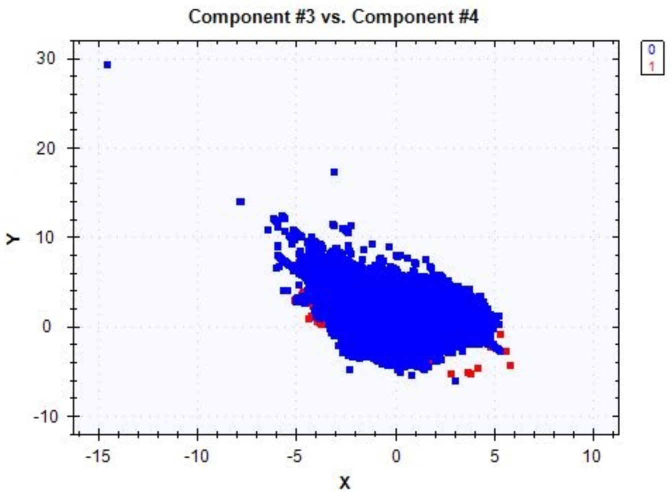
There is a noticeable cutoff point that separates frauds (red points in the scatter plot) from non-frauds (blue points in the scatter plot). From this scatter plot, it seems fraudulent samples typically have Y-values (the second principal component values) of less than -5.

The following is a scatter plot between the second and third principal components:

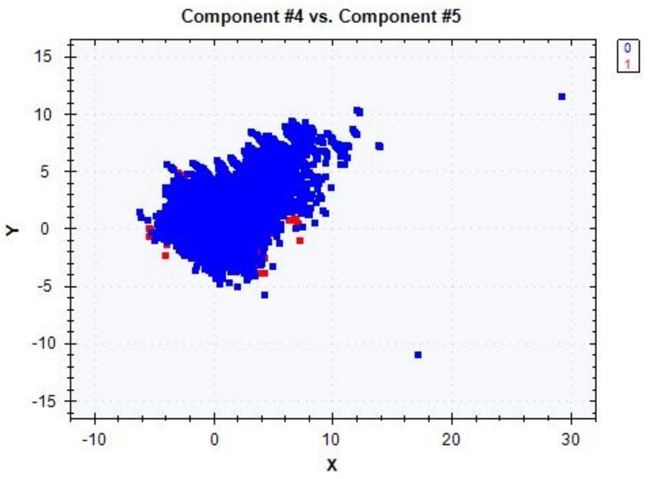


The pattern seems to be weaker in this plot, compared to the previous scatter plot, but there still seems to be a distinct line that separates many fraud cases from non-fraud cases.

The following scatter plot is between the third and fourth principal components:



And lastly, the following is a scatter plot between the fourth and fifth principal components:

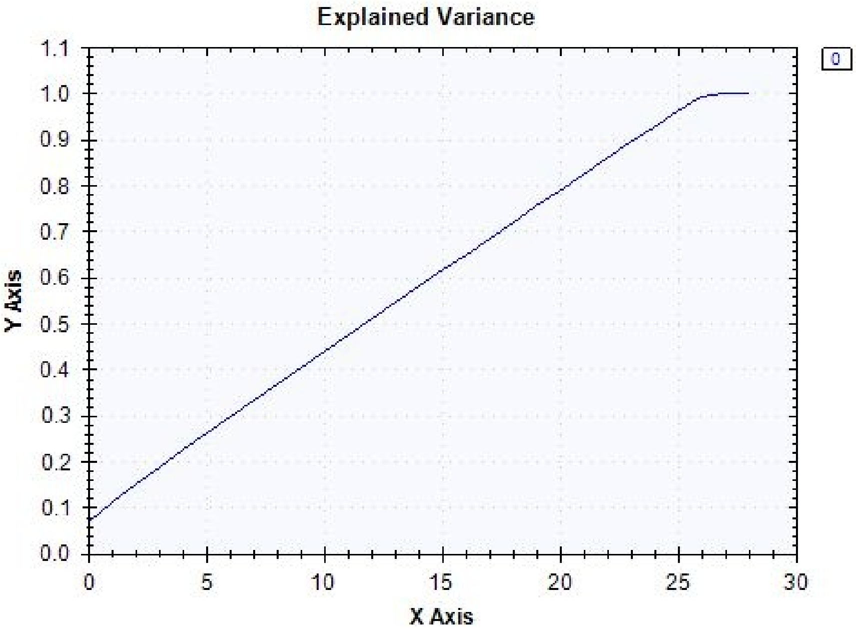


In the last two scatter plots, we cannot find a noticeable pattern to separate the two target classes from each other. Given that there were some separable lines we could find when we looked at the first three principal components and their scatter plots, our anomaly detection model for credit card fraud detection will be able to learn how to classify frauds, when it learns from this data in a higher dimension and multiple principal components.

Lastly, let's take a look at the proportion of variance explained by the principal components. Take a look at the following code first:

```
DataSeriesBox.Show(
    pca.Components.Select((x, i) => (double)i),
    pca.Components.Select(x => x.CumulativeProportion)
).SetTitle("Explained Variance");
System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "explained-variance.csv"),
    pca.Components.Select((x, i) => String.Format("{0},{1:0.0000}", i + 1,
x.CumulativeProportion))
);
```

As we discussed in `Chapter 9`, *Cyber Attack Detection*, we can use the `Components` property within a `PrincipalComponentAnalysis` object to extract the cumulative proportion of variance explained by each component. As you can see from the third line in the code, we iterate through the `Components` property and extract `CumulativeProportion` values. Then, we display a line chart by using the `DataSeriesBox` class. When you run this code, you will see the following chart for the cumulative proportion of variance explained by the principal components:



As you can see from this chart, by the twentieth principal component, about 80% of the variance in the data, is explained. We will use this chart to make a decision on how many principal components to use when we build an anomaly detection model in the following section.

Lastly, we need to export this data, as we just created a newly PCA-transformed dataset in this feature engineering step and we want to use this new data to build models. You can export this data by using the following code:

```
Console.WriteLine("exporting train set...");

System.IO.File.WriteAllLines(
    Path.Combine(dataDirPath, "pca-features.csv"),
    transformed.Select((x, i) => String.Format("{0},{1}", String.Join(",", x),
labels[i]))
);
```

As you can see from this code snippet, we are exporting this data into a CSV file named `pca-features.csv`. We will use this data to build anomaly detection models for credit card fraud detection in the following step.

The full code that was used in this feature engineering step can be found at the following link: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.10/FeatureEngineering.cs.

One-class SVM versus PCC

We are now ready to build anomaly detection models for the credit card fraud detection project. In this step, we are going to experiment with two different approaches. We are going to build a PCC, similarly to what we did in `Chapter 9`, *Cyber Attack Detection*. Also, we are going to introduce a new learning algorithm, the one-class SVM, which learns from normal credit card transaction data and decides whether a new data point is similar to the normal data that it was trained with.

