# 8 Dealing with Common Data Problems

The ability to quickly assess the shortcomings of data and correct them can be the difference between being able to accomplish what you need to on time or falling behind. In this chapter, we're going to give you the tools to identify some of these problems, which you'll find are present in much of the data found in the industry.

We'll first look at when there can be too much data. This can be an issue where features can have an extremely high correlation with one another and in turn complicate a model. You'll see how to find this information and then remove the offending entries.

After that, we'll check into ways to get rid of blank, empty, or **Not a Number** (**NaN**) data that muddy the waters. This problem causes empty spaces without adding value.

We'll also look at what to do when you have categorical values. There are times when you'll need to maintain the relationship between categories, and times when you'll want to obfuscate them so that your model doesn't deduce any unnecessary relationship. We'll explore and examine both.

Moving on to feature limit caps, we'll cover what they are and how they can happen, but also how they can create bubbles in data that need to be sorted out before you can work effectively with it.

Finally, we'll look at the basics of data time manipulation, which you can use to slice and pull apart segments to just grab the parts that you need, such as pulling a month from a timestamp.

In this chapter, we will cover the following topics:

- Dealing with too much data
- Finding and correcting incorrect data entries
- Working with categorical values with one-hot encoding
- Feature scaling
- Working with date formats

Let's get started!

## Technical requirements

There are a few things that you will need to get the most out of this chapter. They are as follows:

- Anaconda Distribution. This includes conda and Navigator.You can download that from the following URL: `https://www.anaconda.com/products/distribution`
- A conda environment with `scikit-learn`, `pandas`, and `matplotlib`.
- A Jupyter notebook to perform all the coding segments. You also can use any IDE of choice or even the command line, but the assumption is that you will be working in a notebook.

After you have that set up, we can look at our first topic – how to deal with having extra data.

## Dealing with too much data

It's true that more data is usually better, but this isn't always the case. There are many times when having extra data has a negative impact on an outcome. Such a case was covered in *Chapter 1*, *Understanding the AI/ML Landscape*, where a father gave his child an extra example of what a tiger was, but that extra example was actually of a panther. That additional bit of information would then turn into a negative addition to the training set and create a worse learning outcome for your model.

How are you supposed to know this? Understand the data. This will be a common theme in this chapter, the book, and in the real world. If you don't start there, then everything else is more challenging. It's similar to being able to understand bias, as discussed in *Chapter 6*, *Overcoming Bias in AI/ML*.

Sometimes though, you won't or can't have a full grasp of the data, but you can use tools to help you out. The first clue that you can use is whether certain features of a training dataset have high correlations with one another.

### Checking feature correlation

Feature correlation is a great indicator to check whether there are potential overlaps in data. pandas gives you an easy way to do this, using the `corr()` function. Let's go through an example now, using a dataset that shows the recruiting rank for respective colleges in the sport of American college football.

The dataset will consist of the following features, which are represented in the columns in the dataset.

- **Rank**: The descending order of the team based on the recruiting points, as determined by `247Sports.com`
- **Ave**: The average rank for each recruit based on a 100-point scale
- **Points**: The total points for all recruits for that school
- **Wins**: The total wins for the school in 2021

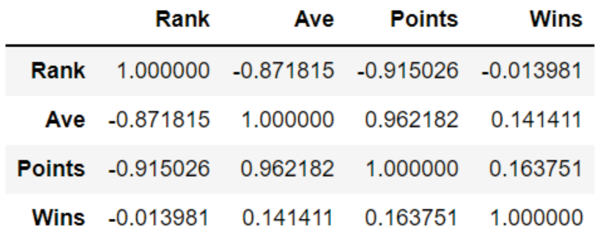
In your Jupyter notebook, first, make sure that pandas is installed and read in the NCAA data. **NCAA** stands for the **National College Athletic Association**:

```
import pandas as pd
df_ncaa = pd.read_csv('ncaa_data.csv')
```

Then, use the `corr()` function, which will check the correlation between all the columns in the given dataset. The output will be a table that shows a score from –1.0 to 1.0 for each column, showing how it correlates to each other column:

```
df_ncaa.corr()
```

This code will output the correlation table, where you can easily see the relationships between the potential features:



| | Rank | Ave | Points | Wins |
|---|---|---|---|---|
| **Rank** | 1.000000 | -0.871815 | -0.915026 | -0.013981 |
| **Ave** | -0.871815 | 1.000000 | 0.962182 | 0.141411 |
| **Points** | -0.915026 | 0.962182 | 1.000000 | 0.163751 |
| **Wins** | -0.013981 | 0.141411 | 0.163751 | 1.000000 |

Figure 8.1 – The NCAA correlation matrix

In this table, you can see that **1** shows up in all spots where a column relates to itself. This is to be expected, as the following breakdown shows:

- **1**: Perfectly positive correlation. An example might be the cost of fuel per week for your car and the liters/gallons of fuel used (assuming the cost per liter/gallon doesn't change).
- **0**: No correlation. There isn't a relationship at all. You might see this in the temperature in France versus how many books are bought in Texas on any given day.
- **-1**: Perfectly negative correlation. This could happen when you look at the speed of a car and the distance to the destination.

This is what is known as the **Pearson correlation coefficient**, and it's very rare, if ever, to see the previous exact numbers in the real world. If you do, then you will want to recheck that you aren't accidentally referencing and comparing the number to itself or have some other error. More than likely, you will have some decimal numbers in between the figures, indicating the relative strength of the correlation.

Taking the absolute value of the Pearson correlation, anything above .3 is generally considered to have some correlation, and anything above .8 has a strong relationship. Above .95 is incredibly strong.

