```
#%% md
# 2 Data wrangling<a id='2_Data_wrangling'></a>
#%% md
## 2.1 Contents<a id='2.1_Contents'></a>
* [2 Data wrangling](#2_Data_wrangling)
  * [2.1 Contents](#2.1_Contents)
  * [2.2 Introduction](#2.2_Introduction)
    * [2.2.1 Recap Of Data Science Problem](#2.2.1_Recap_Of_Data_Science_Problem)
    * [2.2.2 Introduction To Notebook](#2.2.2_Introduction_To_Notebook)
  * [2.3 Imports](#2.3_Imports)
  * [2.4 Objectives](#2.4_Objectives)
  * [2.5 Load The Ski Resort Data](#2.5_Load_The_Ski_Resort_Data)
  * [2.6 Explore The Data](#2.6_Explore_The_Data)
    * [2.6.1 Find Your Resort Of Interest](#2.6.1_Find_Your_Resort_Of_Interest)
    * [2.6.2 Number Of Missing Values By Column](#2.6.2_Number_Of_Missing_Values_By_Column)
    * [2.6.3 Categorical Features](#2.6.3_Categorical_Features)
      * [2.6.3.1 Unique Resort Names](#2.6.3.1_Unique_Resort_Names)
      * [2.6.3.2 Region And State](#2.6.3.2_Region_And_State)
      * [2.6.3.3 Number of distinct regions and states](#2.6.3.3_Number_of_distinct_regions_and_states)
      * [2.6.3.4 Distribution Of Resorts By Region And State](#2.6.3.4_Distribution_Of_Resorts_By_Region_And_State)
      * [2.6.3.5 Distribution Of Ticket Price By State](#2.6.3.5_Distribution_Of_Ticket_Price_By_State)
        * [2.6.3.5.1 Average weekend and weekday price by state](#2.6.3.5.1_Average_weekend_and_weekday_price_by_state)
        * [2.6.3.5.2 Distribution of weekday and weekend price by state](#2.6.3.5.2_Distribution_of_weekday_and_weekend_price_by_state)
```

```
24     * [2.6.4 Numeric Features](#2.6.
   4_Numeric_Features)
25       * [2.6.4.1 Numeric data summary](#2.6.4.
   1_Numeric_data_summary)
26       * [2.6.4.2 Distributions Of Feature Values](#2.
   6.4.2_Distributions_Of_Feature_Values)
27         * [2.6.4.2.1 SkiableTerrain_ac](#2.6.4.2.
   1_SkiableTerrain_ac)
28         * [2.6.4.2.2 Snow Making_ac](#2.6.4.2.
   2_Snow_Making_ac)
29         * [2.6.4.2.3 fastEight](#2.6.4.2.3_fastEight)
30         * [2.6.4.2.4 fastSixes and Trams](#2.6.4.2.
   4_fastSixes_and_Trams)
31   * [2.7 Derive State-wide Summary Statistics For Our
    Market Segment](#2.7_Derive_State-
   wide_Summary_Statistics_For_Our_Market_Segment)
32   * [2.8 Drop Rows With No Price Data](#2.
   8_Drop_Rows_With_No_Price_Data)
33   * [2.9 Review distributions](#2.
   9_Review_distributions)
34   * [2.10 Population data](#2.10_Population_data)
35   * [2.11 Target Feature](#2.11_Target_Feature)
36     * [2.11.1 Number Of Missing Values By Row -
   Resort](#2.11.1_Number_Of_Missing_Values_By_Row_-
   _Resort)
37   * [2.12 Save data](#2.12_Save_data)
38   * [2.13 Summary](#2.13_Summary)
39
40 #%% md
41 ## 2.2 Introduction<a id='2.2_Introduction'></a>
42 #%% md
43 This step focuses on collecting your data, organizing
    it, and making sure it's well defined. Paying
   attention to these tasks will pay off greatly later
   on. Some data cleaning can be done at this stage, but
    it's important not to be overzealous in your
   cleaning before you've explored the data to better
   understand it.
44 #%% md
45 ### 2.2.1 Recap Of Data Science Problem<a id='2.2.
   1_Recap_Of_Data_Science_Problem'></a>
```

```
46 #%% md
47 The purpose of this data science project is to come
   up with a pricing model for ski resort tickets in our
    market segment. Big Mountain suspects it may not be
   maximizing its returns, relative to its position in
   the market. It also does not have a strong sense of
   what facilities matter most to visitors, particularly
    which ones they're most likely to pay more for. This
    project aims to build a predictive model for ticket
   price based on a number of facilities, or properties
   , boasted by resorts (*at the resorts).*
48 This model will be used to provide guidance for Big
   Mountain's pricing and future facility investment
   plans.