Preparation for model training

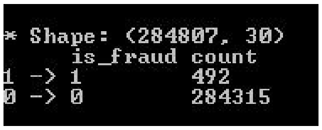
First, we need to load the data that we created in the previous feature-engineering step. You can use the following code to load the data:

```
// Read in the Credit Card Fraud dataset
// TODO: change the path to point to your data directory
string dataDirPath = @"<path-to-dir>";

// Load the data into a data frame
string dataPath = Path.Combine(dataDirPath, "pca-features.csv");
Console.WriteLine("Loading {0}\n\n", dataPath);
var featuresDF = Frame.ReadCsv(
    dataPath,
    hasHeaders: false,
    inferTypes: true
);
featuresDF.RenameColumns(
    featuresDF.ColumnKeys
        .Select((x, i) => i == featuresDF.ColumnCount - 1 ? "is_fraud" :
String.Format("component-{0}", i + 1))
);
```

If you recall from the previous feature-engineering step, we did not export the data with the column names. So, we are loading the data into a Deedle data frame, `featuresDF`, with the `hasHeaders` flag set to `false`. Then, we give the proper column names for each feature by using the `RenameColumns` method. Let's quickly check the target class distributions within this dataset, using the following code:

```
Console.WriteLine("* Shape: ({0}, {1})", featuresDF.RowCount, featuresDF.ColumnCount);

var count = featuresDF.AggregateRowsBy<string, int>(
    new string[] { "is_fraud" },
    new string[] { "component-1" },
    x => x.ValueCount
).SortRows("component-1");
count.RenameColumns(new string[] { "is_fraud", "count" });
count.Print();
```

The output of this code looks as follows:



```
* Shape: (284807, 30)
     is_fraud count
1 -> 1        492
0 -> 0        284315
```

As seen previously, in the data analysis step, the majority of the samples belong to non-fraudulent transactions and only a small portion of the data is fraudulent credit card transactions.

Principal component classifier

We will first try to build an anomaly detection model using principal components, similar to what we did in Chapter 9, *Cyber Attack Detection*. For training and testing a PCC model, we wrote a helper function, named `BuildPCAClassifier`. The detailed code for this helper function can be found at the following repo: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.10/Modeling.cs. Let's take a look at this helper function step by step.

You will see the following lines of code when you look at the code for the `BuildPCAClassifier` method:

```
// First 13 components explain about 50% of the variance
int numComponents = 13;
string[] cols = featuresDF.ColumnKeys.Where((x, i) => i < numComponents).ToArray();

// First, compute distances from the center/mean among normal events
var normalDF = featuresDF.Rows[
    featuresDF["is_fraud"].Where(x => x.Value == 0).Keys
].Columns[cols];

double[][] normalData = BuildJaggedArray(
    normalDF.ToArray2D<double>(), normalDF.RowCount, cols.Length
);
```

First, we are sub-selecting the first thirteen principal components that explain about 50% of the variance. Then, we create a non-fraudulent credit card transaction group, `normalDF` and `normalData`, so that we can use this subset to build an anomaly detection model.

The next thing we do is start computing the **Mahalanobis distance** metric to measure the distance between a data point and the distribution of the non-fraudulent credit card transactions. If you recall, we used the same distance metric in Chapter 9, *Cyber Attack Detection*, and we recommend you review the *Model building* section in Chapter 9, *Cyber Attack Detection*, for a more detailed explanation about this distance metric. The code to compute the distances looks like the following:

```
double[] normalVariances = ComputeVariances(normalData);
double[] rawDistances = ComputeDistances(normalData, normalVariances);

double[] distances = rawDistances.ToArray();
```

```
double meanDistance = distances.Average();
double stdDistance = Math.Sqrt(
    distances
    .Select(x => Math.Pow(x - meanDistance, 2))
    .Sum() / distances.Length
);

Console.WriteLine(
    "* Normal - mean: {0:0.0000}, std: {1:0.0000}",
    meanDistance, stdDistance
);
```

As you can see from this code snippet, we are using two helper functions, `ComputeVariances` and `ComputeDistances`, to compute the variances of feature values and the distances. The following is the code for the `ComputeVariances` method:

```
private static double[] ComputeVariances(double[][] data)
{
    double[] componentVariances = new double[data[0].Length];

    for (int j = 0; j < data[0].Length; j++)
    {
        componentVariances[j] = data
            .Select((x, i) => Math.Pow(data[i][j], 2))
            .Sum() / data.Length;
    }

    return componentVariances;
}
```

This code should look familiar, as this is the same code we used in `Chapter 9`, *Cyber Attack Detection*, to build a PCC model for cyber attack detection. Additionally, the following is the code for the `ComputeDistances` method:

```
private static double[] ComputeDistances(double[][] data, double[] componentVariances)
{

    double[] distances = data.Select(
        (row, i) => Math.Sqrt(
            row.Select(
                (x, j) => Math.Pow(x, 2) / componentVariances[j]
            ).Sum()
        )
    ).ToArray();

    return distances;
}
```

This code should also look familiar, as we used this same code in `Chapter 9`, *Cyber Attack Detection* as well. Using these two methods, we computed the mean and the standard deviation of the distance measures within the non-fraudulent transaction data. The output looks like the following:

```
* Normal - mean: 3.2319, std: 1.5984
```

With the distance measures within the normal transaction group computed, we now compute the distances between the fraudulent transactions and the distribution of non-fraudulent transactions. The following is the part of the `BuildPCAClassifier` code that computes the distances for frauds:

```
// Detection
var fraudDF = featuresDF.Rows[
    featuresDF["is_fraud"].Where(x => x.Value > 0).Keys
].Columns[cols];

double[][] fraudData = BuildJaggedArray(
    fraudDF.ToArray2D<double>(), fraudDF.RowCount, cols.Length
);
double[] fraudDistances = ComputeDistances(fraudData, normalVariances);
```

As you can see from this code snippet, we first separate the fraud data from the whole dataset and create a two-dimensional array variable, `fraudData`, which we use for distance-measuring calculations. Then, using the `ComputeDistances` function that we wrote, we can compute the distances between the fraudulent credit card transactions and the distribution of non-fraudulent transactions. With these distances measures, we then start analyzing the fraud detection rates for each of the target false-alarm rates. Take a look at the following code snippet:

```
// 5-10% false alarm rate
for (int i = 0; i < 4; i++)
{
    double targetFalseAlarmRate = 0.05 * (i + 1);
    double threshold = Accord.Statistics.Measures.Quantile(
        distances,
        1 - targetFalseAlarmRate
    );

    int[] detected = fraudDistances.Select(x => x > threshold ? 1 : 0).ToArray();

    Console.WriteLine("\n\n---- {0:0.0}% False Alarm Rate ----", targetFalseAlarmRate *
100.0);
    double overallRecall = (double)detected.Sum() / detected.Length;
    Console.WriteLine("* Overall Fraud Detection: {0:0.00}%", overallRecall * 100.0);
}
```

This code snippet should look familiar, as this is similar to what we did in Chapter 9, *Cyber Attack Detection*. One thing that is different here, however, is the fact that we only have two target classes (fraud versus non-fraud), whereas we had five target classes (normal versus four different types of cyber attack) in Chapter 9, *Cyber Attack Detection*. As you can see from this code, we experiment with five different target false alarm rates from 5% to 10%, and analyze the fraud detection rates for the given target false alarm rate. We will take a deeper look at this code in the following model evaluation step.

