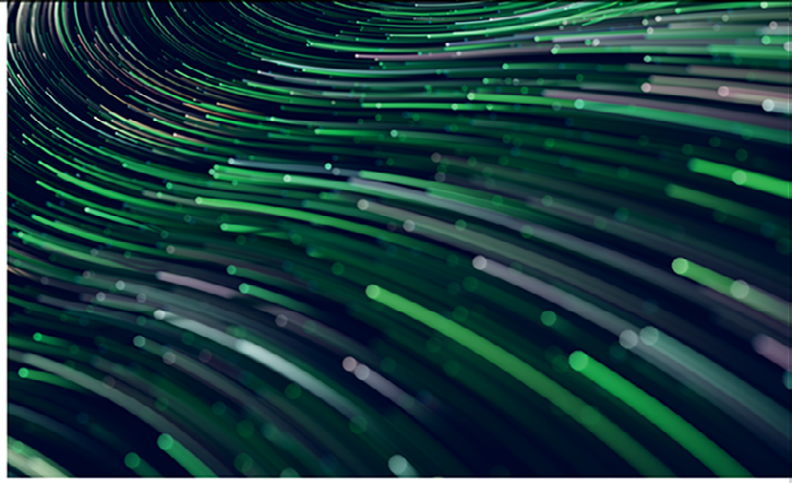
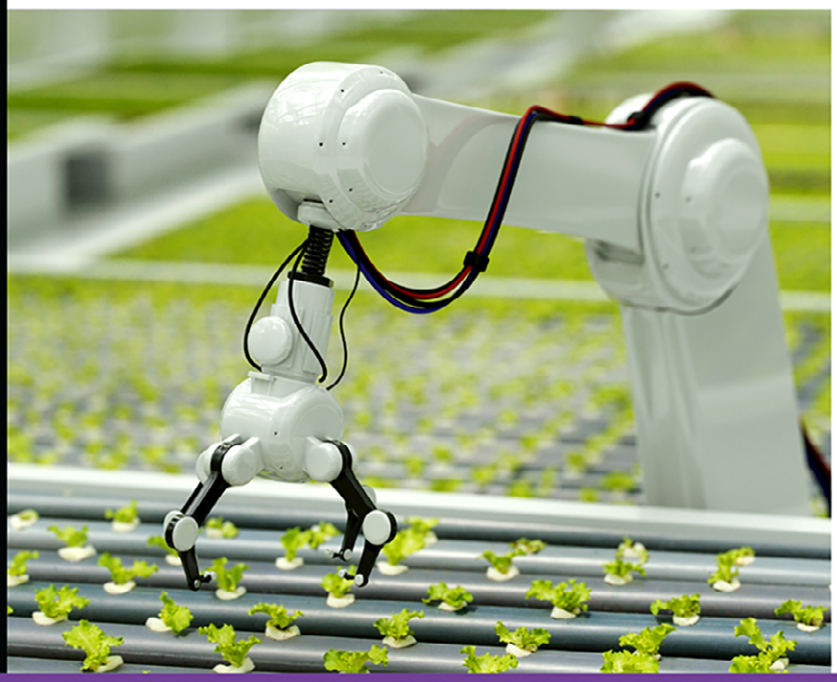
```
#create the month from the date
df_minus_3['month'] = pd. to_datetime
df_minus_3['invoicedate_minus_3']
).dt.month

#calculate number of days between the
df_minus_3['since_first_sale'] =
pd.to_datetime(df_minus_3['invoiceda
for_model['invoicedate']))

#convert calculation to a numeric value
df_minus_3['since_first_sale'] = df_min
```





Data Source
Exploration
Clustering
Algorithms,
Decision Tr
Random Fore
Monitoring,

Applied Machine Learning

Using Machine Learning to Solve Business Problems

Jason Hodson

Rheinwerk Computing

Contents

Chapter 5
Picking Your Model

At long last, you've gotten through much of the hard work required to build an effective machine learning model. Take a pause and give yourself some credit! In this chapter, we'll dive into the foundational algorithms of machine learning.

We'll now shift our focus to the models themselves and explore strategies you can use to pick them. This chapter focuses on select algorithms and models and is not intended to be comprehensive. Instead, we focus on the most commonly used algorithms to help you understand how they build upon each other. Although knowing all these algorithms is useful, it can be unnecessary noise when you're just getting started. Speaking from experience, we discussed upwards of 20 different algorithms during my master's program, and I've only used a small subset of these in practice.

This chapter begins with a discussion on the model selection approach and a framework you can use for it. We'll then discuss a selection of algorithms, starting with regression models and moving into the traditional tree-based models typically associated with machine learning. We'll also talk about clustering, the misfit of the machine learning world.

A word of warning: Don't skip Section 5.5.1 on decision trees! The decision tree algorithm is the foundation of random forest and Gradient Boosting Machine (GBM) models. The chapter is deliberately structured to introduce these important topics using simple decision trees before you progress to more complex models. While you're unlikely to use the decision tree algorithm in practice, understanding it is critical to learn the random forest and GBM algorithms that are more common on the job.

5.1 The Simpler the Model, the Better

The longer you work in the technical space, the more you'll come to understand that the simpler solution is usually the better solution. If you need to add an additional 1,000 lines of code to make your model 1% better, is it actually worth it? If you work on incredibly high-stakes use cases where 1% represents millions or billions of dollars, then maybe it is. However, in most enterprise analytics scenarios, the risk and time associated with the additional 1,000 lines of code isn't worth the 1% gain.

When Complicated Becomes a Problem

I had just started a new role, and one of my responsibilities was to take over an existing set of models. The underlying tech stack was not ideal and the complexity of the models created several challenges for me. I spent many hours digging through the code only to realize that the stakeholders weren't actually using the model.

As I worked with the previous owner of the models, it became clear how complex the modeling process was from the various data sources and how they were brought into the process to the actual model itself. This resulted in a situation where my team wasn't adding the right value for the organization. It also impacted the former owner of the model because he received a number of additional questions he otherwise wouldn't have. If nothing else, heed this warning for your own self-interest!

That said, I've haven't always created simple processes in my own work. When transitioning out of one of my roles, I had a number of technical processes to hand over to another member of the team. While the nature of the work was complicated, it was my own code, so I found it easy to understand. This meant I had minimal documentation and hadn't gone back to see where I could have simplified the code and the process.

The takeaway here should be to simplify your approach as much as possible. It'll help you and others support your stakeholders now and in the future.

Think about this from the stakeholder perspective as well. A simpler approach is easier for others to explain, but it's also easier to explain yourself. When you're in the depths of building out your process, you'll be at the height of your understanding of the code and all its small nuances. Once you've deployed your model into production, time will pass, and you'll become less familiar with those nuances. You may revisit your code after a few months in a panic thinking you missed something or you're doing something in error.

You'll also experience this when conducting a knowledge share or cross-training session for others who are new on your team or need to understand the model. A simple process and a simple model make the knowledge share easier on yourself and other members of your team. Part of maturing in your machine learning journey is understanding that scalability and longevity are important, so you'll sometimes need to sacrifice incremental increases in your model's performance to ensure its longevity.

It's helpful to think about this from a business perspective. If the incremental increase in accuracy is 0.25%, does the added complexity of a change justify this increase? Depending on your business case, sometimes that answer is yes, and sometimes it's no. However, in a situation where the change is a connection to two additional data sources, complicated functions, and an overall increase in your code base, it's unlikely to be a favorable tradeoff.

There's also a courtesy component to this as well. It's unlikely you'll be the one who creates a model and owns it for its entire life, so thinking ahead is important! The ease of taking over a model varies significantly depending on how it was structured.

From a purely technical perspective, the more complex your model, the longer it takes to train and execute. Say you can train a simpler model in five minutes, but a more complex model trains in twenty minutes. This extra time slows down your ability to iterate on the model and add new features with additional value.

In practice, you'll likely need to be iterative about which data you bring into your model at which stage. Preparing your data for a model is the most time-consuming part of the modeling process, so you shouldn't attempt to bring in all the possible data at once. It's better to focus on a smaller subset of your data to improve your development speed.

Finally, consider how likely it is that your stakeholders will introduce additional complexity. If your data is generated by a human-dependent process, you'll probably have to account for nuances that make your model more complex.

5.2 Model Decision Framework

There are a number of resources available online that show you the full suite of options in the modeling space. You can find examples in the scikit-learn (sklearn) documentation at *https://scikit-learn.org/stable/machine_learning_map.html*.

Figure 5.1 shows the framework we'll use for this chapter, which consists of the general order of algorithms to consider as well as their associated complexity.

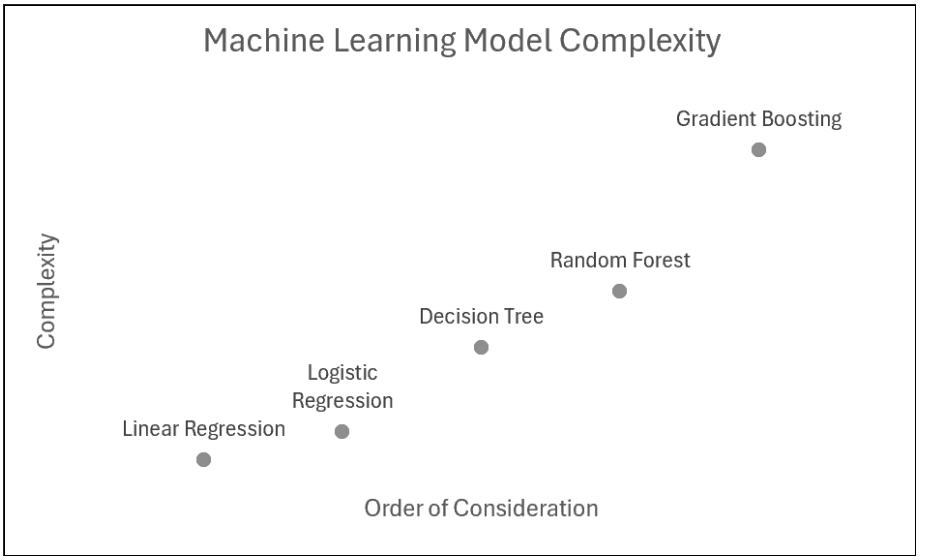


Figure 5.1 Model Complexity

You'll see that regression should usually be considered first, and it has relatively low complexity (more on this in Section 5.4). Then, we get into the tree-based models, which have an increasing level of complexity.