49 #%% md
50 ### 2.2.2 Introduction To Notebook<a id='2.2.
   2_Introduction_To_Notebook'></a>
51 #%% md
52 Notebooks grow organically as we explore our data. If
    you used paper notebooks, you could discover a
   mistake and cross out or revise some earlier work.
   Later work may give you a reason to revisit earlier
   work and explore it further. The great thing about
   Jupyter notebooks is that you can edit, add, and move
    cells around without needing to cross out figures or
    scrawl in the margin. However, this means you can
   lose track of your changes easily. If you worked in a
    regulated environment, the company may have a a
   policy of always dating entries and clearly crossing
   out any mistakes, with your initials and the date.
53
54 **Best practice here is to commit your changes using
   a version control system such as Git.** Try to get
   into the habit of adding and committing your files to
    the Git repository you're working in after you save
   them. You're are working in a Git repository, right?
   If you make a significant change, save the notebook
   and commit it to Git. In fact, if you're about to
   make a significant change, it's a good idea to commit
    before as well. Then if the change is a mess, you've
    got the previous version to go back to.
```

```
55
56 **Another best practice with notebooks is to try to
   keep them organized with helpful headings and
   comments.** Not only can a good structure, but
   associated headings help you keep track of what you'
   ve done and your current focus. Anyone reading your
   notebook will have a much easier time following the
   flow of work. Remember, that 'anyone' will most
   likely be you. Be kind to future you!
57
58 In this notebook, note how we try to use well
   structured, helpful headings that frequently are self
   -explanatory, and we make a brief note after any
   results to highlight key takeaways. This is an
   immense help to anyone reading your notebook and it
   will greatly help you when you come to summarise your
    findings. **Top tip: jot down key findings in a
   final summary at the end of the notebook as they
   arise. You can tidy this up later.** This is a great
   way to ensure important results don't get lost in the
    middle of your notebooks.
```

```
59 #%% md
60 In this, and subsequent notebooks, there are coding
   tasks marked with `#Code task n#` with code to
   complete. The `___` will guide you to where you need
   to insert code.
```

```
61 #%% md
62 ## 2.3 Imports<a id='2.3_Imports'></a>
```

```
63 #%% md
64 Placing your imports all together at the start of
   your notebook means you only need to consult one
   place to check your notebook's dependencies. By all
   means import something 'in situ' later on when you're
    experimenting, but if the imported dependency ends
   up being kept, you should subsequently move the
   import statement here with the rest.
```

```
65 #%%
66 #Code task 1#
67 #Import pandas, matplotlib.pyplot, and seaborn in the
    correct lines below
68 import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import jupyterlab
import jupyter
import sklearn
import lxml

from library.sb_utils import save_file

#%% md
## 2.4 Objectives<a id='2.4_Objectives'></a>
#%% md
There are some fundamental questions to resolve in this notebook before you move on.

* Do you think you may have the data you need to tackle the desired question?
    * Have you identified the required target value?
    * Do you have potentially useful features?
* Do you have any fundamental issues with the data?
#%% md
## 2.5 Load The Ski Resort Data<a id='2.5_Load_The_Ski_Resort_Data'></a>
#%%
# the supplied CSV data file is the raw_data directory
ski_data = pd.read_csv('../raw_data/ski_resort_data.csv')
#%% md
Good first steps in auditing the data are the info method and displaying the first few records with head.
#%%
#Code task 2#
#Call the info method on ski_data to see a summary of the data
ski_data.info
#%% md
`AdultWeekday` is the price of an adult weekday ticket. `AdultWeekend` is the price of an adult weekend ticket. The other columns are potential
```

```
 99 features.
100 #%% md
101 This immediately raises the question of what
    quantity will you want to model? You know you want
    to model the ticket price, but you realise there are
     two kinds of ticket price!
102 #%%
103 #Code task 3#
104 #Call the head method on ski_data to print the first
     several rows of the data
105 ski_data.head
106 #%% md
107 The output above suggests you've made a good start
    getting the ski resort data organized. You have
    plausible column headings. You can already see you
    have a missing value in the `fastEight` column
108 #%% md
109 ## 2.6 Explore The Data<a id='2.6_Explore_The_Data
    '></a>
110 #%% md
111 ### 2.6.1 Find Your Resort Of Interest<a id='2.6.
    1_Find_Your_Resort_Of_Interest'></a>
112 #%% md
113 Your resort of interest is called Big Mountain
    Resort. Check it's in the data:
114 #%%
115 #Code task 4#
116 #Filter the ski_data dataframe to display just the
    row for our resort with the name 'Big Mountain
    Resort'
117 #Hint: you will find that the transpose of the row
    will give a nicer output. DataFrame's do have a
118 #transpose method, but you can access this
    conveniently with the `T` property.