One-class SVM

The next approach we are going to explore for credit card fraud detection is training a one-class SVM. A one-class SVM is a special case of a SVM, where an SVM model is first trained with a data and then, when it sees a new data point, the SVM model can determine if the new data point is close enough to the data that it was trained with. For training a one-class SVM model, we wrote a helper function, `BuildOneClassSVM`, and the full code for this function can be found at the following repo: https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.10/Modeling.cs. Let's go through this helper function step by step.

First, let's look at the part of the code that sub-selects non-fraudulent credit card transaction data that will be used to train the one-class SVM. The code looks like the following:

```
// First 13 components explain about 50% of the variance
int numComponents = 13;
string[] cols = featuresDF.ColumnKeys.Where((x, i) => i < numComponents).ToArray();

var rnd = new Random(1);
int[] trainIdx = featuresDF["is_fraud"]
    .Where(x => x.Value == 0)
    .Keys
    .OrderBy(x => rnd.Next())
    .Take(15000)
    .ToArray();
var normalDF = featuresDF.Rows[
    trainIdx
].Columns[cols];

double[][] normalData = BuildJaggedArray(
    normalDF.ToArray2D<double>(), normalDF.RowCount, cols.Length
);
```

Similar to the previous PCC model that we built, we are using the first thirteen principal components that explain about 50% of the total variance. Next, we are going to sub-select records from the non-fraudulent transaction samples and build a train set. As you can see from this code, we are randomly selecting 15,000 non-fraudulent samples as a train set.

Now that we have a train set to train a one-class SVM model with, let's take a look at the following code:

```
var teacher = new OneclassSupportVectorLearning<Gaussian>();
var model = teacher.Learn(normalData);
```

We are using the `OneclassSupportVectorLearning` algorithm in the Accord.NET framework to train a one-class SVM model. As you can see, we built an SVM model with the `Gaussian` kernel in this chapter, but you can experiment with different kernels. Now, the only step left is to test this one-class SVM model that we just trained. The following code shows how we built a test set to evaluate this model:

```
int[] testIdx = featuresDF["is_fraud"]
    .Where(x => x.Value > 0)
    .Keys
    .Concat(
        featuresDF["is_fraud"]
        .Where(x => x.Value == 0 && !trainIdx.Contains(x.Key))
        .Keys
        .OrderBy(x => rnd.Next())
        .Take(5000)
        .ToArray()
    ).ToArray();

var fraudDF = featuresDF.Rows[
    testIdx
].Columns[cols];

double[][] fraudData = BuildJaggedArray(
    fraudDF.ToArray2D<double>(), fraudDF.RowCount, cols.Length
);

int[] fraudLabels = featuresDF.Rows[
    testIdx
].GetColumn<int>("is_fraud").ValuesAll.ToArray();
```

As you can see from this code, we are taking all the fraud samples and 5,000 randomly sub-selected non-fraud samples as a test set. With this test set, we are going to evaluate how well this one-class SVM model performs at detecting credit card frauds.

We are going to look closer into the evaluation code in the following section, but let's take a quick look at how we can evaluate the performance of the one-class SVM model that we just trained. The code looks like the following:

```
for(int j = 0; j <= 10; j++)
{
    model.Threshold = -1 + j/10.0;

    int[] detected = new int[fraudData.Length];
    double[] probs = new double[fraudData.Length];
    for (int i = 0; i < fraudData.Length; i++)
    {
        bool isNormal = model.Decide(fraudData[i]);
        detected[i] = isNormal ? 0 : 1;
    }

    Console.WriteLine("\n\n---- One-Class SVM Results ----");
    Console.WriteLine("* Threshold: {0:0.00000}", model.Threshold);
    double correctPreds = fraudLabels
        .Select((x, i) => detected[i] == 1 && x == 1 ? 1 : 0)
        .Sum();
    double precision = correctPreds / detected.Sum();
    double overallRecall = correctPreds / fraudLabels.Sum();
    Console.WriteLine("* Overall Fraud Detection: {0:0.00}%", overallRecall * 100.0);
    Console.WriteLine("* False Alarm Rate: {0:0.00}%", (1 - precision) * 100.0);
}
```

As you can see from this code, we iterate through different values of thresholds, similar to how we set different thresholds for the previous PCC model. As in the third line of the code, you can use the `Threshold` property of the model to get or set the threshold that determines whether a record is normal or abnormal. Similar to how we evaluate the PCC, we are going to look at the fraud detection rate and the false-alarm rate for model validations.

The full code we used in the model building step can be found at the following link: https://github.com/yoonhwang/c-sharp-machine-learning/edit/master/ch.10/Modeling.cs.

Evaluating anomaly detection models

We have trained two anomaly detection models—one using principal components and another using a one-class SVM algorithm. In this section, we are going to take a closer look at the performance metrics and the codes used to evaluate these models.

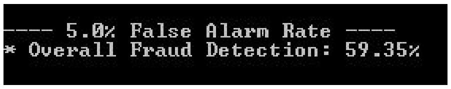
Principal Component Classifier

As briefly mentioned in the previous section, we are going to look at the credit card fraud detection rates for each of the target false alarm rates. The code for evaluating the PCC model looks like the following:

```
// 5-10% false alarm rate
for (int i = 0; i < 4; i++)
{
    double targetFalseAlarmRate = 0.05 * (i + 1);
    double threshold = Accord.Statistics.Measures.Quantile(
        distances,
        1 - targetFalseAlarmRate
    );

    int[] detected = fraudDistances.Select(x => x > threshold ? 1 : 0).ToArray();

    Console.WriteLine("\n\n---- {0:0.0}% False Alarm Rate ----", targetFalseAlarmRate *
100.0);
    double overallRecall = (double)detected.Sum() / detected.Length;
    Console.WriteLine("* Overall Fraud Detection: {0:0.00}%", overallRecall * 100.0);
}
```

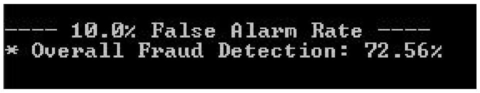
Similar to `Chapter 9`, *Cyber Attack Detection*, we iterate through the target false alarm rates from 5% to 10% and inspect the detection rates for the given false alarm rates. Using the target false alarm rate variable, `targetFalseAlarmRate`, we compute the threshold using the `Accord.Statistics.Measures.Quantile` method. With this calculated threshold, we flag all records with distances greater than this threshold as fraud, and others as non-fraud. Let's look at the evaluation results.