The trends and themes in this framework won't always hold true in every situation. The goal is to give you a general framework you can use to think about model selection, since it can be a confusing and intimidating task when you're getting started.

5.2.1 How Important Is Interpretability?

Interpretability is the ability to understand *why* your model is making the decisions it's making. We'll discuss this concept in more detail in Chapter 6, but for the purposes of picking our model, we need to take a moment to think about it. We've just established that regression is lower in complexity, but let's throw another curve ball into the mix. If interpretability isn't high on your priority list, you can skip regression. This is primarily due to the additional data cleaning and considerations required for regression, which are discussed in Section 5.4. The tradeoff in time it takes to train a regression model versus a decision tree is relatively minimal. Any minimal increase in time to train a decision tree offsets the added effort required to ensure your data is set up properly for a regression model.

The biggest drawback you'll see with tree-based models is that they're *black boxes* that are hard to interpret. However, you can use the techniques we'll discuss in Section 5.5 to interpret tree-based models, so this is by no means a binary decision where you're either getting a better model or a model with interpretability. However, if your stakeholder is hyperfocused on understanding the *why* behind the prediction, or the model is being used in a day-to-day operational setting, understanding how a model makes its predictions yields significant value. The coefficients generated by regression models provide a high degree of specificity in interpretation that is hard to replicate with a machine learning model.

Let's summarize our decision-making process:

- **If interpretability is very important**
  Stick with regression as the default and move to the next question.
- **If interpretability is not important**
  Skip the next question and adjust to tree-based models.

5.2.2 How Many Rows and Columns?

If you've identified that interpretability is important, the next step is identifying whether your data can support a regression use case. You should think about rows and columns together because of the limitations in how the backend math of regression works.

Here's a mental model for considering this is: The more rows you have, the more columns you can use. As the size of your data and number of records grows, it allows the math behind the model to look across more columns and find relationships between them. A dataset with only 100 rows and 30 columns won't work for regression.

As a rule of thumb, it's best to keep your number of columns at 30 or fewer if you have fewer than 100,000 rows of data to train your model on. If you have hundreds of thousands of rows, then 50 columns or fewer is generally acceptable. For larger datasets, the number of rows per column should be around 3,000–5,000 rows per column.

These guidelines allow the math behind the regression to operate correctly. They also ensure you're not breaking any of the assumptions in regression.

Keep Multicollinearity in Mind

As we'll discuss more in the next section, multicollinearity occurs when you have two columns that are highly correlated with each other. Regression assumes there is no multicollinearity in the dataset. As your column count grows, this becomes a more challenging dynamic to manage.

To summarize our rules:

- **If you have an appropriate column-to-row ratio**
  Stick with regression and move to the next question.
- **If you do not have an appropriate column-to-row ratio**
  Adjust to a tree-based model and move to the next question.

5.2.3 What Is Being Predicted?

At this point, you've identified the category of model you'll be using: either a regression approach or tree-based approach. Now, you'll need to understand what you're predicting and the associated next step based on the type of model you selected. The two categories of what we're predicting are called *regression* (if you're confused, keep reading) and *classification*.

Regression

Regression in this context translates to predicting a number. If you're predicting the number of sales, this is a regression or regressor prediction. Regression models naturally do this, and linear regression does this explicitly. When most people think about predictive models, they're likely thinking about a model that predicts a specific number.

Sometimes Regression Is Regression... Sometimes It's Not

I'm not entirely sure who thought it was a good idea to name the overall approach to predicting a number "regression" when this terminology is already reserved for linear and logistic regression, but it is what it is. This has confused me on a few occasions when onboarding onto a new project or team, so it's never a bad idea to clarify what someone means when they say "regression."

Classification

Classification is well-named. The objective of the classification model is to classify your data. In practice, it's still technically predicting a number. For example, if you're building a model to predict whether an employee will leave the company, your model will classify them either as someone who will stay or as someone who will leave. Someone who will leave is often coded in the data as a 1 and someone who will stay is coded as a 0.

Probabilities are at the core of classification. While it can depend on the use case, how valuable is it to provide a binary prediction? Psychologically, it creates a perception of confidence. However, using the employee turnover model example, what if your model predicts an employee will leave in the next six months, but they're still employed at the company on month seven?

Thinking probabilistically is often more valuable for stakeholders, but it also keeps your model from taking unnecessary heat for being wrong. All models are wrong—the good ones are just less wrong. As an alternative, what if your model predicted the probability someone would leave in the next six months? For a specific employee, the same binary prediction that they may leave could actually only be a 25% probability. Most decision-makers will interpret a 25% probability of an event occurring differently than just being told the event *will* occur.

So why the lecture about probabilistic thinking? Because all classification models start with probability and are converted into binary terms that minimize the frequency of false positives and false negatives. The difference in the code is relatively trivial, as the model is outputting both, so as always it goes back to the use case.

There is also a spectrum to consider as well. Going back to the employee turnover example, what if a probability of 15% is considered high in this business context? In scenarios where your target variable is considered imbalanced (one outcome you're predicting is more likely than another), it can be challenging for your stakeholders to understand the full context. For employee turnover, most companies will retain the majority of their employees in a six-month span rather than see them leave (I hope). This can lead to your model output recommending that the optimal binary cutoff point for turnover should be 15%. While the math may be optimal, a stakeholder is likely to question the value of your model's outputs. In this scenario, you can consider grouping your data together into logical categories. One approach could be to group anyone with a probability of

50% or greater as high risk, 15% to 49% as medium risk, and anything below that as low risk. While you've introduced subjectivity into the model's output, you've also met the stakeholder where they need to be to effectively consume your model's output.

Translating to Model Selection

Linear regression and logistic regression models have distinct use cases. If you're trying to predict a number, you'd use linear regression (should we call it "regression regression"?). If you're trying to classify data, you'd use logistic regression.

For tree-based models, the regression versus classification distinction is almost completely abstracted from our perspective. Each model has a regressor and classifier function that can be loaded in, and the inputs required are more or less the same (e.g., `DecisionTreeClassifier` and `DecisionTreeRegressor`). In practice, this is quite nice. It's easier to switch between approaches when you can just change the name of the function without having to change all your hyperparameters. However, it can muddy the waters from a learning perspective, because there isn't much of a distinction when you're applying it.

As we work through the use cases, we'll apply the various models to each one so you can see what we discussed in practice.

5.3 Train-Test Split

Regardless of what model you're using, you need to split your data into a test set and a train set. The training data is what you build your model with. The test data is withheld to understand the quality of your model. When we split the data, we're splitting the *rows* of the data, not the columns. This means our training and test data will have the same columns, but different rows.

You may have heard of the term *overfitting* in the context of modeling. If your model performs very well on your training data but not on your test data, it may be overfit (that is to say, your model isn't generalizable). Having a test dataset helps you combat this. The goal of building a machine learning model is to help predict the future. By definition, we don't know the future, so we need to make sure our model is generalizable with the data we already know about. Withholding some amount of data to test the model on is necessary to do this.

Conceptually, it's most helpful to think about this in the context of a time series dataset. Even though this book doesn't formally cover time series predictive modeling, the same underlying context applies. Say you have five years' worth of data for your company sales, and you need to predict sales for year six, which hasn't started yet. The recommended approach would be to build your model as if year five had yet to start. You'd build your model on data from the first four years and then test its performance on year five. The goal of time series is to predict future dates, which is why it's best practice to

use a specific date cutoff instead of randomly splitting the data into the training and test sets.