Using what we just learned, let's look and see whether any of the columns might be redundant for our dataset. Taking a look at the **Points** row, we can see that there is a strong correlation between rank and average, with **-0.91** and **.96** respectively. It's easy to see why this would be the case if you consider the average gathered from the points, and the rank is a direct sort for the team with the highest recruiting point total.

Are all these columns needed? Probably not. It would be safe to remove the derived variables and just keep the source of truth, which is the raw point total. Let's do that now.

## Removing unnecessary columns in pandas

Recall that we can simply remove a column using the `drop()` command, which can be called on a pandas DataFrame. The first parameter is the column names, also known as the labels. This will be a list of strings you pass in. The second tells pandas that you want to drop columns, which is the *y* axis in a two-dimensional table, and officially the `1` axis. The `0` axis would be for dropping rows.

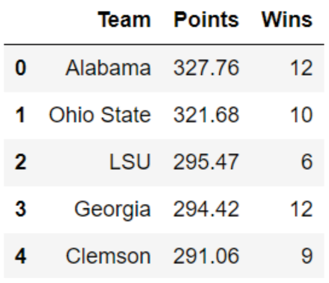
Here, we drop the `Rank` and `Ave` columns:

```
df_ncaa_slim = df_ncaa.drop(['Rank','Ave'],axis=1)
```

After this, you can check to confirm that we are left with only the data we want:

```
df_ncaa_slim.head()
```

The `head()` method will show us our new trimmed down dataset, as shown in *Figure 8.2*:



| | Team | Points | Wins |
|---|---|---|---|
| **0** | Alabama | 327.76 | 12 |
| **1** | Ohio State | 321.68 | 10 |
| **2** | LSU | 295.47 | 6 |
| **3** | Georgia | 294.42 | 12 |
| **4** | Clemson | 291.06 | 9 |

Figure 8.2 – The trimmed down NCAA dataset

You can see here that this allows us to work with just the features/columns that are useful to us.

You don't always want or need to remove things that have a high correlation; let's see a few ways that we might not want to drop these columns.

## Taking caution when removing columns

Be cautious of the famous *correlation doesn't equal causation* saying. This means that just because there might be a correlation between two things, there might not be a true relationship between them.

In our earlier examples, we uncovered that there was a direct connection between our two features, but that might not always be the case. Take the following, which is my favorite *correlation doesn't equal causation* example – higher ice cream sales correlate to more shark attacks.

Should stores stop selling ice cream to prevent attacks? Probably not. In this scenario, the likely cause could be the related factor of summertime, leading to more people cooling off with ice cream and hitting the beaches.

There are also cases where it's still useful to keep data separate even though there is a correlation. Take the example of the number of rooms in a house and the number of bedrooms. The number of bedrooms is a subset of the total rooms and also has a correlation. In this setting, it may prove valuable to keep both. Many times, the number of bedrooms will hit a limit, even as new rooms such as a theater, study, or game room add to the total number of rooms available.

You also need to be aware that some algorithms are able to deal with so-called multicollinearity, and you don't need to try and perform your own analysis. Approaches such as the random forest and lasso regression aren't impacted much by this analysis. It can be valuable to look at this analysis to understand the data and perhaps adjust it, but be careful with overcorrecting by thinking you need to remove potentially valuable data if there is any relationship shown.

Sometimes, there is the opposite of too much data, and that's when you see blanks or missing items. Let's see what our options are in that scenario.

## Working with missing values

Blank values are a part of many datasets, and there are many different ways this can come about. Maybe a value got deleted from the database, there was an issue with recording a value, or it could even be a legitimate situation where the value isn't applicable.

**NaN** stands for **Not a Number**. It is a special designation that pandas uses to tell us when a value is missing or unavailable. This is inherited from NumPy, and you might see them shown as NaN, NAN, or nan – all are equivalent. We'll use NaN in this book.

**pandas' NA Value**

pandas since version 1.0 has an experimental value of **NA** for missing values. The reason for this is to ensure a native and uniform missing value across all data types, but at the time of writing (version 1.4.2), this is still subject to change without warning, and as such, it won't be talked much about in this chapter.

We'll cover two scenarios where NaN isn't desired and another where it's a valid situation, where there shouldn't be data. First, we need to find these values.

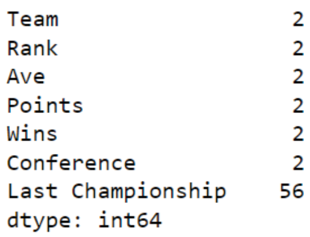
## Detecting NaN values

There is a quick way to detect whether the data we are working with has NaN values by using the `isnull()` method to get all the elements that are `null` and summing up the results with `sum()`. The following code will do just that:

```
df_ncaa.isnull().sum()
```

The first part, `df_ncaa.isnull()`, takes the `df_ncaa` DataFrame and returns a new Boolean array, with each field holding `True` or `False` respective of the corresponding element that is missing. That new Boolean array is then fed into `sum()`, which adds up the values for each column, with `False` equaling `0`, and `True` equaling `1`. In this case, it doesn't make much sense to separate it into different commands, which is why it's left as one.

This will output the number of NaN values that are present across each of the columns in our dataset, as shown in the following figure:



```
Team                  2
Rank                  2
Ave                   2
Points                2
Wins                  2
Conference            2
Last Championship    56
dtype: int64
```

Figure 8.3 – The total NaNs in the NCAA dataset

We can now see there are many under the **Last Championship** column and only **2** in the other columns. We can dive into those and see which rows have these NaNs.

## Dealing with valid NaN values

It might seem counter-intuitive to think of some NaNs being valid, but as always, think of the context of the dataset you are working in. Is it guaranteed that every college will have won a championship? Not at all. In that scenario, having a blank entry is valid. We don't want to do anything about those.