The following is the fraud detection rate at the 5% false alarm rate:



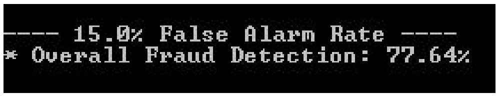
```
---- 5.0% False Alarm Rate ----
* Overall Fraud Detection: 59.35%
```

The following is the fraud detection rate at the 10% false alarm rate:



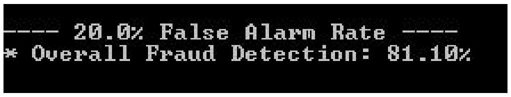
```
---- 10.0% False Alarm Rate ----
* Overall Fraud Detection: 72.56%
```

The following is the fraud detection rate at 15% false alarm rate:



```
---- 15.0% False Alarm Rate ----
* Overall Fraud Detection: 77.64%
```

Lastly, the following is the fraud detection rate at 20% false alarm rate:



```
---- 20.0% False Alarm Rate ----
* Overall Fraud Detection: 81.10%
```

As you can see from these results, as we relax and increase the target false alarm rate, the fraud detection rate improves. At the 5% target false alarm rate, we could only detect about 59% of the fraudulent transactions. However, at the 20% target false alarm rate, we can detect over 80% of the fraudulent credit card transactions.

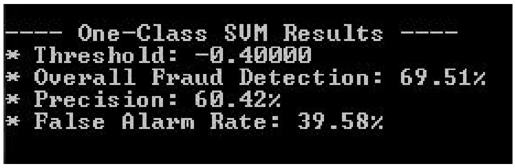
One-class SVM

Let's now take a look at how the one-class SVM model performed on the credit card fraud dataset. The code for the model evaluation looks like the following:

```
for(int j = 0; j <= 10; j++)
{
model.Threshold = -1 + j/10.0;

int[] detected = new int[fraudData.Length];
double[] probs = new double[fraudData.Length];
for (int i = 0; i < fraudData.Length; i++)
{
bool isNormal = model.Decide(fraudData[i]);
detected[i] = isNormal ? 0 : 1;
}

Console.WriteLine("\n\n---- One-Class SVM Results ----");
Console.WriteLine("* Threshold: {0:0.00000}", model.Threshold);
double correctPreds = fraudLabels
.Select((x, i) => detected[i] == 1 && x == 1 ? 1 : 0)
.Sum();
double precision = correctPreds / detected.Sum();
double overallRecall = correctPreds / fraudLabels.Sum();
Console.WriteLine("* Overall Fraud Detection: {0:0.00}%", overallRecall *
100.0);
Console.WriteLine("* False Alarm Rate: {0:0.00}%", (1 - precision) * 100.0);
}
```

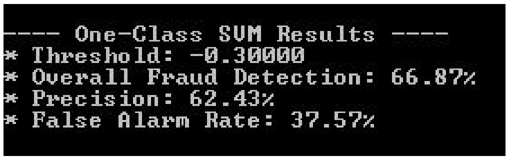
As you can see from this code, we iterate through different thresholds from -1.0 to 0.0, in increments of 0.1. You can set the threshold for the model by updating the `Threshold` property of the one-class SVM model object. This threshold will instruct the model on how to determine which record is fraudulent and which is not. When making a decision on the final model, you will need to experiment with different values for thresholds to settle on the best threshold that fits your requirements. Let's take a look at some of the performance results.

The following shows the performance metrics for the threshold at -0.4:



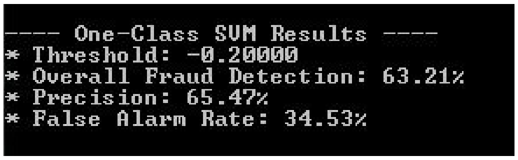
```
---- One-Class SVM Results ----
* Threshold: -0.40000
* Overall Fraud Detection: 69.51%
* Precision: 60.42%
* False Alarm Rate: 39.58%
```

The following shows the performance metrics for the threshold at -0.3:



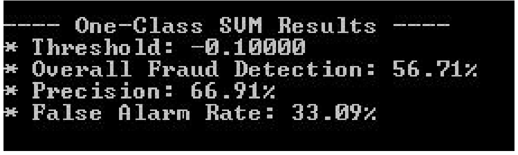
```
---- One-Class SVM Results ----
* Threshold: -0.30000
* Overall Fraud Detection: 66.87%
* Precision: 62.43%
* False Alarm Rate: 37.57%
```

The following shows the performance metrics for the threshold at -0.2:



```
---- One-Class SVM Results ----
* Threshold: -0.20000
* Overall Fraud Detection: 63.21%
* Precision: 65.47%
* False Alarm Rate: 34.53%
```

Lastly, the following shows the performance metrics for the threshold at -0.1:



```
---- One-Class SVM Results ----
* Threshold: -0.10000
* Overall Fraud Detection: 56.71%
* Precision: 66.91%
* False Alarm Rate: 33.09%
```

As you can see from these results, as we increase the threshold, the false alarm rate decreases, but the fraud detection rate decreases as well. It is clear that there is a trade-off between higher precision and a higher fraud detection rate. At a threshold of -0.4, the model was able to detect about 70% of the fraudulent credit card transactions with a roughly 40% false alarm rate. On the other hand, at a threshold of -0.1, the model could only detect about 57% of the fraudulent credit card transactions, but the false alarm rate was only about 33%.

Summary

In this chapter, we built another anomaly detection model for credit card fraud detection. We started this chapter by looking at the structure of the anonymized credit card fraud data, and then started analyzing the distributions of the target and feature variables. While we were analyzing the distribution of the target classes, we noticed that there was a large class imbalance between the fraud and non-fraud classes. This is normal when we face any kind of anomaly detection project, where the normal class outweighs by far the positive class. Then, we started analyzing the distributions of the anonymized features. Due to the fact that the features were anonymized for confidentiality issues, we could not arrive at any intuitions from the dataset.

However, we were able to understand the distributions better and how we cannot easily separate frauds from non-frauds using raw features. We then applied PCA and exported the PCA features for the model building step. We experimented with two approaches to building a credit card fraud detection model—the Principal Component Classifier and the one-class SVM. We evaluated the performances of these models by looking at the fraud detection rates at various false alarm rates. It was clear that there are trade-offs between improving the false alarm rates and improving the fraud detection rates.