This same concept applies to the general `train_test_split` function available in sklearn, as shown in Listing 5.1. If our data isn't specific to time series data (our second and third use cases are examples of this), we use the `train_test_split` function. In creating a training and testing dataset in this way, we're randomly holding out *X*% of the data rather than identifying a logical point in time to split the data. The purpose of this function is to split the rows from `x` and `y` into their respective train and test objects.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.2,
  random_state=42
  )
```

Listing 5.1 Train-Test Split Code

There are a few components to understand here, the first being the objects you're creating. The output of `train_test_split()` is four different objects:

1. Your input variables that are used to create the model.
2. Your input variables for the testing of your model.
3. What you're trying to predict so your model has the "answer" to train itself.
4. The "answer" to compare against the model's predictions for your testing data.

The inputs to the `train_test_split()` function are:

- **`X`**
  All the columns used to create the prediction.
- **`y`**
  The column you're trying to predict.
- **`test_size`**
  What percentage of the data should be held out for testing.
- **`random_state`**
  Any number that will enable you to create reproducible results as you iterate on your model.

The `test_size` parameter may require some explanation. What is the right amount of data to leave out for testing? In general, you'll likely select a number between 20% and 30%. The smaller and/or more nuanced your dataset, the more data you want to reserve for testing. This selection of your `test_size` ties back to the concept of overfitting and

how to reduce the risk of overfitting going undetected. In general, the larger your test size, the lower the risk of you not detecting the overfitting in your modeling process.

The `random_state` parameter enables your results to be more reproducible—but how? The specific number you provide does not matter; it's the consistency in which you apply that number. If you use 17 and you run the `train_test_split()` function multiple times, you'll always get the same rows going into your train and test datasets. Especially when you're iterating on your model, this ensures your model performance results are driven by intentional changes, rather than the difference in how the function split your data into a training set and test set.

As we go through each model, you'll find that the process remains consistent. Applying the same `X` and `y` allows you to test multiple models and evaluate their performance. This increases your speed of iteration significantly!

5.4 Regression Models

Regression models are the backbone of analytics, and the underlying math is integral to how value is derived from data. At their core, regression models are a fancier version of correlations. As promised at the beginning of the book, this section won't include a complicated math lesson, but we'll revisit high school algebra class for a quick refresher of the underlying concepts. We'll then walk through linear and logistic regression in detail and apply them to our three use cases.

5.4.1 What Are Regression Models?

Regression models are based on the $y = mx + b$ equation. High school math may have been many years ago for some of you, so as a refresher, y is the value we're trying to predict, m is the slope of the line, x is the data point we know, and b is the y intercept. The biggest difference is that as you add additional variables, you create more variables like mx.

The regression model looks to optimize this equation to identify the line of best fit, which essentially means it's trying to create the version of the $y = mx + b$ equation that best matches the data by reducing the distance between the line the equation would make and the actual data. To further explain this, we'll walk through the simplified example shown in Figure 5.2.

You'll notice that we're only using two variables (student intelligence and exam scores), which is intentional for visualization purposes. It's easy for us to consume a two-dimensional line graph, but in practice you're not building a model with only one column to predict another column. When you have many columns in your data, the same concept is applied—you're just adding another variable to your equation. For example, if your

data has three columns being used to predict a fourth column, *y = mx + b* will become *y = m1x + m2x + m3x + b*. As you add new columns, the regression model adds another *mx* for each of them.

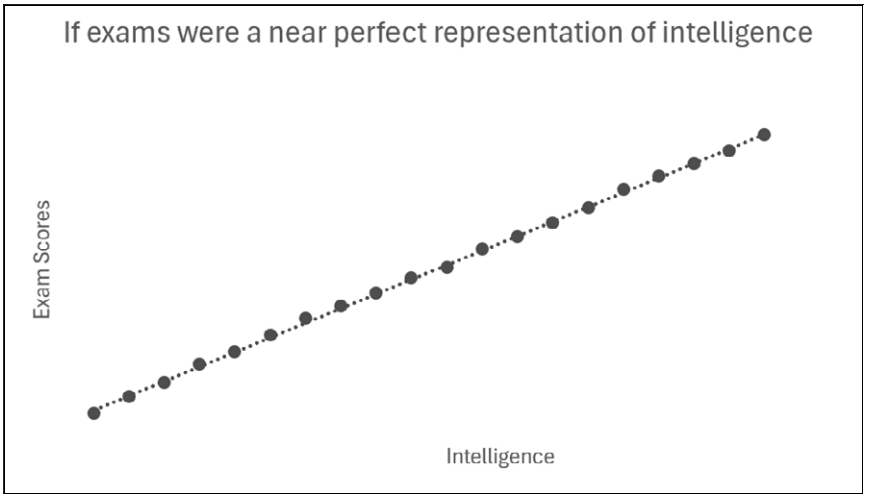


Figure 5.2 Near-Perfect Regression Line

For this example, the equation of this line is *y = 1.8256x + 1.7258* (Excel has an option to show you any line graph line's linear equation). Since this is an equation, the predictive nature of regression involves inputting *x* when its value is still unknown. If you haven't already started making this connection, this is why regression is so much more interpretable than other machine learning algorithms! You have an equation where you can set all other variables to 0, which gives you the overall relationship to the column you're trying to predict. Regression models will output this for you as *coefficients*. In our example, the coefficient for is 1.8256, or the *m* in our linear equation.

Chapter 6 will cover evaluating your model, but the most common evaluation metric in regression is R^2. This metric evaluates the amount of variance your regression equation is account for, with 1 meaning it's a perfect match and 0 meaning it's capturing none of the variance. For this equation, the R^2 is 0.9995 (almost as if it was planned to be nearly perfect).

Now what happens when we take the same example, but the data is messier? Let's look at Figure 5.3.

With this messier data, we still see a distinct linear pattern. The R^2 for this line is still very high at 0.9553, meaning the model can pick up 96% of the variance in this dataset.

What happens when we take this same data and give it a trend where students at the upper end of the intelligence distribution score higher on their exams (see Figure 5.4)?

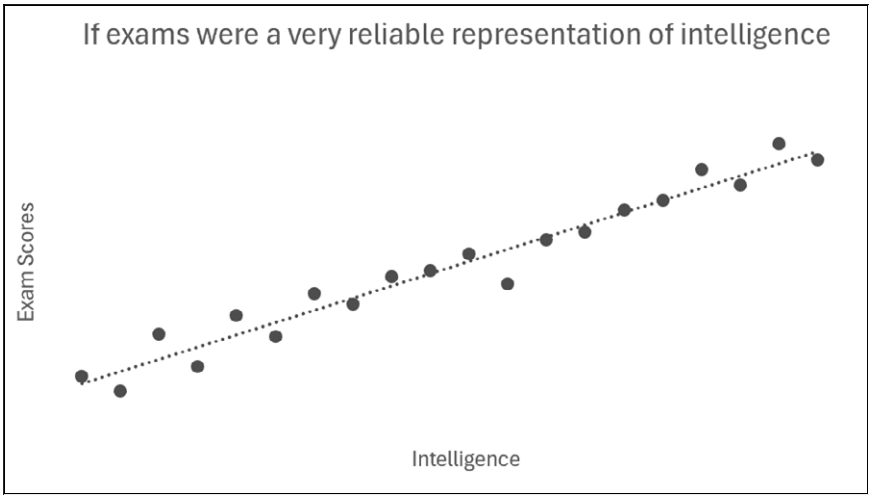


Figure 5.3 Messier Example of Data

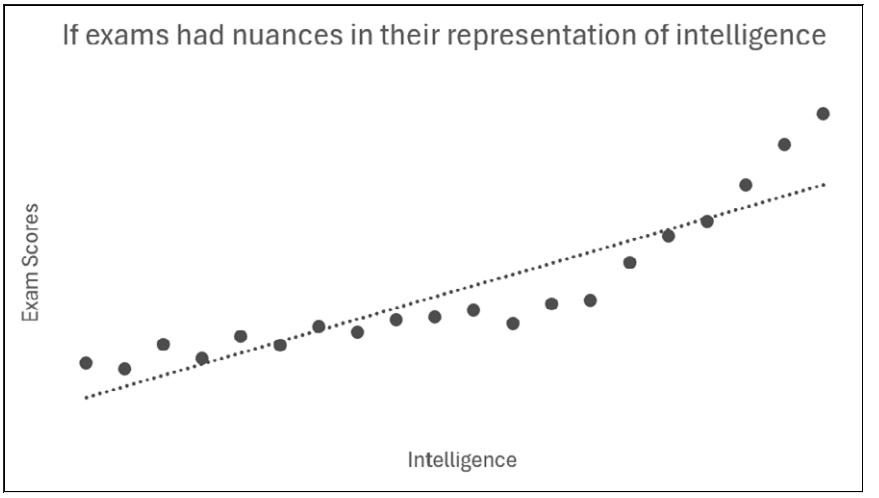


Figure 5.4 Higher Intelligence Students Score Higher on Exam

By making this change, the line is more than twice as steep, and the R^2 has dropped down to 0.8125! It's important to understand why this is and where there are limitations in linear regression. Linear regression is only able to map a straight line through the data. As you add additional variables, if those variables don't explain the trend of students with higher intelligence, the model has no way of knowing the top 25% of students have a different trend of exam scores. This also creates a dangerous dynamic where your model is better at predicting one group versus another. If you want to sound smart at a dinner party, this is called *Simpson's paradox*.

5.4.2 Multicollinearity

The math that regression is based on can be quite picky. A regression model has many assumptions. The most important assumption to consider when you're preparing your data is to make sure none of your columns are highly correlated with each other. This is called multicollinearity.

In general, you should address any correlation above 0.7 between two of the columns you're using to build a model. You can find the correlation of any two columns using the `pandas` function `corr`. When you provide the function with your columns, it outputs a correlation matrix.

There are varying approaches for addressing the high correlation between columns:

- The simplest approach is to remove one of the two correlated variables. One common cause for multicollinearity is bringing together different datasets where two of the columns have essentially the same data, just from a different source. In a case like this, it makes sense to just remove one of those columns. In other cases where it isn't as clear-cut, use your knowledge of the data and business process. If one column is more reliable than another, use the more reliable one!
- The other approach is dimensionality reduction, which we covered in Chapter 4, Section 4.3.6. The goal of dimensionality reduction is to reduce the number of columns while maintaining as much of the information as possible. Regardless of the technique, two highly correlated columns will be consolidated based on how that technique operates.

You may still be wondering why this actually matters. What harm can multicollinearity cause? The biggest issue is that your coefficients won't be calculated properly. One of the benefits of regression is that it isolates a variable's impact on what you're trying to predict. When you have two columns with the same impact, the underlying math cannot isolate properly. This results in small changes in your data leading to significantly different predictions.

5.4.3 Linear Regression

The demonstration shown in Section 5.4.1 was linear regression. In its simplest form, your model is a formula that generates a straight line through your data. One of the major challenges with linear regression is that straight line component. If there are different trends for different groups, the best approach is often to create two models instead of one. For example, if you're predicting the sales of two brands, their underlying sales drivers may be different, so a linear regression model may get confused by the independent trends of these distinct brands. The output is a bad model that compromises between both groups, and that isn't useful for anyone.

In practice, this creates three major limitations:

- Finding trends across your data can be time-consuming and difficult. Trends also change over time, so the effort required to identify the trends that will drive how you separate your data can lead to a time-intensive retraining process.
- As you split your data, each additional split reduces your sample size. Combine this with your required test train split, and you can run into sample size issues. You should have hundreds of rows at an absolute minimum to consider linear regression in this predictive modeling capacity.
- Creating separate models increases the effort and time required to maintain the models, because each one requires testing and validation.

The use case for linear regression is to predict a number, meaning it works best for numeric predictions, such as the score of a test, a sales number, or someone's IQ.

The code for creating a linear regression is quite straightforward. There aren't any hyperparameters to tune like there are with the machine learning models (which we'll cover in Section 5.5). As you can see in Listing 5.2, after you've created your training and testing data, it only takes five lines of code to build the model and identify its accuracy on the testing data. First, we use the `train_test_split` function to generate the four necessary components to train and test our model. Then, we instantiate our model (a fancy word for creating it) and fit it using the `fit` function. Finally, we generate our predictions using the `predict` function and generate the accuracy with the `accuracy_score` function.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.2,
  random_state=42
  )

model = LinearRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

Listing 5.2 Linear Regression Code

Now that we've gone through the basics of linear regression, let's discuss it in the context of our use cases. You'll see that this is where many of the nuances and complexities of these particular use cases become reality.

Use Case 1: Linear Regression

Can we satisfy Chris with linear regression? Let's find out! As a reminder, Chris wants to know how many sales he can expect in the next 3 months.

We'll leverage some of our existing data prep, which we saved to a *.csv* file. This will be used as the data source. As a reminder, we're working with the columns listed in Table 5.1.

Column Name	Column Type
Invoice	Integer
StockCode	Object
Description	Object
Quantity	Float
InvoiceDate	Object
Price	Float
Customer ID	Float
Country	Object
New_invoicedate	Object
Invoicedate	Object
Description Grouped	Object
Customers Grouped	Float

Table 5.1 Column Names and Types

You may notice a handful of things. First, we're not bringing in our dummy column data, which is intentional for the purposes of linear regression—we're seeing how the model will perform without it. As a starting point, we'll see if we can build a model without this data. Listing 5.3 shows us loading in the data using the `pd.read_csv` function.