## Dealing with invalid NaN values

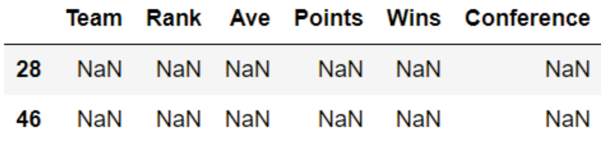
The other situation is when there are values that need to be removed. We can check which values have NaNs by first ignoring the **Last Championship** column, as we know it has many NaNs that are valid:

```
df = df_ncaa.iloc[:,0:-1]
```

We then use a masking technique to find just the rows where there are NaNs:

```
df[df.isna().any(axis=1)]
```

You'll see that there are two entries that consist of nothing but NaNs, which we can deduce are clearly there in error, as shown in the following output:



```
    Team  Rank  Ave  Points  Wins  Conference
28   NaN   NaN  NaN     NaN   NaN         NaN
46   NaN   NaN  NaN     NaN   NaN         NaN
```

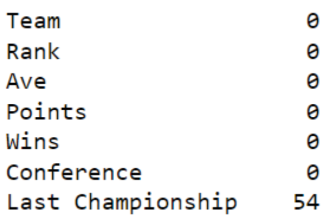
Figure 8.4 – The NaN rows in our data

We can then use the built-in methods to drop values that are NaNs, but if we do that, it will grab elements that have NaNs in our **Last Championship** column, which we know are valid. We don't want to discard that data just because the school hasn't won a championship.

We can get around this by setting the `how='all'` parameter to say that the criterion for dropping a row is when all the values are NaN. In that case, we can assume there was an error in the data entry. Per our previous discovery, rows 28 and 46 fit this criterion:

```
ncaa_clean = df_ncaa.dropna(how='all')
```

Checking again for NaN values, you can see that we've removed all NaN items from everything but the **Last Championship** column:



```
Team                  0
Rank                  0
Ave                   0
Points                0
Wins                  0
Conference            0
Last Championship    54
```

Figure 8.5 – The updated NaN values in the NCAA dataset

We've now cleaned up the NaN values, which gives us more confidence that we have the data needed in place to create models.

## Finding and correcting data entries

In the age of computers, human error will always come into play. Unfortunately, those mistaken keystrokes will manifest themselves in the datasets that we are tasked to work with. This will be present in everything from medical information to a car's service record.

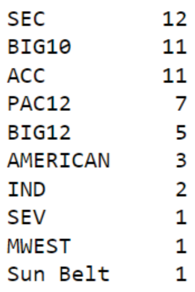
You can check for anomalies in a few ways; one is to simply group items together and see which stand out among the other items in that group. Looking back at our college football dataset, we want to confirm that the school's conferences are all correct.

We can simply call on the `Conference` column, which will be in a pandas series object. This object has many methods you can access, but the one we are interested in is pandas' `Series.value_counts()` method.

Let's use that to check whether there are lone conferences:

```
df_ncaa_error.Conference.value_counts()
```

This will show the following:



```
SEC         12
BIG10       11
ACC         11
PAC12        7
BIG12        5
AMERICAN     3
IND          2
SEV          1
MWEST        1
Sun Belt     1
```

Figure 8.6 – A count by conference

We can see that there are a few with just one record. From our knowledge about conferences, we know that **Sun Belt** and **MWest** (Mountain West) are valid, but **SEV** stands out as an issue. Calling on that specific conference will let us see which team that holds.

## Retrieving specific pandas items by condition

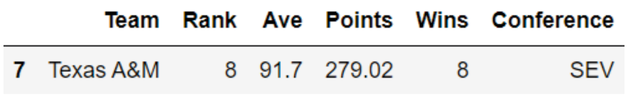
pandas has a particularly useful ability to let us grab specific items based on simple to complex conditions in the DataFrame. In our current example, we need to check which element has **SEV** as the conference. This is as simple as using a Boolean mask, as discussed in *Chapter 4*, *Working with Jupyter Notebooks and NumPy*. First, create a mask using a conditional that will satisfy what we are looking for:

```
mask = df_ncaa_error['Conference'] == 'SEV'
```

Then, apply that mask to the full DataFrame to give us the answer we need:

```
df_ncaa_error[mask]
```

This mask will then hide all the things we don't want to see while revealing the areas we are concerned about:



| | Team | Rank | Ave | Points | Wins | Conference |
|---|---|---|---|---|---|---|
| **7** | Texas A&M | 8 | 91.7 | 279.02 | 8 | SEV |

Figure 8.7 – Using a mask to show bad data

We know (or some quick research will tell us) that **A&M** is part of **SEC**, so we can be confident that the value **SEV** is a typo. We can easily make this correction using the `loc` method, which can not only locate an item via an index but also grab a specific column as well.

We already have our mask ready, so we can make use of it again here. We also want to specify that we want just the **Conference** column and that we'll set this to the corrected **SEC** conference. We'll print out the results using this same `loc` method. This will allow us to verify that it has been corrected:

```
df_ncaa_error.loc[mask, 'Conference'] = 'SEC'
```

```
df_ncaa_error.loc[7]
```

The results are as follows:

```
Team          Texas A&M
Rank                  8
Ave                91.7
Points           279.02
Wins                  8
Conference          SEC
Name: 7, dtype: object
```

Figure 8.8 – Using a mask to verify the updated data

It is now clear that we've successfully corrected the issue and can continue using our cleaned dataset with the correct conferences set up.

The conference itself is a categorical value, and AI models don't work well unless everything is a number. There are a few things to consider when making that conversion.