This chapter was the last chapter about building ML models in C#. In the next chapter, we are going to summarize what we have done so far throughout all the chapters, and what additional real-life challenges there are when building ML models. Also, we are going to discuss some other software packages, as well as some other data-science technologies out there, that can be used for your future ML projects.

What's Next?

We have come a long way. From the basics and steps for building **machine learning** (**ML**) models to actually developing numerous ML models for various real-world projects, we have covered a lot so far. After a brief introductory chapter, where we learned the basics of ML and the essential steps that go into building ML models, we started building ML models. In `Chapter 2`, *Spam Email Filtering* and `Chapter 3`, *Twitter Sentiment Analysis*, we discussed building classification models using text datasets. In `Chapter 4`, *Foreign Exchange Rate Forecast* and `Chapter 5`, *Fair Value of House and Property*, we used financial and real estate property data to build regression models. Then in `Chapter 6`, *Customer Segmentation*, we covered how to use clustering algorithms to draw intuitive insights into customer behavior using the e-commerce dataset. In `Chapter 7`, *Music Genre Recommendation* and `Chapter 8`, *Handwritten Digit Recognition*, we expanded our knowledge of building ML models to build music recommendation and image recognition models using music records and handwritten digit image data. In `Chapter 9`, *Cyber Attack Detection* and `Chapter 10`, *Credit Card Fraud Detection* we built anomaly detection models for cyber attack detection and credit card fraud detection.

In this chapter, we are going to review the types of ML model we have built, the projects we have worked on so far, and code snippets for training various ML models using the Accod.NET framework. We will also discuss some of the challenges when using and applying ML in real-life projects and situations. Lastly, we are going to cover some of the other software packages that can be used for future ML projects, as well as other common technologies that are frequently used by data scientists.

In this chapter, we will cover the following topics:

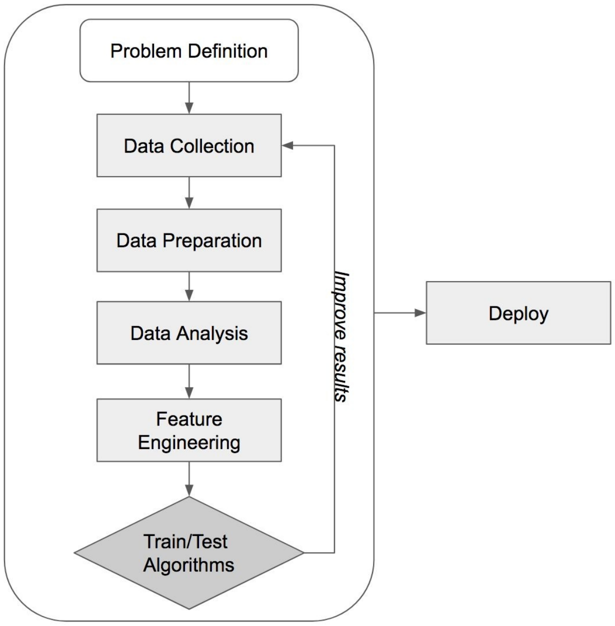
- A review of what we have learned so far
- Real-life challenges in building ML models
- Other common technologies used by data scientists

Review

From the first chapter onward, we have discussed and covered a large amount of material. From discussing the basics of ML to building classification, regression, and clustering models, it is worth reviewing what we have done so far before we end this book. Let's review some of the essential concepts and code that will be helpful for your future C# ML projects.

Steps for building ML models

As discussed in `Chapter 1`, *Basics of Machine Learning Modeling*, it can be challenging for aspiring data scientists and ML engineers to understand the flow and approaches to building real-world ML models that will be used in production systems. We have discussed the steps for building machine learning models in detail in `Chapter 1`, *Basics of Machine Learning Modeling*, and we have followed those steps in each of the projects that we have worked on so far. The following diagram should be a good recap of the essential steps in building real-world ML models:



As you should already, we always start a ML project with the problem definition. In this step, we define the problems that we are going to solve with ML and why we need ML models to solve such problems. This is also the step where we brainstorm our ideas and the prerequisites, such as the types of data required, as well as the types of learning algorithms that we are going to experiment with. Lastly, this is where we need to clearly define the success criteria for the project. We can define some evaluation metrics not only for the prediction performance of ML models, but also the execution performance of your models, especially if the models need to be run in a real-time system, and output the prediction results within a given time window.

From the problem definition phase, we move on to the data collection step. For those projects that we have worked on in this book, we used publicly available data that was already compiled and labeled. However, in real-world situations, data might not be available to start with. In this case, we will have to come up with approaches to collect the data. For example, if we are planning to build ML models for user behavior predictions for users on our website or application, then we can collect user activities on the website or application. On the other hand, if we are building a credit model to score the credit worthiness of potential borrowers, most likely we will not be able to collect data ourselves. In this case, we will have to resort to third-party data vendors who sell credit-related data.

Once we have gathered all of our data, the next thing we will have to do is prepare and analyze the data. During the data preparation step, we will need to validate the dataset by looking at the formats of the data fields, the existence of duplicate records, or the number of missing values. With these criteria checked, we can then start analyzing the data to see if there is any noticeable pattern in the dataset. If you recall, we typically analyzed the target variable distribution first and then we started analyzing the distributions of the features for each of the target classes to identify any noticeable patterns that could separate the target classes from each other. During the data analysis step, we focused on gaining some insights into the patterns in the data, as well as the structure of the data itself.

With insight and understanding of the data from the data analysis step, we can then start building features that will be used for our ML models. As Andrew Ng mentioned, applied ML is basically feature engineering. This is one of the most critical steps in building ML models and in determining the performance of our prediction models. If you recall, we discussed how to use one-hot encoding to transform text features into an encoded matrix of 1s and 0s for our text classification problems. We also discussed building time series features, such as moving averages and Bollinger Bands and using log transformations for highly skewed features, when we were building regression models. This feature engineering step is where we need to be creative.

Once we have all the features ready, we can then move on to training and testing various learning algorithms. Depending on whether the target variable is continuous or categorical, we can decide whether to build a classification model or regression model. If you recall from previous projects, we trained and tested our models by using k-fold cross-validation or by splitting the dataset into two subsets and training with one group and testing with another hold-out group. Until we find the model that we are satisfied with, we will have to repeat the the previous steps. If we do not have enough data, we will have to go back to the data collection phase and try to collect more data for more accurate models. If we handled duplicate records or missing values poorly, we will have to go back to the data preparation step to clean up the data. If we can build more and better features, then repeating the feature engineering step can help by improving the performance of our ML models.

The last step in building ML models is to deploy them to production systems. All the models should have been fully tested and validated by this point. It will be beneficial to have some monitoring tools in place before the deployment, so that the performance of the models can be monitored.