```
#read libraries
import pandas as pd
import numpy as np
```

```
#create in cleaned data from previous chapter
df_cleaned = pd.read_csv("df_cleaned.csv")
```

Listing 5.3 Loading In Data

It's important to think in the context of the model being used to actually predict the future. There is an entire category of predictive modeling specific to time series data. Time series models are designed to identify time patterns in the data, such as seasonality and other recurring trends. Our use case has elements of time series data, so we could consider leveraging a traditional time series model. While this isn't a book about time series models, this distinction can help you understand why it makes more sense to use a regression or machine learning model in the real world.

Time series models have many benefits and limitations. The challenge with these models is you're predicting an end result. For example, in this use case, we'd be predicting sales with no other variables as inputs. This means the prediction relies solely on the *historical* trends of sales data to predict the future. Feature-based models are the opposite; they're able to account for other information that may be influencing the number you're trying to predict, but they don't naturally identify the time-based trends in the data. That's why feature engineering is important—it enables a feature-based model to see trends in the data. Ultimately, it's best to use an ensemble approach, where you build both a time series model and a more feature-based model (such as regression or machine learning). Given that the scope of this book is specific to machine learning models, we won't go into building a time series model.

Lagging variables is one approach to creating a feature-based model. You perform lagging by joining your data onto itself after shifting it back *n* number of hours, days, weeks, or months. This is what enables a forward-looking prediction once you've trained your feature-based model for a time-based use case. For example, in this use case, we're predicting sales. If we only have features that are the last *n* number of months ago in our data, we can use it to predict future sales.

Before we start playing with this data, let's take a step back to make sure you understand how to set it up. Our data is currently a log of all transactions, but that isn't the data structure that will get us what we need. When building a model—especially a regression model—you need to summarize your data to an appropriate level for the model. How this looks goes back to your use case.

Chris wants to predict the next 3 months of sales. To do this, we'll aggregate our data up to the day level. From there, we're able to add the results up to monthly data. This strikes a balance between keeping enough data to build a model (which is why we don't aggregate up to a monthly level) and giving too much detail (like keeping it at the product level). We'll use the pandas `groupby()` function for this, as shown in Listing 5.4. This summarizes our data up to the daily level. The rationale for taking a distinct count of `Description`, `Customer ID`, and `Country` is that we're trying to capture the potential relationship of how products, customers, and countries may be impacting sales.

```
#create an aggregated version of the data
df_data_grouped = df_cleaned.groupby(
  'invoicedate'
  ).agg({
    'Description': 'nunique',
    'Customer ID': 'nunique',
    'Country': 'nunique',
    'Quantity': 'sum',
    'Price':'sum'
  }).reset_index()
```

Listing 5.4 Grouping Data by Day

To facilitate this variable lagging, we'll use the `datetime` library. We'll start by lagging our data by 3 months, as shown in Listing 5.5. This adds a new `invoicedate` column to our summarized dataset, which will then be used to join the data for the purposes of lagging our data.

```
#load in data specific libraries
from datetime import datetime
from dateutil.relativedelta import relativedelta

#create date from the invoice timestamp
df_data_grouped['invoicedate'] = pd.to_datetime(
  df_data_grouped['invoicedate']
  ).dt.date

#add 3 months to the date
df_data_grouped['invoicedate_minus_3'] =
  df_data_grouped['invoicedate'] + relativedelta(months=3)
```

Listing 5.5 Lagging by 3 Months

Next, we set up our data for the join. Lagging variables can get confusing; it's easiest to find a date to ground yourself on to be the focus of predicting future values. The following code contains the main date (`invoicedate`) and the target variable we'll use to create the model (`Price`):

```
#select only the date and price
df_for_model = df_data_grouped[['invoicedate', 'Price']]
```

To set ourselves up for success, we'll only select our desired columns. It's helpful to reduce the columns before joining data, which can be done using double brackets (`[[`) to name each column you'd like to keep (see Listing 5.6). This is cleaner and reduces the risk of confusing you (or others) about which columns belong to the original dataset and which are lagged.

```
#select only needed columns for the model
df_minus_3 = df_data_grouped[[
  'invoicedate_minus_3',
  'Description',
  'Customer ID',
  'Country',
  'Quantity',
  'Price'
  ]]
```

Listing 5.6 Cleaning Up Data Before Join

We'll also want to do some feature engineering. As shown in Listing 5.7, feature engineering gives the model context about time. The first step is identifying in which month an activity occurred, with the goal of giving the model context about yearly trends by identifying the month within the year. We use the `dt.month` function to extract the month from the date. The second step gives the model context about the overall timeline. We're calculating how many days have passed since sales were tracked. We do this by subtracting the two dates and then converting the difference into the number of days using the `dt.days` function.

```
#create the month from the date
df_minus_3['month'] = pd.to_datetime(
  df_minus_3['invoicedate_minus_3']
  ).dt.month

#calculate number of days between the date and the first sale
df_minus_3['since_first_sale'] =
  pd.to_datetime(df_minus_3['invoicedate_minus_3']) - min(pd.to_datetime(df_for_
    model['invoicedate']))

#convert calculation to a numeric value
df_minus_3['since_first_sale'] = df_minus_3['since_first_sale'].dt.days
```

Listing 5.7 Time-Based Feature Engineering

Now we can bring the data together by using the `pd.merge` function, joining the date column from each DataFrame. The tradeoff with lagging your variables is that your oldest day now has to be reduced by your largest time gap. In this example, we'll lose out on the first 3 months of our dataset because those data elements don't have any data to reference 3 months prior. This blank data is filtered out in the last line of Listing 5.8.

```
#merge lagged data back onto main dataset
df_for_model = pd.merge(
  df_for_model,
```

```
  df_minus_3,
  how = 'left',
  left_on = 'invoicedate',
  right_on = 'invoicedate_minus_3',
  suffixes = ('','_3months')
  )

#remove records where lagged data is blank
df_for_model = df_for_model[pd.notna(df_for_model['invoicedate_minus_3'])]
```

Listing 5.8 Joining and Removing Blank Data That Can't Be Lagged

Our data is now ready for the actual modeling process! In most cases, we want to randomly split our data; however, this doesn't make sense when we're dealing with time-based data. If the goal is to predict the next 3 months of sales, it's better to first see if you can predict the most recent 3 months of sales. This replaces the `train_test_split()` you'd normally do to randomly split the dataset. You can also use `shape` to get a sense of how many rows are in the training set and test set.

As shown in Listing 5.9, we use the `relativedelta` function to calculate the date at which to split the data into training and test setsz. This date is then used to split our data into the training and test sets using the traditional Python subsetting approach.

```
#calculate the maximum data date
max_data_date = max(df_data_grouped['invoicedate']) - relativedelta(months=3)

#split into training and test data
training_data = df_for_model[df_for_model['invoicedate'] <= max_data_date]
test_data = df_for_model[df_for_model['invoicedate'] >= max_data_date]

#print number of rows and columns for each training and test datasets
print(training_data.shape)
print(test_data.shape)
```

Listing 5.9 Creating Training and Test Datasets

In this split, there are 372 rows in the training set and 67 in the test set. Only 15% is left for the test set, which is lower than ideal, but it's okay for our time-based use case. The goal of a larger test set is to ensure generalizability. In a time series example like this one, if the model can generalize for the next 3 months of test data, we can feel more confident that it will create reliable future predictions. We'll make sure only the numeric columns are included by removing our date columns, as shown in Listing 5.10.

```
#select only numeric columns for the training data
training_data = training_data[[
  'Description',
```

```
  'Customer ID',
  'Country',
  'month',
  'since_first_sale',
  'Price_3months',
  'Price'
  ]]

#select only numeric columns for the test data
test_data = test_data[[
  'Description',
  'Customer ID',
  'Country',
  'month',
  'since_first_sale',
  'Price_3months',
  'Price'
  ]]
```

Listing 5.10 Selecting Only the Necessary Columns

Since we're not using `train_test_split()` to split the day, we now need to split both the training and test datasets into the `x` and `y` datasets using the `iloc` function, which uses the position of a row or column to select data. In this code, we're selecting all columns, noted by the colon (`:`) before the comma. We then use `:-1` to select all but the last column for the `X` datasets and `-1` to select only the last column for the `y` datasets, as shown in Listing 5.11.

```
#create the predictors for the model's training and test data
X_train = training_data.iloc[:, :-1]
X_test = test_data.iloc[:, :-1]

#create the target for the training and test data
y_train = training_data.iloc[:, -1]
y_test = test_data.iloc[:, -1]
```

Listing 5.11 Split into Required x and y Columns for Model

Now we can train the model! Since we're predicting a number, we need to select a different metric to evaluate the model. A good one to start with is the mean absolute error (MEA) function, `mean_absolute_error()`.

Mean Absolute Error

Chapter 6 discusses the various metrics in more depth, but MAE may be the easiest to explain to stakeholders since it shows the average of the error. For example, by taking the absolute value, errors of +5 and -5 don't cancel out to 0.

In Listing 5.12, we've used the same model workflow where we instantiate the model, fit, predict, and then measure the accuracy on the test dataset.