## Working with categorical values with one-hot encoding

Machine learning and statistics can be quite good at determining relationships between numbers. But what if you have a feature that is categorical and doesn't have a relationship? The definition of a **categorical feature** is when the variable is a label or category with discrete possibilities, such as colors , the animal kingdom, or cities.

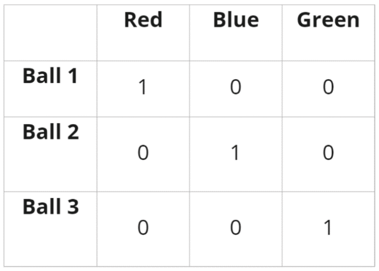
One option when you have this type of data is to use use **one-hot encoding**. This is the process of converting a categorical value into a set of ones and zeroes so that the model can interpret them as independent, but not infer that there is a relationship between them. This also prevents the inference that some categories are superior or inferior.

You can see an example of what this looks like in the following figure. Say you are looking at sales data for bouncy balls and one of the features is the color. There are three colors – red, blue and green. This is represented as data in the following table:

| | **Color** |
|---|---|
| **Ball 1** | Red |
| **Ball 2** | Blue |
| **Ball 3** | Green |

Figure 8.9 – The ball color categories

Since these categories can't be directly interpreted because they are simply text, you can use one-hot encoding to represent this data in a way that can be used to train a model. If you expand each color into its own column, you can use a binary indicator of ones and zeroes to represent each discrete category. Our ball example would look like the following table:



| | Red | Blue | Green |
|---|---|---|---|
| **Ball 1** | 1 | 0 | 0 |
| **Ball 2** | 0 | 1 | 0 |
| **Ball 3** | 0 | 0 | 1 |

Figure 8.10 – A one-hot encoding example

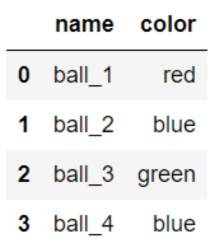
Here, you can see that the color of the ball can be determined simply by looking at which column has **1**. For **Ball 1**, the **Red** column is hot (represented as a **1**), and the rest have a dummy variable, which just means we've put zeroes in those spots. So, only one of the columns will ever be *hot*, hence the name of the encoding type. This terminology of dummy variables will be explored as we talk about how to make use of one-hot encoding a dataset in pandas.

## One-hot encoding with pandas

Let's get the one-hot encoded results from our ball color example using pandas. We will first recreate our simple DataFrame per *Figure 8.9* with one added ball to see the results more easily:

```
import pandas as pd
df = pd.DataFrame({
    'name': ['ball_1', 'ball_2','ball_3', 'ball_4'],
    'color': ['red', 'blue', 'green', 'blue']
    })
df
```

You will see the DataFrame, which will be very similar to the previous table:



| | name | color |
|---|---|---|
| 0 | ball_1 | red |
| 1 | ball_2 | blue |
| 2 | ball_3 | green |
| 3 | ball_4 | blue |

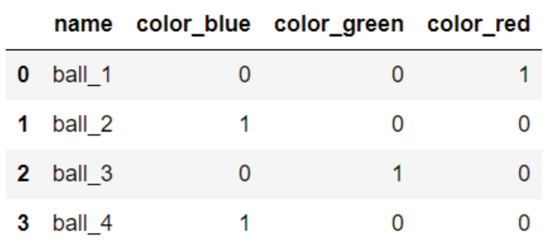
Figure 8.11 – A one-hot encoding example setup

Next, we will use the pandas `get_dummies` function to one-hot-encode our results. As we know, one-hot encoding generates mostly dummy values (zeros) in the expanded DataFrame, and this is where the name comes from. The `column` parameter specifies what we should encode, and the prefix is simply a string that will be prepended to make it easier to read:

```
one_hot_df = pd.get_dummies(df, prefix=['color'], columns = 
['color'])
```

```
one_hot_df
```

The result of this will be our one-hot encoded result. Note that **ball_4** has the same fields for its color fields as **ball_2**, since they are both **blue**:



| | name | color_blue | color_green | color_red |
|---|---|---|---|---|
| 0 | ball_1 | 0 | 0 | 1 |
| 1 | ball_2 | 1 | 0 | 0 |
| 2 | ball_3 | 0 | 1 | 0 |
| 3 | ball_4 | 1 | 0 | 0 |

Figure 8.12 – A one-hot encoding results

This result is what we expected, but there are some things to think about when deciding to use one-hot encoding .

## When to not use one-hot encoding

One-hot encoding isn't always the approach you want, and there are a few things you will want to consider.

You want to be cautious when there are exceptionally large numbers of categories or unique elements. Think back to our ball example with just three colors. What if instead of 3 possibilities we had 10,000? Then, what if we started with two color features, representing the primary and secondary colors? You would massively explode the dataset from a few columns to 20,000! This would significantly slow down the processing time it takes to train or analyze the model.

There is also another type of encoding you might want to consider when relationships come into play, which is ordinal encoding. We'll touch on this next.

## Ordinal encoding

For things that do, in fact, have a relationship, it might be better to not one-hot-encode. Variables that have an inherent ordering between them are referred to as **ordinal variables**. Examples include grades on a test or satisfaction ratings on a survey. These, in fact, do have subjective *better and worse* aspects to them, as a *D* grade is worse than an *A*, and *very dissatisfied* is worse than *very satisfied*.

If you have ordinal variables, it would be better to leave them in a format that leaves this relationship intact, even though they are a categorical rating. If the magnitude is relevant to the problem trying to be solved, then they might not bea viable candidate for one-hot encoding.