119 ski_data[ski_data['Name'] == 'Big Mountain Resort'].
    T
120 #%% md
121 It's good that your resort doesn't appear to have
    any missing values.
122 #%% md
123 ### 2.6.2 Number Of Missing Values By Column<a id='2
```

```
.6.2_Number_Of_Missing_Values_By_Column'></a>
#%% md
**Count the number of missing values in each column
** and sort them.
#%%
#Code task 5#
#Count (using `.sum()`) the number of missing values
 (`.isnull()`) in each column of
#ski_data as well as the percentages (using `.mean
()` instead of `.sum()`).
#Order them (increasing or decreasing) using
sort_values
#Call `pd.concat` to present these in a single table
 (DataFrame) with the helpful column names 'count'
and '%'
missing = pd.concat([ski_data.isnull().sum(), 100 *
ski_data.isnull().mean()], axis=1)
missing.columns=['count', '%']
missing.sort_values(by='count')
#%% md
`fastEight` has the most missing values, at just
over 50%. Unfortunately, you see you're also missing
 quite a few of your desired target quantity, the
ticket price, which is missing 15-16% of values. `
AdultWeekday` is missing in a few more records than
 `AdultWeekend`. What overlap is there in these
missing values? This is a question you'll want to
investigate. You should also point out that `isnull
()` is not the only indicator of missing data.
Sometimes 'missingness' can be encoded, perhaps by a
 -1 or 999. Such values are typically chosen because
 they are "obviously" not genuine values. If you
were capturing data on people's heights and weights
but missing someone's height, you could certainly
encode that as a 0 because no one has a height of
zero (in any units). Yet such entries would not be
revealed by `isnull()`. Here, you need a data
dictionary and/or to spot such values as part of
looking for outliers. Someone with a height of zero
should definitely show up as an outlier!
#%% md
```

```
138 ### 2.6.3 Categorical Features<a id='2.6.
    3_Categorical_Features'></a>
139 #%% md
140 So far you've examined only the numeric features.
    Now you inspect categorical ones such as resort name
     and state. These are discrete entities. 'Alaska' is
     a name. Although names can be sorted alphabetically
    , it makes no sense to take the average of 'Alaska'
    and 'Arizona'. Similarly, 'Alaska' is before '
    Arizona' only lexicographically; it is neither 'less
     than' nor 'greater than' 'Arizona'. As such, they
    tend to require different handling than strictly
    numeric quantities. Note, a feature _can_ be numeric
     but also categorical. For example, instead of
    giving the number of `fastEight` lifts, a feature
    might be `has_fastEights` and have the value 0 or 1
    to denote absence or presence of such a lift. In
    such a case it would not make sense to take an
    average of this or perform other mathematical
    calculations on it. Although you digress a little to
     make a point, month numbers are also, strictly
    speaking, categorical features. Yes, when a month is
     represented by its number (1 for January, 2 for
    Februrary etc.) it provides a convenient way to
    graph trends over a year. And, arguably, there is
    some logical interpretation of the average of 1 and
    3 (January and March) being 2 (February). However,
    clearly December of one years precedes January of
    the next and yet 12 as a number is not less than 1.
    The numeric quantities in the section above are
    truly numeric; they are the number of feet in the
    drop, or acres or years open or the amount of
    snowfall etc.
141 #%%
142 #Code task 6#
143 #Use ski_data's `select_dtypes` method to select
    columns of dtype 'object'
144 ski_data.select_dtypes('object')
145 #%% md
146 You saw earlier on that these three columns had no
    missing values. But are there any other issues with
```

```
these columns? Sensible questions to ask here include:

* Is `Name` (or at least a combination of Name/Region/State) unique?
* Is `Region` always the same as `state`?
#%% md
#### 2.6.3.1 Unique Resort Names<a id='2.6.3.1_Unique_Resort_Names'></a>
#%%
#Code task 7#
#Use pandas' Series method `value_counts` to find any duplicated resort names
ski_data['Name'].value_counts().head()
#%% md
You have a duplicated resort name: Crystal Mountain.
#%% md
**Q: 1** Is this resort duplicated if you take into account Region and/or state as well?
#%%
#Code task 8#
#Concatenate the string columns 'Name' and 'Region' and count the values again (as above)
(ski_data['Name'] + ', ' + ski_data['Region']).value_counts().head()
#%%
#Code task 9#
#Concatenate 'Name' and 'state' and count the values again (as above)
(ski_data['Name'] + ', ' + ski_data['state']).value_counts().head()
#%% raw
**NB** because you know `value_counts()` sorts descending, you can use the `head()` method and know the rest of the counts must be 1.
#%% md
**A: 1** No, Crystal Mountain is no longer duplicated when taking into account either region or state, otherwise it would appear in the first row.