```
#read in the linear regression and mae libraries
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

#create model
model = LinearRegression()

#fit the model to the data
model.fit(X_train, y_train)

#create the predictions for the test data
y_pred = model.predict(X_test)

#use MAE to measure the accuracy
accuracy = mean_absolute_error(y_test, y_pred)

#print the accuracy results
print(f"Accuracy: {accuracy}")
```

Listing 5.12 Building Linear Regression Model with 3-Month Lag

The MAE result is 2,052, meaning for each day the model is off by roughly $2,000 (on an absolute basis). Is this a lot? Yes, given that that's a notable amount of each day's overall sales. Another way to look at this is by summing your predictions and the test data, as follows:

```
#print the sum of both the actual target for the test data and the model's pre-
dictions
print(sum(y_test))
print(sum(y_pred))
```

The result is $423,620 for the test data and $301,129 for the predictions, which means our prediction is only 71% of the actual sales for the period we're predicting. This is a pretty good indication that our model is underpredicting. If you recall what we found in our data exploration in Chapter 4, Section 4.2, sales increase around September through the end of the year due to the holiday season. Our model doesn't seem to have this context,

so what can we do to get it into the model? The simplest way is to add a dummy variable that will identify this trend in the data. In the following code, we've created the same new column on both the `X_train` and `X_test` datasets, identifying any month after September as a holiday month using the `np.where` function, which behaves like an `if-then` statement:

```
#create new holiday season feature on both the training and test data
X_train['holiday_season'] = np.where(X_train['month'] >= 9, 1, 0)
X_test['holiday_season'] = np.where(X_test['month'] >= 9, 1, 0)
```

After making this update, we can now rerun the model using the same code we used to run the model in Listing 5.12. The results are a notable improvement! Our MAE is down to 1,676. Our model now predicts 83% of our actual sales data, so we've gotten much closer just by adding this new variable.

Let's see if we can improve it even further. We'll plot our predictions of sales compared to the test data over time to see if we can glean any insights using the code from Listing 5.13. First, we use the `plot` function, passing it our date and `y_test` data. Next, we pass the same date but with the `y_pred` data. This allows us to add multiple lines onto the same graph. Figure 5.5 displays the resulting graph.

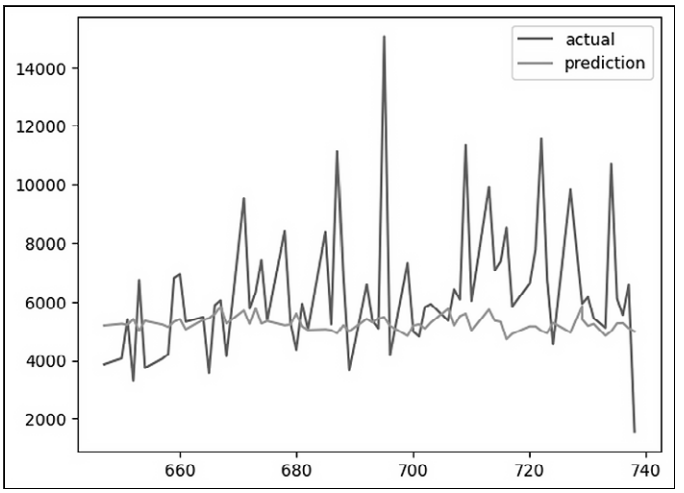


Figure 5.5 Plot of Actuals Versus Predictions

```
#read in plotting library
import matplotlib.pyplot as plt

#plot the actual data
plt.plot(test_data['since_first_sale'], y_test, label = "actual")

#plot the predictions
plt.plot(test_data['since_first_sale'], y_pred, label = "prediction")
```

```
#show the legend
plt.legend()
```

Listing 5.13 Plotting Actuals Versus Prediction

When we plot the data this way, it's clear there are spikes in sales on specific days that aren't being accounted for. There appears to be some type of trend. Let's see if the day of the week has any impact on sales. This presents a new question: What's the right way to bring in this data? Do we want to bring it in as a numeric variable (0-6), or should we bring it in as a categorical variable and dummy code? Let's try both!

To start, we'll go back and recreate our dataset, as shown in Listing 5.14.

```
#select only the date and price
df_for_model = df_data_grouped[['invoicedate', 'Price']]

#select only specific fields
df_minus_3 = df_data_grouped[[
  'invoicedate_minus_3',
  'Description',
  'Customer ID',
  'Country',
  'Quantity',
  'Price']]

#create month feature
df_minus_3['month'] = pd.to_datetime(
  df_minus_3['invoicedate_minus_3']
  ).dt.month

#create days since first sale feature
df_minus_3['since_first_sale'] =
  pd.to_datetime(df_minus_3['invoicedate_minus_3']) - min(pd.to_datetime(df_for_
    model['invoicedate']))

#convert days since first sale to numeric value
df_minus_3['since_first_sale'] = df_minus_3['since_first_sale'].dt.days

#join onto original data
df_for_model = pd.merge(
  df_for_model,
  df_minus_3,
  how = 'left',
  left_on = 'invoicedate',
```

```
  right_on = 'invoicedate_minus_3',
  suffixes = ('','_3months'))
```

Listing 5.14 Recreate the Data

Now we add in the new `weekday` data and recreate our training and test datasets, as shown in Listing 5.15.

```
#add day of the week feature
df_for_model['day_of_week'] = pd.to_datetime(
  df_for_model['invoicedate']).dt.weekday

#remove values that the lagged data isn't populated for
df_for_model = df_for_model[pd.notna(df_for_model['invoicedate_minus_3'])]

#calculate the max data date for splitting our model into train and test
max_data_date = max(df_data_grouped['invoicedate']) - relativedelta(months=3)

#split into training and test data
training_data = df_for_model[df_for_model['invoicedate'] <= max_data_date]
test_data = df_for_model[df_for_model['invoicedate'] >= max_data_date]

#select training data fields
training_data = training_data[[
  'Description',
  'Customer ID',
  'Country',
  'month',
  'since_first_sale',
  'Price_3months',
  'day_of_week',
  'Price'
  ]]

#select test data fields
test_data = test_data[[
  'Description',
  'Customer ID',
  'Country',
  'month',
  'since_first_sale',
  'Price_3months',
  'day_of_week',
  'Price'
  ]]
```

```
#split into X training and X test
X_train = training_data.iloc[:, :-1]
X_test = test_data.iloc[:, :-1]

#split into y training and y test
y_train = training_data.iloc[:, -1]
y_test = test_data.iloc[:, -1]

#create the holiday season feature on both the training and test data
X_train['holiday_season'] = np.where(X_train['month'] >= 9, 1, 0)
X_test['holiday_season'] = np.where(X_test['month'] >= 9, 1, 0)
```

Listing 5.15 Recreate Our Training and Test Datasets

Then, we rerun our model and the code for the graph, as shown in Listing 5.16.

```
#load in linear regression and MAE libraries
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

#create linear regression model
model = LinearRegression()

#fit model to the data
model.fit(X_train, y_train)

#create prediction on the test data
y_pred = model.predict(X_test)

#calculate and print the MAE of the test data results
accuracy = mean_absolute_error(y_test, y_pred)
print(f"Accuracy: {accuracy}")

#load in plotting library
import matplotlib.pyplot as plt

#plot both the actual and prediction data on the graph
plt.plot(test_data['since_first_sale'], y_test, label = "actual")
plt.plot(test_data['since_first_sale'], y_pred, label = "prediction")
plt.legend()
```

Listing 5.16 Rerun Data with the Weekday Numeric Variable

The outcome is essentially unchanged, meaning we didn't get any value in adding the weekday data as a number. The graph in Figure 5.6 shows how our model still doesn't account for the spikes in sales.

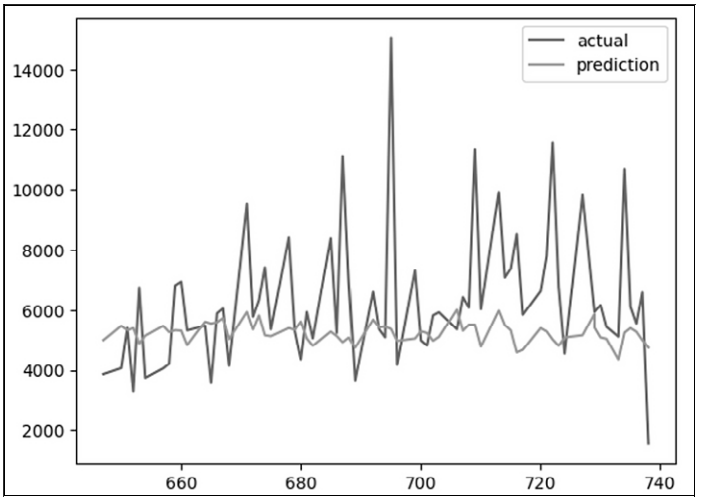


Figure 5.6 Model with Numeric Weekday

When happens when we bring in the data as categorical variables? We can do this using the `get_dummies` function we used in Chapter 4, as shown in Listing 5.17. We provide the function with the data of the `week` column. It then converts it into 7 columns, and we use the `pd.concat` function to add this back to the `X_train` DataFrame. We repeat these steps for the `X_test` data to ensure both DataFrames have the same transformations executed on them. If we don't do this, the model will not operate as intended.