We have followed these steps quite thoroughly throughout the chapters, so you will realize how comfortable and familiar with these steps you are when you start working on your future ML projects. However, there are a couple of essential steps that we could not fully cover in this book, such as the data collection and model deployment steps, so you should always keep in mind the importance and goals of those steps.

Classification models

This first two ML models we built in `Chapter 2`, *Spam Email Filtering* and `Chapter 3`, *Twitter Sentiment Analysis*, were classification models. In `Chapter 2`, *Spam Email Filtering*, we built a classification model to classify emails into spam and ham (non-spam emails). In `Chapter 3`, *Twitter Sentiment Analysis*, we built a classification model for Twitter sentiment analysis, where the model classified each tweet into one of the three emotions—positive, negative, and neutral. Classification problems are common among ML projects. Building a model to predict whether a customer will buy an item in an online store is a classification problem. Building a model to predict whether a borrower will pay back his/her loan is also a classification problem.

If there are only two classes in the target variable, typically a positive outcome and a negative outcome, then we call it a binary classification. A good example of a binary classification problem is the spam email filtering project that we did in `Chapter 2`, *Spam Email Filtering*. If there are more than two classes in the target variable, then we call it a multi-class or multinomial classification. We had a case of having to classify a record into three different classes in the Twitter sentiment analysis project in `Chapter 3`, *Twitter Sentiment Analysis*; this was a good example of a multinomial classification problem. We had two more classification projects in this book. If you recall, we had eight different genres or classes in our target variable for the Music Genre Recommendation project in `Chapter 7`, *Music Genre Recommendation*, and we had 10 different digits in our target variable for the handwritten digit recognition project in `Chapter 8`, *Handwritten Digit Recognition*.

We experimented with numerous learning algorithms, such as logistic regression, Naive Bayes, **Support Vector Machine** (**SVM**), random forest, and neural network, for the aforementioned classification projects. To remind you how to train these learning algorithms in C#, we will reiterate how we initialized some of those learning algorithms in C# using the Accord.NET framework.

The following code snippet shows how we can train a binary logistic regression classifier:

```
var learner = new IterativeReweightedLeastSquares<LogisticRegression>()
{
    MaxIterations = 100
};
var model = learner.Learn(inputs, outputs);
```

For multinomial classification problems, we trained a logistic regression classifier using the following code:

```
var learner = new MultinomialLogisticLearning<GradientDescent>()
{
    MiniBatchSize = 500
};
var model = learner.Learn(inputs, outputs);
```

When building a Naive Bayes classifier, we used the following code:

```
var learner = new NaiveBayesLearning<NormalDistribution>();
var model = learner.Learn(inputs, outputs);
```

If you recall, we used `NormalDistribution` when the features had continuous variables, as in the case of the Music Genre Recommendation project, where all the features were audio spectrum features and had continuous values. One the other hand, we used `BernoulliDistribution`, where the features can only take binary values (0 versus 1). In the case of the Twitter sentiment analysis project in Chapter 3, *Twitter Sentiment Analysis*, all the features we had could only take 0s or 1s.

The following code shows how we could train a `RandomForestLearning` classifier:

```
var learner = new RandomForestLearning()
{
    NumberOfTrees = 100,

    CoverageRatio = 0.5,

    SampleRatio = 0.7

};
var model = learner.Learn(inputs, outputs);
```

As you might already be known, we could tune hyperparameters, such as the number of trees in the random forest (`NumberOfTrees`), the proportion of variables that can be used at maximum by each tree (`CoverageRatio`), and the proportion of samples used to train each of the trees (`SampleRatio`), to find better performing random forest models.

We used the following code to train a SVM model:

```
var learner = new SequentialMinimalOptimization<Gaussian>();
var model = learner.Learn(inputs, outputs);
```

If you recall, we could use different kernels for SVMs. On top of the `Gaussian` kernel, we could use `Linear` and `Polynomial` kernels as well. Depending on the type of dataset you have, one kernel works better than the others and various kernels should be tried to find the best performing SVM model.

Lastly, we could train a neural network using the following code:

```
var network = new ActivationNetwork(
    new BipolarSigmoidFunction(2),
    91,
    20,
    10
);

var teacher = new LevenbergMarquardtLearning(network);

Console.WriteLine("\n-- Training Neural Network");
int numEpoch = 10;
double error = Double.PositiveInfinity;
for (int i = 0; i < numEpoch; i++)
{
    error = teacher.RunEpoch(trainInput, outputs);
    Console.WriteLine("* Epoch {0} - error: {1:0.0000}", i + 1, error);
}
```

As you might recall from `Chapter 8`, *Handwritten Digit Recognition*, we trained a neural network model by running it through the dataset multiple times (epochs). After each iteration or epoch, we noticed the error rate decreased, as the neural network learned more and more from the dataset. We also noticed that in each epoch, the rate of improvements in the error rate was in diminishing return, so after enough epochs there would be no significant improvement in the performance of a neural network model.

You can view the code samples at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.11/ClassificationModelReview.cs`.

Regression models

We have also developed multiple regression ML models. In Chapter 4, *Foreign Exchange Rate Forecast*, we worked on the Foreign Exchange Rate Forecast project, where we built models that could predict future exchange rates between Euros and US dollars. In Chapter 5, *Fair Value of House and Property*, we trained different ML models that could predict house prices for the Fair Value of House and Property project. Regression problems are also common in real-world ML projects. Building a model that predicts the lifetime value of a customer is a regression problem. Building a model that predicts the maximum amount of money that a potential borrower can borrow without going bankrupt is another regression problem.

We have explored numerous machine learning algorithms for regression projects in this book. We have experimented with linear regression and linear SVM models in Chapter 4, *Foreign Exchange Rate Forecast* for the Foreign Exchange Rate Forecast project. We have also tried using different kernels, such as `Polynomial` and `Guassian` kernels, for SVM models in Chapter 5, *Fair Value of House and Property* for the Fair Value of House and Property project. To remind you how to train these regression models in C#, we will reiterate how we could use C# and the Accord.NET framework to build these models.

The following code snippet shows how we can train a linear regression model:

```
var learner = new OrdinaryLeastSquares()
{
    UseIntercept = true
};
var model = learner.Learn(inputs, outputs);
```

When building a SVM with the linear kernel, we used the following code:

```
var learner = new LinearRegressionNewtonMethod()
{
    Epsilon = 2.1,
    Tolerance = 1e-5,
    UseComplexityHeuristic = true
};
var model = learner.Learn(inputs, outputs);
```

As you might recall, `Epsilon`, `Tolerance`, and `UseComplexityHeuristic` are hyperparameters that can be tuned further for better model performance. When building a SVM model, we recommend you try various combinations of the hyperparameters to find the best performing model for your business case.