For example, the following shows the results of a college survey. All that was captured was the majors of the students and their satisfaction with the college. The rating scale was from 1 to 5:

| **Major** | **Answer** |
|---|---|
| Engineering | 3 |
| Business | 5 |
| Engineering | 4 |
| Business | 5 |

Figure 8.13 – Simple survey results

In this scenario, you could one-hot-encode the major, but you wouldn't want to use that approach on the answer. There is an objective ordering on the answers from 1 to 5, where 5 is better than 4, for example. We wouldn't want to lose that context by one-hot encoding.

## Feature scaling

When you are working with a large spread of numbers, the higher the deviation, the harder it will be to train a good model on them. This issue with deviation is for a number of reasons we won't cover now, but we'll cover scaling techniques more in depth in the Scaling the data section in *Chapter 9, Building a Regression Model with scikit-learn*. But you should know that sometimes you will come across datasets where someone has already scaled the data.

You can't always know where a dataset has come from, so you may not have the benefit of understanding why a particular decision was made.

This data could come from a colleague, a Kaggle competition, or it is just an example dataset included in `scikit-learn`, like the one we are using now. This is the same California training dataset that was used in *Chapter 2*, *Analyzing Open Source Software*, and we'll assume that you already have the `y_test` and `y_predict` setup. If not, refer back to *Chapter 2*, *Analyzing Open Source Software*.

Let's plot the training dataset to see where we are starting from. All of the following are the same as the previous examples, with the addition of the `alpha` parameter, which allows a floating-point number to indicate how transparent or opaque we want something to be in the graph. This will allow us to see density much more clearly. Usually, `.2` is a good value to start at:

```
import matplotlib.pyplot as plt
plt.title('Actual vs Predicted California Housing price')
plt.xlabel('Actual price')
plt.ylabel('Predicted price')
plt.scatter(y_test,y_predict,alpha=.2)
plt.plot([0, 5], [0, 5],"r-")
plt.show()
```

The preceding code block will output the following plot, showing the predicted versus the actual home value, with the darker spots showing greater density:

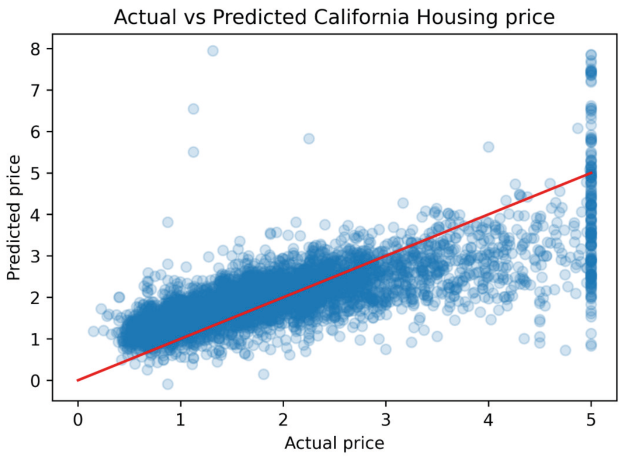


Figure 8.14 – A data anomaly from plotting the California dataset

This plot of the data shows a strange anomaly on the right, as you can see. Do you think the actual price of all those houses was exactly the same? Probably not. One issue is that the actual price is capped at 5.0, and yet the model might (correctly) predict that these units are above the scaled 5.0.

Another issue is the scale itself. You would be hard-pressed to find a house anywhere that is around five dollars. Let's unpack what could be going on by starting with another way to visualize the data.

## Creating a histogram with pandas

By using pandas' built-in histogram features, we can see whether there are some extreme values that might be an issue.

First, let's build a histogram of the values to see whether they follow a normal distribution. A **histogram** is a way to display data in which values are put into buckets of an equal range, and you then count how many items are in each bucket. This is also known as **binning**. We'll use a high number of bins, considering we have a broad range of possibilities for the price of the house. The more bins, the smaller the range that is in each bin.

We'll use the target of the California dataset and then call the `hist()` function, with the bin number set to `100`:

```
target_value.hist(bins = 100)
```

Using this, we see what we expected. There are many houses that have their price listed in the same bucket, as you can see on the far right of the following figure:

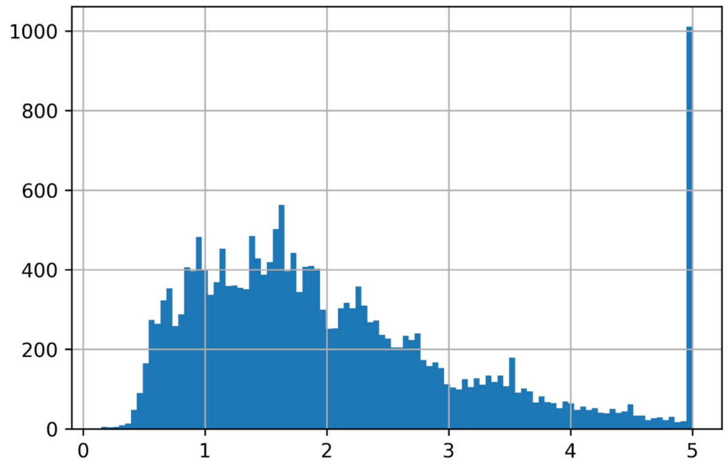


Figure 8.15 – The California home price histogram's original data

Feel free to play around with the `bins` number to see the resultant graph. Let's show a quick extreme example with only `5` bins:

```
target_value.hist(bins = 5)
```

The following is the resultant graph, which is the same as *Figure 8.15* but with much larger bin sizes, due to there only being five bins to put things into:

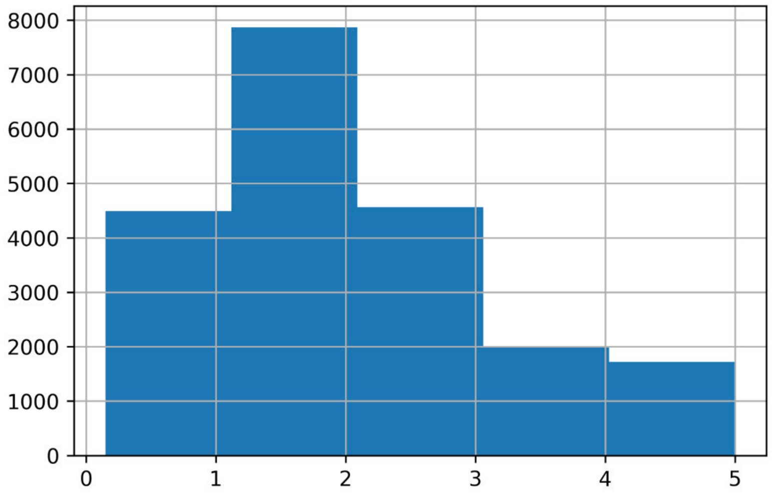


Figure 8.16 – The California home price histogram with a small bin number

It seems obvious that if we started with this small number of bins, we would have lost the valuable insight that there is an anomaly at the high end of the data.

Regarding the anomaly seen on the histogram with 100 bins in *Figure 8.15*, this is one of the downsides of capping data at a set limit. You are sacrificing some of the clarity of the pricing for the exact target you need to train on! Some of those houses could be two or three times that upper limit, but there is no way to know that.

There are many reasons that this can happen. Maybe there was a poll that had multiple choices, or maybe the researchers that gathered the data thought it would be better to bin the data so that it would be easier to work with. Whatever the case, we are losing pure information and need to see whether we can correct that.

Before we do much else, we need some objective way to measure whether what we are going to do actually improves the model. There are a few ways to do that for a regression problem.

## Using the R2 score to evaluate a model

One way to put a number on how good this model is when trained with default data is to use the R2 score. The R2 score is an indicator of how well our model does compared to if we simply took the mean of the target values and assumed that every house would sell for that:

- **R2 = 0.0**: The model performs exactly the same as simply taking the mean for every dependent variable value – in this case, the house price.
- **R2 = 1.0**: The model fits the data perfectly. This might not be desirable, as it could mean the model overfits the training data and might not perform well on data it hasn't seen before.
- **R2 < 0.0**: The model performs worse than if you just took the mean for everything. This is an indicator that the model needs a lot of work, the data does, or both.

Let's do a quick calculation on the R2 score here. We'll cover in more detail the `r2` score and other ways to evaluate a model's accuracy in *Chapter 9*, *Building a Regression Model with scikit-learn*.

We'll cover the same steps of creating a model as we did in previous chapters, so we'll skip the steps of importing the California test dataset and the `train/test` split. See *Chapter 2*, *Analyzing Open Source Software*, for a refresher on model creation.

Here, we'll focus on model training and getting the R2 score:

1. First, create a model, as we did previously:

```
from sklearn.linear_model import LinearRegression
linear_regressor = LinearRegression()
y_predict = linear_regressor.fit(X_train,y_train).
predict(X_test)
```

2. Then, use `sklearn`, which lets us easily get the `r2` score by using the predicted values and comparing them to the `test` values that we held out of the full dataset:

```
from sklearn.metrics import r2_score
r2_score(y_test, y_predict)
```

This will give us the result of `0.626`, which is okay, but let's see whether we can improve on that later in the chapter.

## Using the MSE score to evaluate a model

Along with the `r2` score, another common way to measure a regression model is with the Mean squared error. The **Mean Squared Error** (**MSE**) is simply the equation in which you take the mean of how far away the actual value is from the predicted value squared. The raw distance from the actual point to the predicted point is the error, hence the name *MSE*.

The following figure shows the equation for it:

$$MSE(y, y\hat{}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} (y_i - \hat{y}_i)^2$$

Figure 8.17 – The MSE equation

Here are the components of the previous equation:

- **Yi** = the actual value
- **Ŷi** = the predicted value
- **n** = the total number of data points

The MSE will give you a score where lower is better, which breaks down roughly as follows:

- **MSE = 0**: There is a perfect prediction across all the data points. Similar to getting a perfect R2 score, this can be very suspicious, as it might indicate that the data fits a little too well.
- **MSE = > 0**: Your predicated target isn't perfect and grows increasingly poor the higher this gets. One flaw in this measurement is that while you can compare two models to each other across the exact same dataset, there isn't a range of *pretty good* to *horrible*. In one problem or scenario, a MSE of 20 might be incredible, but in another, that same value could be wildly off from what the `true` values are. It all depends on the range of the target value.

The MSE is also simple to calculate using `sklearn`, and assuming you are using the same `y_test` and `y_predict` variables as you did earlier with R2, you can simply use the `mean_squared_error` method to calculate it, as we can see in the following:

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_predict)
```

This will give you the expected MSE, which in our case is the following:

```
0.61222
```

These R2 and MSE values are fine, but there is also one more way to evaluate our results that we'll talk about, which is the MAE.

## Using the MAE score to evaluate a model

The **Mean Absolute Error** (**MAE**) is very similar to the MSE, except that for each data point, you take the absolute value of the error instead of squaring it. This has the benefit of preventing massive numbers from forming if you are dealing with larger numbers, as well as preventing a skewed smaller number in the case of squaring a number less than one.

The equation is as follows:

$$\mathbf{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

Figure 8.18 – The MAE equation

Here are the components of the previous equation:

- **yi** = the actual value
- **xi** = the predicted value
- **n** = the total number of data points:

We can use the following code to calculate the MAE again using our same values as before:

```
from sklearn.metrics import mean_absolute_error
print(f"MAE is: {mean_absolute_error(y_test, y_predict)}")
```

This gives us a score of 0.5117 but we think we can improve the data points if we focus on removing the capped values we have on the median income. But what should we do about that?

## Overcoming the limits of capped values

One solution is to simply remove all data that hits the upper limit. This might be the safest way to ensure that we don't include any data that is capped at this ceiling but might, in fact, go much higher.

### Dropping pandas rows based on a condition

In our example, `y_train` is the training data representing the housing price. We need to clear out data that is above our threshold of five, as that is what we know hits that ceiling.

Set this criterion as the condition to check for, and then use that condition to create a Boolean representation of whether each element satisfies this criterion or not. Use the following code to do that:

```
target_cond = target_value < 5.0
target_cond
```

You'll see that this gives you your Boolean mask as follows:

```
0          True
1          True
2          True
           ...
20637      True
```

```
20638    True
20639    True
```

Since we know that the training features have the same number of rows due to this data matching up with the target value, we can now apply this mask to our training features:

```
filtered_training_data = training_data[target_cond]
```

This will give you a dataset of 19,648 rows × 6 columns to train with.

Let's also check the histogram for the new target values to make sure we have removed that top ceiling. We'll call the same histogram method as we did previously, just with the new subgroup of the target:

```
target_value[target_cond].hist(bins = 100)
```

As you can see, we have removed that large column at the end. This is what we want and expect:

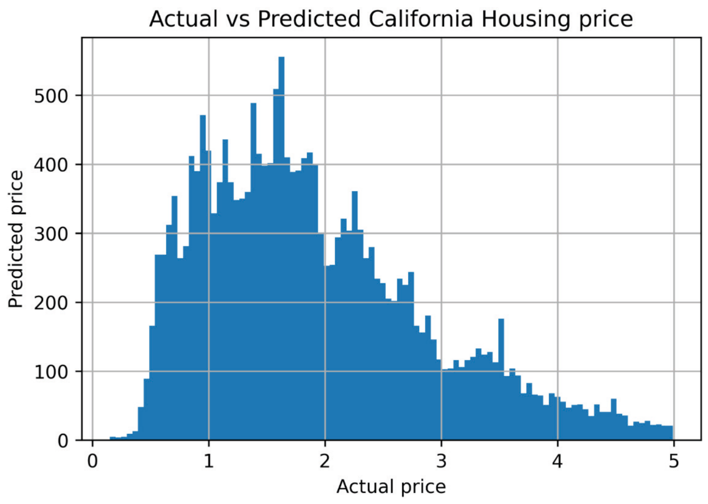


Figure 8.19 – The California home price histogram filtered

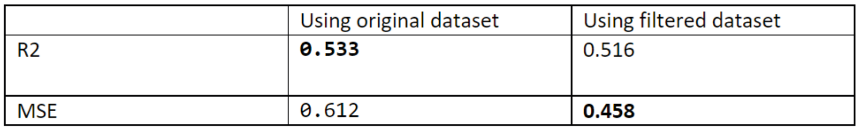
Now, let's train a model in the same way with this new data and check the R2 and MSE scores again:

```
X_train, X_test, y_train, y_test = train_test_split(filtered_
training_data, target_value[target_cond], test_size = 0.2,
random_state=1)
linear_regressor = LinearRegression()
y_predict = linear_regressor.fit(X_train,y_train).predict(X_
test)
print(f"Filtered Prediction")
print(f"R2 is: {r2_score(y_test, y_predict)}")
print(f"MSE is: {mean_squared_error(y_test, y_predict)}")
```

We'll see that this gives us both scores for this new dataset:

```
Filtered Prediction
R2 is: 0.5159223449106703
MSE is: 0.4582190705946869
```

Let's compare the new values to the old ones, using the values we already know:



| | Using original dataset | Using filtered dataset |
|---|---|---|
| R2 | **0.533** | 0.516 |
| MSE | 0.612 | **0.458** |

Figure 8.20 – Comparing the R2 and MSE scores

You might notice something odd. The original dataset that we believed was not particularly good has a slightly better **R2** score, but the filtered one has a much better **MSE** score!

There are three conclusions that we should take away from this:

- Cleaning data doesn't always guarantee a better model. There are a huge number of factors that go into whether a model will be able to accurately predict something, and changing one aspect might have unforeseen impacts.
- Looking at multiple scoring metrics is a good idea. There is no single clear way to know whether a model is good or not, and if you just looked at the `r2` score, you might not even try to improve upon things. Conversely, if you just used the MSE as a measurement, then you might lose the insight that removing this extra data might not be the home run you thought it was.
- Sometimes, you'll make mistakes. The insight you think you have about data might turn out to be wrong. Maybe, in this case, the houses showed accurate data, and people were just very hesitant to list their house at the $500,000 mark, similar to how people are more likely to buy products priced at $19.99 than $20.00 for psychological reasons.

In the end, the MSE for the filtered data is roughly a 25% improvement, which is a significant mark. We should stick with this one. The higher `r2` score might be due to the relatively few houses at the capped rate compared to the total, but those high sales numbers brought the average up.

However, what if we wanted to figure out how we got this price cap in the first place? You might want to try to get the original dataset or gather more information on the process used to construct it.

### Recovering the raw dataset

It might be a possibility to simply get back to the original dataset yourself and start from there. If you got this from a colleague, online, or some other source, it may be quite simple to track it down.

If the dataset was created by someone you know, you also might be able to talk to them to see whether it was the gathering methods or data manipulation afterward that arrived at this capped amount.

There might also be documentation in the repository that tells you what scaling was used and how to reverse it. It is possible that there was something more advanced that compressed the higher numbers, and thus you might have lost the ability to reverse the scaling.

## Working with date formats

Dates and times are often found in datasets and can present a few unique problems with data, becoming a huge thorn in a data scientist's side. There are many formats across the world, which differ across countries and systems. For example, the United States commonly uses the month/day/year format (mm/dd/yyyy), but in Europe, you are more likely to see day/month/year (dd/mm/yyyy).

Python has a built-in `datetime` object, but we'll make use of pandas' built-in `datetime` type as well. This will allow us to easily perform a few different operations on them, including grabbing just the month value, specifying a specific format, and other operations.

Time zones also come into play. There are many different rules across the world on what happens when. This is one reason UTC has become more common. UTC is a set standard that can be used no matter what your specific time zone is.

### Specifying a date field in pandas

The easiest way to call out a date is to specify it when reading data. Assuming you are using the `read_csv()` method, you can use the `parse_dates` parameter to tell pandas which column contains the data. We'll use our college dataset with the date that the college last won a championship.

Read in the CSV file and then check `info()` to verify that the second column is now correctly pulled in as a `datetime` object:

```
df_date_format = pd.read_csv('date_format.csv', parse_
dates=[1])
df_date_format.info()
```

You'll see that the second column is listed as a `datetime` object, as expected. Note that **64** just shows that this object is 64 bits; it's part of the internal workings of how pandas is set up, and you don't need to worry about it:

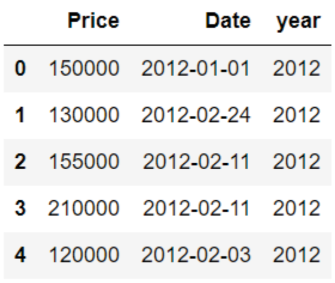
```
Data columns (total 2 columns):
#    Column   Non-Null Count   Dtype            -
--   ------   --------------   -----
0    Price    12 non-null      int64
1    Date     12 non-null      datetime64[ns]
dtypes: datetime64[ns](1), int64(1)
```

Now, say we just want the year, as the month and day aren't significant to understanding the data or answering the question we want. We can do this by creating a new column with just the year from this `data` object.

You can call on `dt` to access the `datetime` methods and from them, call on the year:

```
df_date_format['year'] = df_date_format['Date'].dt.year
df_date_format.head()
```

This will show you the following:



| | Price | Date | year |
|---|---|---|---|
| **0** | 150000 | 2012-01-01 | 2012 |
| **1** | 130000 | 2012-02-24 | 2012 |
| **2** | 155000 | 2012-02-11 | 2012 |
| **3** | 210000 | 2012-02-11 | 2012 |
| **4** | 120000 | 2012-02-03 | 2012 |

Figure 8.21 – Pulling out the year from pandas' datetime object

As you can now see, the year is in a clean column all by itself at the end.

### Converting string to dates with the to_datetime method

You won't always have the benefit of getting the dates in a clean data format, but there is a quick and easy way to convert them with pandas. The `to_datetime()` method will take in a single item, array-like, or a DataFrame object and convert it to a `datetime` object.

A quick example shows you how easy it is to use. Let's start with a list of dates such as the following:

```
dates = pd.Series(['09/10/1956', '7/05/1957', '9/08/1981',
'06/10/1983', '07/13/1987', '9/14/1990', '5/02/1992',
'10/08/1994'])
```

We then simply pass in this list along with the `infer_datetime_format=True` argument to signal that we want to infer what the format is from the input data, using the following code:

```
dates_df = pd.to_datetime(dates, infer_datetime_format=True)
```

We will now be delivered a DataFrame with the same elements that were passed in but consisting of the `datetime` type, which we can manipulate and change as we see fit.

There are many other operations we can perform after we have a `datetime` object. For a more complete list, see the official documentation here: `https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html`.

## Summary

Every situation and dataset you see will be unique; however, the problems you encounter with them won't be. In this chapter, you saw issues that will come up repeatedly with the datasets you'll be working with.

We saw how having too much data can be a problem by having highly correlated features, and how you can find that correlation and remove it. We used the example of college recruiting points and rank, but you can easily find others in the real world, such as housing prices – you might have the price per square footage but also have those as separate features.

Working with categorical data is common, but at the end of the day, machine learning models need numbers to be able to work. We saw that there are times when we want to keep relationships between categorical values, such as a rating system, and other times when we don't. We saw how we can use one-hot encoding to encode these categories when we don't want to keep the relationships.

Working with missing or blank values was another issue, and we looked at how we can isolate those items that show up as NaNs. We saw that some were not an issue, and there were those that can cause problems. For those that can be an issue, we checked out how we can remove elements that consist of nothing but NaN values so that we can avoid similar issues in the future.

Next, we looked at how we could isolate fields with incorrect data by finding entries in categorical columns that stood out on their own. This can give us an indicator when an entry wasn't intended; when the number of entries is high enough, there's a low probability that there would be a single entity in a category.

We also saw how we can detect capped values by looking at histograms and how pandas lets you conditionally remove elements. We saw how this can change the `r2` and MSE scores used to evaluate models and how to use exercise caution when giving either one of those too much weight.

Finally, we took a quick peek at working with `datetime` objects and how you can extract specific components of `datetime`, such as the month or the year.

Hopefully, with these tools, you will be able to decide when data needs to be cleaned up leading to errors being corrected more quickly. You now can increase your speed at correcting said issues, which gets you to your end goal of solving problems and creating models quicker.

In our next chapter, we are going to put together many of the things we have learned up to this point in this book to create a larger regression model using `scikit-learn`, along with being able to visualize the results.