```
#dummy code the day of the week on the training data
train_weekday_dummies = pd.get_dummies(
  X_train['day_of_week'],
  prefix='weekday'
  )

#add dummy coded day of the week to the training data
X_train = pd.concat(
  [X_train, train_weekday_dummies],
  axis = 1
  )

#dummy code the day fo the week on the test data
test_weekday_dummies = pd.get_dummies(
  X_test['day_of_week'],
  prefix='weekday'
  )
```

```
#add dummy coded day of the week to the test data
X_test = pd.concat(
  [X_test, test_weekday_dummies],
  axis = 1
  )
```

Listing 5.17 Dummy Code the Weekday Variable

Now we can rerun the model with the same code we've been using. Our error gets lower: It's now 1,560, about 100 lower than our last iteration. What's interesting now is that if you sum the predictions and the actual prices for the testing day, you'll find the gap has grown to about 82%. So, what's happening? The model is predicting a lower value than the last iteration, but if you look at Figure 5.7, you can see it's starting to account for some of the spikes in price—just not to the degree it needs to.

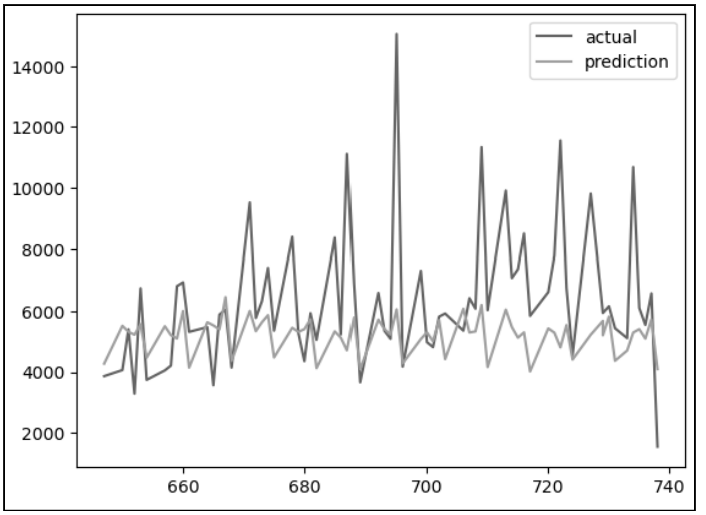


Figure 5.7 Weekday as Categorical Variables

Where do we go from here? This is a good example of a situation where you would likely want to evaluate other models that are better at identifying the trends in the data. This example was written without the benefit of hindsight, and we can hypothesize that the interaction of month and day of week from a tree-based model is more likely to yield the results we're looking for to better account for the spikes. This is covered in Section 5.5.1, where you'll see that a primary benefit to the tree-based models is their ability to find these nonlinear interactions within the data.

Something to remember—especially when Chris is your stakeholder—is that the past can't always predict the future. This is a common aspect to consider when educating your stakeholders. If they don't have an analytics background, they may think you can build a model for anything. Sometimes that just isn't the case!

Use Case 2: Linear Regression

The objective for our second use case is to predict the chances that someone will repeat an order within the next 7 days. Any time you hear any word related to the probability of an outcome (chance, probability, etc.), this is the sure sign that linear regression won't be the model you're going to use. Instead, you'll use logistic regression, which we'll discuss in Section 5.4.4.

Use Case 3: Linear Regression

This use case is about predicting the probability of an area having crime on a given day. You might be thinking that linear regression isn't the best approach here because the stakeholder is requesting a probability, not a number. However, think about how the data has presented itself. When we summarized it by area and day, almost every single area had crime on that given day. So why would we try to predict a probability? The model will ultimately be used to inform stakeholders on where to allocate resources, and it can achieve this by knowing which area on a given day will have more or less crime.

We'll explore how linear regression can achieve this. As always, the first step is to read our data, as shown in Listing 5.18. We'll use the summarized dataset we created in Chapter 4, Section 4.3.6.

```
#read in pandas
import pandas as pd

#read in data from previous chapter
df = pd.read_csv("df_summarized")
```

Listing 5.18 Import Summarized Dataset

We'll run a simple model right away to find our starting point with the final model. For now, we'll intentionally exclude the area name as a feature in the model to see what the results look like without it. We'll also drop the `DATE` column, which can't be included because it's not a numeric column, as well as the `crime_count` column, which is our target variable. (We're already pushing the limit of linear regression with the nearly 50 columns we're attempting to use as predictors.) Listing 5.19 shows the code for this. We follow the same modeling workflow as other use cases, which entails splitting our data into train and test sets, creating the model, fitting it, and then evaluating it. Our evaluation metric in this case is a combination of MAE on its own as well as MAE divided by the mean of `y_test`. We do this to normalize the MAE metric to the context of the dataset.

```
#read in sklearn libraries for the modeling process
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
```

```
#split data into what will be used to predict and what is being predicted
X = df.drop(columns=['DATE', 'AREA NAME', 'crime_count'])
y = df['crime_count']

#perform standard train test split
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42)

#create and fit the regression model
model = LinearRegression()
model.fit(X_train, y_train)

#predict with the modeling using the test data
y_pred = model.predict(X_test)

#calculate the mean absolute error
mae = mean_absolute_error(y_test, y_pred)

#calculate the mean absolute error divided by average of the target
mae_by_avg_target = mae / y_test.mean()

#print both results
print(mae)
print(mae_by_avg_target)
```

Listing 5.19 Create Linear Regression Model

The result is relatively promising, given the context. The MAE comes back at 6.5, which is roughly 25% of the average target value (for those familiar, this is a similar calculation to mean absolute percentage error [MAPE]). The MAE number is harder to interpret on its own, but the result of 25% can loosely be used as an accuracy percent: The model has an error rate of 25%. Not bad for an initial model where we've done nothing to account for the trends in the data with date specific features!

Given this good starting point, let's take a look at the coefficients to discover what the model finds valuable. In Listing 5.20, we've used the `coef_` function to pull out the coefficients from the model into a DataFrame. The results are shown in Figure 5.8.

```
#extract the coefficents from the model
coefficients = model.coef_

#get the feature names from the model
feature_names = model.feature_names_in
```

```
#add the these into a series to display the coefficients alongside their feature
names
pd.Series(data=coefficients, index=feature_names)
```

Listing 5.20 View Coefficients of the Model

```
Vict Age                                                                -0.082311
crime_description_ASSAULT WITH DEADLY WEAPON, AGGRAVATED ASSAULT        17.965487
crime_description_BATTERY - SIMPLE ASSAULT                              11.685328
crime_description_BURGLARY                                              21.373779
crime_description_BURGLARY FROM VEHICLE                                  2.213384
crime_description_INTIMATE PARTNER - SIMPLE ASSAULT                     -4.621731
crime_description_Other                                                 12.631013
crime_description_ROBBERY                                               10.403193
crime_description_SHOPLIFTING - PETTY THEFT ($950 & UNDER)             -14.095029
crime_description_THEFT FROM MOTOR VEHICLE - GRAND ($950.01 AND OVER)    0.086295
crime_description_THEFT FROM MOTOR VEHICLE - PETTY ($950 & UNDER)      -16.550620
crime_description_THEFT OF IDENTITY                                     27.568402
crime_description_THEFT PLAIN - PETTY ($950 & UNDER)                    -7.034606
crime_description_THEFT-GRAND ($950.01 & OVER)EXCPT,GUNS,FOWL,LIVESTK,PROD  -1.684420
crime_description_VANDALISM - FELONY ($400 & OVER, ALL CHURCH VANDALISMS)   -2.449467
crime_description_VANDALISM - MISDEAMEANOR ($399 OR UNDER)              -1.689151
crime_description_VEHICLE - STOLEN                                     -14.324866
victim_gender_F                                                         14.944671
victim_gender_M                                                         12.896172
victim_gender_Other                                                     13.636149
victim_race_B                                                           24.288182
victim_race_H                                                            0.693986
victim_race_O                                                            7.012393
victim_race_Other                                                        3.854551
victim_race_W                                                            3.260104
victim_race_X                                                            2.367777
crime_premises_DEPARTMENT STORE                                         10.114845
crime_premises_DRIVEWAY                                                 -7.053405
crime_premises_GARAGE/CARPORT                                           13.705114
crime_premises_MARKET                                                   -4.317378
crime_premises_MULTI-UNIT DWELLING (APARTMENT, DUPLEX, ETC)              3.061521
crime_premises_OTHER BUSINESS                                            5.280746
crime_premises_OTHER PREMISE                                             4.523803
crime_premises_OTHER RESIDENCE                                         -14.245634
crime_premises_Other                                                     4.543209
crime_premises_PARKING LOT                                               3.744716
crime_premises_PARKING UNDERGROUND/BUILDING                             11.733309
crime_premises_RESTAURANT/FAST FOOD                                      2.175149
crime_premises_SIDEWALK                                                 14.644929
crime_premises_SINGLE FAMILY DWELLING                                   -5.715358
crime_premises_STREET                                                    0.531290
crime_premises_VEHICLE, PASSENGER/TRUCK                                 -1.249864
crime_weapon_HAND GUN                                                    5.589493
crime_weapon_Other                                                       1.690407
crime_weapon_STRONG-ARM (HANDS, FIST, FEET OR BODILY FORCE)             13.973588
crime_weapon_UNKNOWN WEAPON/OTHER WEAPON                                13.333885
crime_weapon_VERBAL THREAT                                               6.889619
```

Figure 5.8 Coefficient Results

The values in Figure 5.8 suggest that the coefficient shows how much the predicted value changes when the variable increases by one unit. If the coefficient is negative, the result goes down, if the coefficient is positive, the result goes up. Think of this as a

correlation between the variable and the result you're trying to predict. We see a number of high and low values, which tells us there may be specific columns that will become important as we continue to build out our model.

Given how many columns we already have, we'll explore this further as we progress into tree-based models in Section 5.5.1.

In Chapter 4, Section 4.3.6, we promised to revisit principal component analysis (PCA)—and now is the time to do so. We ultimately won't use it for the final model, because as you'll see when we get to the tree-based models, our lower number of columns doesn't require us to apply PCA. However, for learning purposes, we'll apply PCA to the dataset and run through the linear regression model to give you an example of how you can apply PCA in practice to reduce your columns.

Training Principal Component Analysis

You should always train a PCA model on the training dataset, not your test dataset. This ensures that you're not at risk of leaking the answer to your model via the PCA process. Think about your model in practice. After you've built it and start using it to predict future events, you won't be able to run PCA on data that hasn't happened yet.

As shown in Listing 5.21, we're creating the PCA model with five components, fitting it to our data, using the `transform` function to actually apply the PCA to our training and test sets, and then printing out the explained variance of the PCA model.