When we want to use a polynomial kernel for a SVM, we can use the following code:

```
var learner = new FanChenLinSupportVectorRegression<Polynomial>()
{
    Kernel = new Polynomial(3)
};
var model = learner.Learn(inputs, outputs);
```

For a `Polynomial` kernel, you can tune the degree of a polynomial function. For example, for a second degree polynomial (quadratic) kernel, you can initialize the kernel with `new Polynomial(2)`. Similarly, for a fourth degree polynomial kernel, you can initialize the kernel with `new Polynomial(4)`. However, increasing the complexity of a kernel can result in overfitting, so you will need to take care when using a high-degree polynomial kernel for a SVM.

When we want to build a SVM with a Gaussian kernel, we can use the following code:

```
var learner = new FanChenLinSupportVectorRegression<Gaussian>()
{
    Kernel = new Gaussian()
};
var model = learner.Learn(inputs, outputs);
```

You can find the code samples for the aforementioned regression models at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.11/RegressionModelReview.cs`.

Clustering algorithms

We discussed one unsupervised learning algorithm, k-means clustering, and how it can be used to draw insights from an unlabeled dataset. In `Chapter 6`, *Customer Segmentation*, we used the k-means clustering algorithm on an e-commerce dataset and we learned about different customer behaviors from the dataset. We have covered how to use clustering algorithms to build different customer segments, based on their purchase history, but there are many other applications of clustering algorithms. For example, clustering algorithms can also be used in image analysis, for example in partitioning images into sub-sections, and in bioinformatics, such as discovering groups of closely related genes (gene clustering).

We used the following code to build a k-means clustering algorithm using C# and the Accord.NET framework:

```
KMeans kmeans = new KMeans(numClusters);
KMeansClusterCollection clusters = kmeans.Learn(sampleSet);
```

As you might recall, we need to give the number of clusters we want to build to the `KMeans` class. One way to programmatically decide the best number of clusters that we discussed was to look at the Silhouette score, which measures how similar a data point is to its own cluster. Using this Silhouette score, you can iterate through different numbers for the number of clusters and then decide which one works the best for the given dataset.

You can find the code samples for the k-means clustering algorithm at the following link: `https://github.com/yoonhwang/c-sharp-machine-learning/blob/master/ch.11/ClusteringAlgorithmReview.cs`.

Real-life challenges

It would be great if we could just build ML models for all of our business problems. However, that is normally not the case. Often, there are more challenges in getting to the model development phase than in actually building working models. We will discuss the following frequently appearing data science challenges when we are working on ML projects:

- Data issues
- Infrastructural issues
- Explainability versus accuracy

Data issues

Having the right data and enough data is the most important prerequisite for building a working ML model. However, often, this is the most difficult part in developing ML models for a few different reasons. We will discuss a few common challenges that many data scientists face in terms of issues related to data.

First, the data needed might simply not exist. For example, think of a recently formed online retail store wanting to apply ML to understand or predict their customers' spending patterns. Since they are a new business with a small customer base, with not much historical purchase data, they will not have enough data for data scientists to work with. In this case, all they can do is wait for a better time to embark on ML projects, even if they have data scientists on their team. Their data scientists will simply not be able to build anything meaningful with a limited amount of data.

Second, the dataset exists, but it is not accessible. This kind of problem happens often in big corporations. Due to security issues, accessing the data might have been restricted to certain subgroups of an organization. In this case, data scientists might have to go through multiple levels of approval from different departments or business entities or they might have to build a separate data pipeline, through which they can ingest the data that they need. This kind of issue typically means it takes a long time before data scientists can start working on the ML project that they wanted to work on.

Lastly, the data is segmented or too messy. Almost all of the time, the raw datasets that data scientists get include messy data and come from different data sources. There might be too many missing values or too many duplicate records in the data and data scientists will have to spend lots of time cleaning up the raw dataset. The data might be too unstructured. This typically happens when you work with text-heavy datasets. In this case, you might have to apply various text mining and **natural language processing** (**NLP**) techniques to clean up the data and make it usable for building ML models.

Infrastructure issues

Training a ML model on a large dataset requires a large amount of memory and CPU resources. As we get bigger and bigger data, it is inevitable that we run into infrastructural issues. If you do not have enough memory resources for training ML models, you might end up getting *Out of Memory* exceptions after many hours or days of training models. If you do not have enough processing power, then training a complex ML model can take weeks and even months. Getting the right amount of computational resources is a real challenge in building ML models. As the data that is being used for ML grows faster than ever, the amount of computational resources required also grows significantly year after year.

With the emerging popularity of cloud computing service providers, such as AWS, Google, and Microsoft Azure, it became easier to get the required computational resources. On any of those cloud computing platforms, you can easily request and use the amount of memory and CPUs that you need. However, as everything comes with a price, running ML jobs on those cloud platforms can cost lots of money. Depending on your budget, such costs can restrict how much computational resources you can use for your ML tasks, and it needs to be planned cleverly.

Explainability versus accuracy

The last common real-life challenge in ML is the trade-off between the explainability and accuracy of ML models. More traditional and linear models, such as logistic regression and linear regression models, are easy to explain in terms of the prediction output. We can extract the intercept and the coefficients of those linear models and we can get the prediction output using simple arithmetic operations. However, more complex models, such as random forest and SVM, are more difficult to use in terms of explaining the prediction output. Unlike logistic regression or linear regression models, we cannot deduce the prediction output from simple arithmetic operations. Those complex models work more like a black box. We know the input and the output, but what goes in between is a black box to us.

This kind of explainability issue among complex learning algorithms becomes a problem when users or auditors request explanations about the model behavior. If there is such a requirement for explainability, we will have to resort to more traditional linear models, even if more complex models perform better than those linear models.

Other common technologies

As the field of ML and data science is evolving faster than ever, the number of new technologies being built is also growing at a fast pace. There are many resources and tools that help in building ML solutions and applications more easily and quickly. We are going to discuss a few technologies and tools that we recommend you get acquainted with for your future ML projects.

Other ML libraries

The Accord.NET framework that we have used throughout this book is one of the most frequently used and well documented frameworks for ML. However, other libraries that are built for ML in C# are worth mentioning and taking a look at for your future ML projects.

Encog is a ML framework that can be used in Java and C#. It is very similar to the Accord.NET framework that we have been using, in the sense that is has a wide range of numerous ML algorithms available within the framework. This framework is well documented and has lots of sample code that can be referenced for your future machine learning projects. More information and documentation about the **Encog** framework can be found at the following link: `https://www.heatonresearch.com/encog/`.

Weka is another ML framework, but it is different from the Accord.NET framework in the sense that the **Weka** framework is specifically engineered for data mining. It is broadly used by many researchers and has good documentation and even a book that explains how to use **Weka** for data mining. Weka is written in Java, but it can also be used in C#. More information about the **Weka** framework can be found at the following link: `https://www.cs.waikato.ac.nz/~ml/index.html`. Also, information about how to use the **Weka** framework in C# can be found at the following link: `https://weka.wikispaces.com/Use%20WEKA%20with%20the%20Microsoft%20.NET%20Framework`.