```
from sklearn.decomposition import PCA

#create the pca object using 5 components
pca = PCA(n_components=5)

#fit the pca model to the training data
pca.fit(X_train)

#apply the pca model to the X_train data
X_train = pca.transform(X_train)

#apply the pca model to the X_test data
X_test = pca.transform(X_test)

#check the explained variance ratio
print("Overall Explained Variance: ",sum(pca.explained_variance_ratio_))
print("Explained Variance Ratio: ", pca.explained_variance_ratio_)
```

Listing 5.21 Apply PCA to the Data

The explained variance looks good—it's essentially 100% with only five components. Let's see how it performs when we run it through the model in Listing 5.22.

```
#create and fit the regression model
model = LinearRegression()
model.fit(X_train, y_train)

#predict with the modeling using the test data
y_pred = model.predict(X_test)

#calculate the mean absolute error
mae = mean_absolute_error(y_test, y_pred)

#calculate the mean absolute error divided by average of the target
mae_by_avg_target = mae / y_test.mean()

#print both results
print(mae)
print(mae_by_avg_target)
```

Listing 5.22 Run Linear Regression on Data That Has Gone Through PCA

The resulting MAE is 6.9, which is 26% of the average target value. This tells us that the PCA process worked extremely efficiently, with only a slight reduction in model performance while reducing our number of columns from almost 50 down to 5.

We'll end up skipping logistic regression for this use case since we're predicting a numeric value. We'll pick it back up in Section 5.5.1.

5.4.4 Logistic Regression

Logistic regression is used for classification. As discussed in Section 5.2.3, classification models predict probabilities, which then are translated to a binary outcome. Logistic regression is no exception to this. Just like linear regression, logistic regression models are formulas. The logistic regression formula is more complicated than linear regression, so we won't dive into the details. However, the overall concept is the same. The underlying math is what differs.

The use cases for logistic regression usually revolve around binary outcomes. Examples include:

- Customer retention
- Employee turnover
- Customer conversion

As you familiarize yourself with the various models at your disposal, you may start to see a trend. The vast majority of machine learning problems *could* be adjusted to be classification. For example, instead of predicting a sales number, what if you build a model that predicts the probability of reaching a sales goal? This type of approach can be more valuable for stakeholders. For example, if the stakeholder is expected to reach $10 million in sales for the next quarter, predicting a 75% probability of reaching that goal could have more value than predicting that sales over that period of time will reach $9 million.

The code for logistic regression should look very similar to linear regression. As you can see in Listing 5.23, the only change is that we're using `LogisticRegression()` instead of `LinearRegression()`. All the other code remains the same!

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.2,
  random_state=42
  )

model = LogisticRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

Listing 5.23 Logistic Regression Code

Let's dive into logistic regression for our use cases.

Use Case 1: Logistic Regression

Logistic regression isn't the right tool for this use case. Each row we're predicting is by day, so what would the probability or classification be? While we could try to predict a sales number by each day to identify whether a specific target could be met, we'd be using logistic regression for the sake of using logistic regression. Our next use case will be a better demonstration of the value of logistic regression.

Use Case 2: Logistic Regression

Our second use case is an ideal example of logistic regression. Our goal is to identify the probability that a customer will order again in the next 7 days. Logistic regression outputs a value from 0 to 1 identify the probability an event will occur. Logistic regression is the right model to start with for this use case.

Since this is our first model for this use case, we'll start with our data. What is the target variable for this use case? We've already stated the objective is identifying if someone will order within the next 7 days, but how does that translate to our data? Which column should we be using? Take a few moments to think about this before moving to the next paragraph.

This can be a tricky concept to consider when you're getting started with predictive models. The presence of another row indicates there is another order and therefore indicates our target variable. However, our algorithms can't magically pull from multiple rows; all the data required to train the model for each observation must be on the same row. Table 5.2 shows an abbreviated sample of data to illustrate this point.

Customer ID	Order Date	Ordered Again in 7 Days?
12345	1/1	Yes
98765	1/3	No
56789	1/8	Yes

Table 5.2 Example of How Target Variable Needs to Be Set Up

This example illustrates that we're adding *future* values to *historical* values. This is important! Think about it this way: If we're using the model to predict the future, we need to set up our data to look forward into the future, not backwards. In order to build the model, we need to provide it with snapshots in time that translate to what we ultimately want to predict.

For the model's purpose, the yes and no values become a 1 and 0, respectively. This is how the model can then predict the probability of the outcomes, since it's identifying the value between 0 and 1.

So how do we do this? Let's get started!

The first step is the easiest. We'll load in the data that we previously saved from our data cleaning and dummy coding in Chapter 4, Section 4.3. We'll then create a new DataFrame with only a handful of columns that we'll need (who ordered and when) using double brackets ([[) to select the columns. Listing 5.24 shows this code and Figure 5.9 shows a preview of the new DataFrame.

```
#read in pandas
import pandas as pd

#read in data from previous chapter
df = pd.read_csv("use_case_2_cleaned.csv")

#create dataset for lagging
df_lagging = df[['Customer ID', 'order_date']]

#preview data
df_lagging.head()
```

Listing 5.24 Creating Target Variable–Specific DataFrame

	Customer ID	order_date
5d6c2b96db963098bc69768bea504c8bf46106a8a5178e...	2024-09-10
0781815deb4a10a574e9fee4fa0b86b074d4a0b36175d5...	2024-09-10
f93362f5ce5382657482d164e368186bcec9c6225fd93d...	2024-09-10
1ed226d1b8a5f7acee12fc1d6676558330a3b2b742af5d...	2024-09-10
d21a2ac6ea06b31cc3288ab20c4ef2f292066c096f2c5f...	2024-09-10

Figure 5.9 Preview of Target Variable DataFrame

There's nothing fancy going on here; we're just creating a log of each order. This DataFrame can now be joined onto our main DataFrame to get the future dates onto our data. This will be a simple task for anyone who's used SQL before, because SQL allows you to use conditional logic to join date (e.g., date column `x` is greater than or equal to date column `y`). Unfortunately, this is a bit harder with Python. The `merge` function in pandas doesn't support joins on values like "less than or equal to" or "greater than or equal to." You can only join values with an equal comparison between two columns.

Regardless, the first step is to create the start (or floor) of the time window to which the future hours should be allowed to join (see Listing 5.25 and Figure 5.10). We're adjusting by 8 days using the `timedelta` function, so we don't need to be inclusive of the bottom range, which will help us when we join the data.

```
#read in date specific libraries
from datetime import datetime, timedelta

#create new date with difference of 8 days
df_lagging['order_minus_8'] = pd.to_datetime(
  df_lagging['order_date']) - timedelta(days=8)
```

```
#preview the data
df_lagging.head()
```

Listing 5.25 Create Floor Date for Which a Future Order Can Be Associated with a Previous Order

Customer ID	order_date	order_minus_8
5d6c2b96db963098bc69768bea504c8bf46106a8a5178e...	2024-09-10	2024-09-02
0781815deb4a10a574e9fee4fa0b86b074d4a0b36175d5...	2024-09-10	2024-09-02
f93362f5ce5382657482d164e368186bcec9c6225fd93d...	2024-09-10	2024-09-02
1ed226d1b8a5f7acee12fc1d6676558330a3b2b742af5d...	2024-09-10	2024-09-02
d21a2ac6ea06b31cc3288ab20c4ef2f292066c096f2c5f...	2024-09-10	2024-09-02

Figure 5.10 Preview of Customer Order with Date Window

Now that we have our date window, we need to bring the data together. There really isn't a great way to execute this in pandas, so we'll leverage a cross-join scenario where we join each customer ID transaction first and then we filter the dataset using our date window.

Listing 5.26 shows how we clean up the new DataFrame by using the `drop` function to remove the original `order_date` column and then merge the data. We're using an inner join in this case because we only want to return rows where there is a match between each dataset. If there isn't a match, it's too early in the dataset to create the lag. Since this is specific to the lagging, we're only selecting `Customer ID` and `order_date` from `df`. This will simplify future steps—when lagging, we only need the unique columns required for joining and the lagged variable, because we'll ultimately join this data back to the original dataset. This join will result in too many duplicate rows, since we're only joining on the `Customer ID` column.

```
#identify create new column called max_date
df_lagging['max_date'] = df_lagging['order_date']

#drop the order_date column
df_lagging = df_lagging.drop(columns=['order_date'])

#join lagged data onto the original data
df_lag_7 = pd.merge(
  df[['Customer ID', 'order_date']],
  df_lagging,
  how = 'inner',
  on = 'Customer ID'
  )
```

```
#print number of columns and rows for both df and df_lag_7
print(df.shape)
print(df_lag_7.shape)
```

Listing 5.26 Join All Customer Orders Together

The resulting DataFrame now has duplicates that we'll need to carefully address. We started with over 21,000 rows and are now up to almost 87,000 rows. When filtering by date, we also need to keep in mind that some orders won't have any other orders that fall within our date window. This is why we didn't overwrite our original DataFrame. Ultimately, we'll join all this data back onto that DataFrame to get the proper target variable column. As you may be realizing, the complexity and nuance of properly lagging this data can be quite tedious.

Next, we'll filter the dates by selecting rows where the order date is greater than that of the last 8 days and less than the max date column we've created, as shown in Listing 5.27. This reduces the data to give each row a single lag value since our initial join could only be done using `Customer ID`.

```
#filter the data to ensure only the necessary rows are matched
df_lag_7 = df_lag_7[
  (df_lag_7['order_date'] > df_lag_7['order_minus_8']) &
  (df_lag_7['order_date'] < df_lag_7['max_date'])
  ]

#show the number of columns and rows in the data
df_lag_7.shape
```

Listing 5.27 Filter the Cross-Joined Dataset to Be Within the Date Window

The results show we're back down to 4,524 rows. This indicates we're seeing 4,524 instances of customers ordering again within 7 days.

We can now bring this dataset back together with the main dataset, as shown in Listing 5.28. As you'll see, we do some additional cleaning to the target dataset to make the join easier. This includes adding a column of 1s to represent our target variables of 1 and 0. Since not all rows will have a match, and we need to ensure all rows have either a 1 or 0, we're filling in any values with a 0 using the `fillna` function.

Dropping duplicates is extremely important! As we only care if the customer orders at least one time in the next week, we want to ensure we don't create duplicate records for the instances where someone orders more than once in that given week.

```
#select only the customer id and order date fields
df_lag_7 = df_lag_7[['Customer ID', 'order_date']]
```

```
#drop duplicates
df_lag_7 = df_lag_7.drop_duplicates()

#create dummy column of all 1 values
df_lag_7['target_7'] = 1

#join the data to the original dataset after correction to the lagged dataset
df = pd.merge(
  df,
  df_lag_7,
  how = 'left',
  on = ['Customer ID', 'order_date']
  )

#fill in any blanks with 0
df['target_7'] = df['target_7'].fillna(0)
```

Listing 5.28 Merge Target Variable Data Back onto Main Dataset

Before we actually build the model, we should take a step back and think about our use case. Our stakeholder wants to view the last 7 days; however, now is the easiest time to add another interval. Given the number of customer orders, it's worth adding a 2-week or 14-day interval as well. Even if the stakeholder doesn't end up using it, it could be a natural next question. Having this already ready to review, shows you're well prepared and goes over favorably with stakeholders. It also presents an opportunity for you to identify additional insights between these different approaches that can bring value to your stakeholder.

The code for this is shown in Listing 5.29, where we've executed all the same steps we did for the 1-week lag but this time for 2 weeks. If you apply this same methodology more than twice, you should convert the code into a function that you can input the date interval. This will prevent the need to copy and paste the same code over and over again!

```
#select only the customer id and order date fields
df_lagging = df[['Customer ID', 'order_date']]

#create lag for 2 weeks column
df_lagging['order_minus_15'] = pd.to_datetime(
  df_lagging['order_date']) - timedelta(days=15)

#create max_date column from order_date
  df_lagging['max_date'] = df_lagging['order_date']
```

```
#drop the originally named order_date column
df_lagging = df_lagging.drop(columns=['order_date'])

#join the 14 day lag with the lagged dataset
df_lag_14 = pd.merge(
  df[['Customer ID', 'order_date']],
  df_lagging,
  how = 'inner',
  on = 'Customer ID'
  )

#select only rows that fit the desired criteria
df_lag_14 = df_lag_14[
  (df_lag_14['order_date'] > df_lag_14['order_minus_15']) &
  (df_lag_14['order_date'] < df_lag_14['max_date'])
  ]

#select only the customer ID and order date
df_lag_14 = df_lag_14[['Customer ID', 'order_date']]

#drop the duplicates
df_lag_14 = df_lag_14.drop_duplicates()

#create dummy column of all 1's
df_lag_14['target_14'] = 1

#join onto original dataset
df = pd.merge(
  df,
  df_lag_14,
  how = 'left',
  on = ['Customer ID', 'order_date']
  )

#fill the target field with 0's
df['target_14'] = df['target_14'].fillna(0)
```

Listing 5.29 Create 14-Day Target Variable

We're now ready to build the model! We'll build it in stages, since a full 40 columns going into a logistic regression model isn't the best idea. As a starting point, we'll want to evaluate the columns we think will have the most value. In Listing 5.30, we're adding the columns that are likely to be the most valuable using double brackets (`[[]`) and assigning them to `X`. Then we're noting our target variable and assigning it to `y`.

```
#select only desired columns
X = df[[
  'Total', 'KPT duration (minutes)', 'Rider wait time (minutes)',
  'DistanceNumeric', 'Restaurant name_Masala Junction',
  'Restaurant name_Swaad', 'Restaurant name_Tandoori Junction',
  'Restaurant name_The Chicken Junction', 'Subzone_Chittaranjan Park',
  'Subzone_DLF Phase 1', 'Subzone_Greater Kailash 2 (GK2)',
  'Subzone_Sector 135', 'Subzone_Sector 4', 'Subzone_Shahdara',
  'Subzone_Sikandarpur', 'Subzone_Vasant Kunj',
  'Cancellation / Rejection reason_Cancelled by Customer',
  'Cancellation / Rejection reason_Cancelled by Zomato',
  'Cancellation / Rejection reason_Items out of stock',
  'Cancellation / Rejection reason_Kitchen is full',
  'Cancellation / Rejection reason_Merchant device issue'
  ]]

#select target_7 as the target variable
y = df['target_7']
```

Listing 5.30 Select the Columns Most Likely to Be Predictive

Now that we've assigned these columns to `X` and `y`, we can move them into a standard model, as shown in Listing 5.31. One difference you'll see is that there are now parameter options when instantiating our model (`solver` and `random_state`). Other than that, it's the same process we've seen before, just using the `LogisticRegression` function.

```
#read in necessary libraries for the modeling process
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score

#split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.3,
  random_state=42
  )

#create logistic regression model
model = LogisticRegression(solver='liblinear', random_state=42)
```

```
#fit the model to the data
model.fit(X_train, y_train)
```

Listing 5.31 Split Data into Test and Train Sets, Then Create the Model

Now that we have a model, we'll create the predictions on the testing data and then evaluate how the model performs. In Listing 5.32, we're calculating two different approaches to measuring the model's value. The first is the accuracy score using the `accuracy_score` function. The second is the receiver operating characteristic (ROC) area under the curve (AUC)—say that 10 times fast!—using the `roc_auc_score` function.

```
#use the model to predict on the test data
y_pred = model.predict(X_test)

#calculate and print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

#calculate and print out the ROC AUC score
y_pred_proba = model.predict_proba(X_test)[:, 1]
auc_score = roc_auc_score(y_test, y_pred_proba)
print(f"ROC AUC Score: {auc_score:.4f}")
```

Listing 5.32 Create and Test Initial Logistic Regression Model

The results from each metric are different. We'll discuss metrics in more detail in Chapter 6, but here, accuracy represents how many times the model guesses either 1 or 0 correctly. The ROC AUC is more complex (and we'll get there in Chapter 6, Section 6.2.6), but anything below 0.7 is not a great model. The accuracy score we get when running the model is 85%, while the ROC AUC score is 0.6. This may seem confusing initially, but think about the data. Most people don't order again within a week, meaning a model that simply says that no one will order again in the next week would return what appears to be a good accuracy score. That's why the ROC AUC score is a better metric to leverage, given its ability to handle how our data is distributed.

Since we also calculated the target for repeat orders within 14 days, let's see what the model looks like when we use that for the target variable in Listing 5.33.

```
#select y as the 2 week target
y = df['target_14']

#execute the train test split
X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.3,
```

```
  random_state=42)

#create logistic regression model
model = LogisticRegression(solver='liblinear', random_state=42)

#fit the model to the data
model.fit(X_train, y_train)

#use the model to predict on the test data
y_pred = model.predict(X_test)

#calculate and print the accuracy metric
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

#calculate and print the roc auc metric
y_pred_proba = model.predict_proba(X_test)[:, 1]
auc_score = roc_auc_score(y_test, y_pred_proba)
print(f"ROC AUC Score: {auc_score:.4f}")
```

Listing 5.33 Rerun Using the 14-Day Target Variable Column

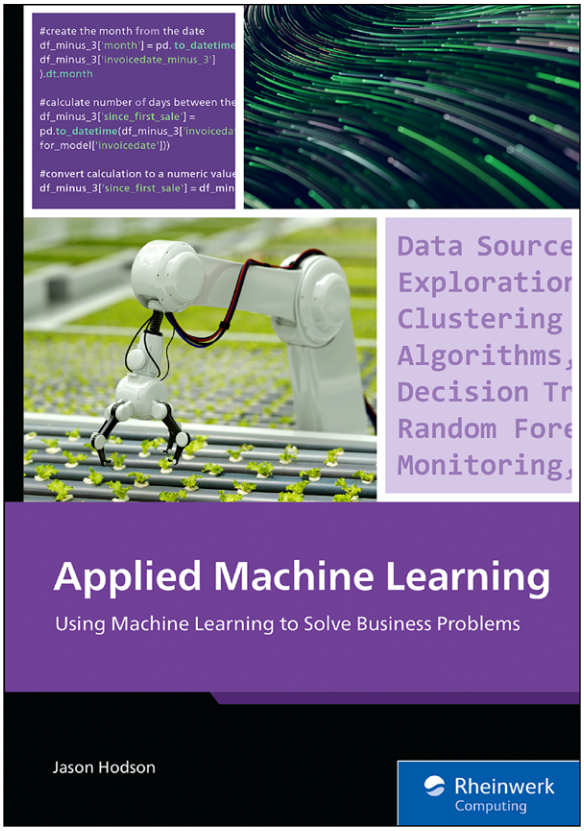
The results are even worse. The accuracy is 76%, and the ROC AUC score is 0.58. Neither of these models are likely to be valuable in practice. You may have noticed how quickly the model runs, which can be a significant benefit. However, it appears that the data may be more complicated than a logistic regression model can handle. Instead, we'll try out tree-based machine learning models to see how those algorithms handle this dataset.

Use Case 3: Logistic Regression

Since this use case calls for predicting a number of crimes rather than a probability, logistic regression isn't the proper model.

5.5 Machine Learning Models

This section covers what is more traditionally considered machine learning. We'll cover tree-based models in increasing complexity. We'll start with the simpler decision tree algorithm and then move to the random forest algorithm, which is a collection of decision trees. Lastly, we'll discuss GBM, a more complex algorithm in which decision trees learn from each other.



Jason Hodson

Applied Machine Learning

Using Machine Learning to Solve Business Problems

- Your practical introduction to applied machine learning
- Select and implement machine learning models to solve business problems
- Evaluate model results and monitor your models long term

www.sap-press.com/6170

The Author

Jason Hodson has worked in data-centric roles for nearly a decade, including HR analytics, forecasting, and more. He is an experienced developer of large data models and technical translator for stakeholders, peers, and team members.

ISBN 978-1-4932-2758-7 • 440 pages • 02/2026

E-book: $54.99 • **Print book:** $59.95 • **Bundle:** $69.99

Rheinwerk Publishing