Lastly, you can always search in NuGet, the package manager for .NET, for any other machine learning frameworks for C#. Any library or package that is available on NuGet can easily be downloaded and referenced in your development environment. It is a good practice to search the following link for any packages you might need or that might be helpful for your future machine learning projects: `https://www.nuget.org/`.

Data visualization libraries and tools

The next set of tools and packages that we are going to discuss is about data visualizations. ML and data visualization are an inseparable combination for data science. For any ML models that you build, you should be able to present your findings, model performance, and model results to users or business partners. Furthermore, for continuous model performance monitoring purposes, data visualization techniques are often used to identify any issues with the models in production systems or any potential deterioration in the model performance. As a result, many data visualization libraries were built to make data visualization tasks easier.

LiveCharts is a .NET library for data visualization. We have used the Accord.NET framework's charting libraries throughout this book, but for more complex plots, we recommend using **LiveCharts**. From basic charts, such as line and bar charts, to complex interactive charts, you can build various visualizations in C# relatively easily. The **LiveCharts** library has thorough documentation and lots of examples along with sample code. You can find more information about how to use **LiveCharts** for data visualizations at the following link: `https://lvcharts.net/`.

Aside from the C#.NET library for data visualization tasks, there are two more data visualization tools that are frequently used in the data science community: **D3.js** and **Tableau**. **D3.js** is a JavaScript library for building and presenting charts on web pages. Often, this JavaScript library is used to create a dashboard for various data science and data visualization tasks. **Tableau** is a business intelligence tool, with which you can drag and drop to create various visualizations. This tool is frequently used to create a dashboard not only by data scientists, but also by non-data professionals. For more information about the **D3.js** library, you can follow this link: `https://d3js.org/`. For more information about Tableau, you can follow this link: `https://www.tableau.com/`.

Technologies for data processing

Lastly, we are going to discuss some commonly used technologies and tools for processing data. Throughout this book, we have mostly used CSV files as input for our ML modeling projects. We have used the Deedle framework to load, manipulate, and aggregate the data. However, often, the type of input data for ML projects varies. For some projects, the data might be stored in SQL databases. For other projects, the data might be stored across distributed filesystems. Furthermore, the source of the input data can even be from real-time streaming services. We will briefly discuss a few commonly used technologies for such cases and where to look for more detailed information in order for you to do further research.

SQL databases, such as SQL Server or PostgreSQL, are the most commonly used technologies for data storage and data processing. Using the SQL language, data scientists can easily retrieve, manipulate, and aggregate data to process and prepare the data for their ML projects. As an aspiring data scientist, it will be beneficial for you to become more comfortable with using the SQL language for processing the data.

Another technology that is often used within the data science community is **Spark**, which is a cluster-computing framework. With **Spark**, you can process a large amount of data at scale. Using clusters of machines and distributing heavy computations across those machines, **Spark** helps in building scalable big data solutions. This technology is widely used among numerous organizations and companies, such as Netflix, Yahoo, and eBay, which have lots of data to process every day.

Lastly, there are numerous stream-processing technologies for real-time ML applications. One of the most popular ones is **Kafka**. This technology is often used when building real-time applications or data pipelines that need to continuously stream the data. In the case of building real-time ML applications, using a stream-processing technology, such as **Kafka**, will be essential for the successful delivery of a real-time ML product.

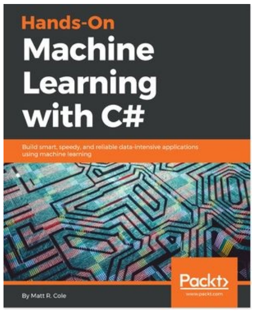
Summary

In this chapter, we reviewed what we have discussed so far in this book. We briefly went over the essential steps in building ML models. Then, we summarized and compiled the code to build various ML models in C# using the Accord.NET framework for classification, regression, and clustering problems. We have also discussed the real-life challenges that we could not cover in this book, but that you will most likely face when you start working on your future ML projects. We discussed challenges in accessing and compiling the data to build ML models, infrastructural challenges that will occur for big data, and the trade-offs between the explainability and accuracy of the ML models. Lastly, we covered some commonly used technologies that we recommend you get acquainted with for your future ML projects. The code libraries and tools that were mentioned in this chapter are only a subset of the tools that are available, and the commonly used tools and technologies are going to evolve year on year. We recommend you consistently research upcoming technologies for ML and data science.

We have covered various ML techniques, tools, and concepts throughout this book. As you have worked through this book from building basic classification and regression models to complex recommendation and image recognition systems, as well as anomaly detection models for real-world problems, I hope you have gained more confidence in building ML models for your future ML projects. I hope your journey throughout this book was worthwhile and meaningful, and that you have learned and gained many new and useful skills.

Other Books You May Enjoy

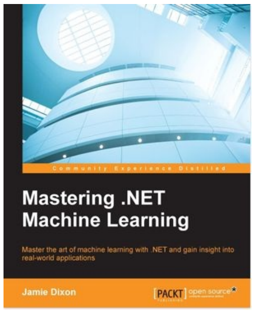
If you enjoyed this book, you may be interested in these other books by Packt:


Hands-On
Machine Learning with C#
Build smart, speedy, and reliable data-intensive applications using machine learning
Packt>
www.packt.com
By Matt R. Cole

Hands-On Machine Learning with C#
Matt R. Cole

ISBN: 9781788994941

- Learn to parameterize a probabilistic problem
- Use Naive Bayes to visually plot and analyze data
- Plot a text-based representation of a decision tree using nuML
- Use the Accord.NET machine learning framework for associative rule-based learning
- Develop machine learning algorithms utilizing fuzzy logic
- Explore support vector machines for image recognition
- Understand dynamic time warping for sequence recognition


Community Experience Distilled
Mastering .NET Machine Learning
Master the art of machine learning with .NET and gain insight into real-world applications
Jamie Dixon
PACKT open source

Mastering .NET Machine Learning
Jamie Dixon

ISBN: 9781785888403

- Write your own machine learning applications and experiments using the latest .NET framework, including .NET Core 1.0
- Set up your business application to start using machine learning.
- Accurately predict the future using regressions.
- Discover hidden patterns using decision trees.
- Acquire, prepare, and combine datasets to drive insights.
- Optimize business throughput using Bayes Classifier.
- Discover (more) hidden patterns using KNN and Naïve Bayes.
- Discover (even more) hidden patterns using K-Means and PCA.
- Use Neural Networks to improve business decision making while using the latest ASP.NET technologies.
- Explore Big Data, distributed computing, and how to deploy machine learning models to IoT devices—making machines self-learning and adapting
- Along the way, learn about Open Data, Bing maps, and MBrace

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!