#%%
ski_data[ski_data['Name'] == 'Crystal Mountain']
```

```
#%% md
So there are two Crystal Mountain resorts, but they are clearly two different resorts in two different states. This is a powerful signal that you have unique records on each row.
#%% md
#### 2.6.3.2 Region And State<a id='2.6.3.2_Region_And_State'></a>
#%% md
What's the relationship between region and state?
#%% md
You know they are the same in many cases (e.g. both the Region and the state are given as 'Michigan'). In how many cases do they differ?
#%%
#Code task 10#
#Calculate the number of times Region does not equal state
(ski_data.Region != ski_data.state).value_counts()
#%% md
You know what a state is. What is a region? You can tabulate the distinct values along with their respective frequencies using `value_counts()`.
#%%
ski_data['Region'].value_counts()
#%% md
A casual inspection by eye reveals some non-state names such as Sierra Nevada, Salt Lake City, and Northern California. Tabulate the differences between Region and state. On a note regarding scaling to larger data sets, you might wonder how you could spot such cases when presented with millions of rows. This is an interesting point. Imagine you have access to a database with a Region and state column in a table and there are millions of rows. You wouldn't eyeball all the rows looking for differences! Bear in mind that our first interest lies in establishing the answer to the question "Are they always the same?" One approach might be to ask the database to return records where they differ, but limit the output to 10 rows. If
```

```
191 there were differences, you'd only get up to 10
    results, and so you wouldn't know whether you'd
    located all differences, but you'd know that there
    were 'a nonzero number' of differences. If you got
    an empty result set back, then you would know that
    the two columns always had the same value. At the
    risk of digressing, some values in one column only
    might be NULL (missing) and different databases
    treat NULL differently, so be aware that on many an
    occasion a seamingly 'simple' question gets very
    interesting to answer very quickly!
192 #%%
193 #Code task 11#
194 #Filter the ski_data dataframe for rows where '
    Region' and 'state' are different,
195 #group that by 'state' and perform `value_counts` on
     the 'state'
196 (ski_data[ski_data['Region'] != ski_data['state']]
197  .groupby('state')['state']
198  .value_counts())
199 #%% md
200 The vast majority of the differences are in
    California, with most Regions being called Sierra
    Nevada and just one referred to as Northern
    California.
201 #%% md
202 #### 2.6.3.3 Number of distinct regions and states<a
     id='2.6.3.3_Number_of_distinct_regions_and_states
    '></a>
203 #%%
204 #Code task 12#
205 #Select the 'Region' and 'state' columns from
    ski_data and use the `nunique` method to calculate
206 #the number of unique values in each
207 ski_data[['Region', 'state']].nunique()
208 #%% md
209 Because a few states are split across multiple named
     regions, there are slightly more unique regions
    than states.
210 #%% md
211 #### 2.6.3.4 Distribution Of Resorts By Region And
```

```
State<a id='2.6.3.4_Distribution_Of_Resorts_By_Region_And_State'></a>
#%% md
If this is your first time using [matplotlib](https://matplotlib.org/3.2.2/index.html)'s [subplots](https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.subplots.html), you may find the online documentation useful.
#%%
#Code task 13#
#Create two subplots on 1 row and 2 columns with a figsize of (12, 8)
fig, ax = plt.subplots(1, 2, figsize=(12, 8))
#Specify a horizontal barplot ('barh') as kind of plot (kind=)
ski_data.Region.value_counts().sort_values(ascending=True).plot(kind='barh', ax=ax[0])
#Give the plot a helpful title of 'Region'
ax[0].set_title('Region')
#Label the xaxis 'Count'
ax[0].set_xlabel('Count')
#Specify a horizontal barplot ('barh') as kind of plot (kind=)
ski_data.state.value_counts().sort_values(ascending=True).plot(kind='barh', ax=ax[1])
#Give the plot a helpful title of 'state'
ax[1].set_title('State')
#Label the xaxis 'Count'
ax[1].set_xlabel('Count')
#Give the subplots a little "breathing room" with a wspace of 0.5
plt.subplots_adjust(wspace=0.5);
#You're encouraged to explore a few different figure sizes, orientations, and spacing here
# as the importance of easy-to-read and informative figures is frequently understated
# and you will find the ability to tweak figures invaluable later on
#%% md
How's your geography? Looking at the distribution of States, you see New York accounting for the
```

```
majority of resorts. Our target resort is in Montana, which comes in at 13th place. You should think carefully about how, or whether, you use this information. Does New York command a premium because of its proximity to population? Even if a resort's State were a useful predictor of ticket price, your main interest lies in Montana. Would you want a model that is skewed for accuracy by New York? Should you just filter for Montana and create a Montana-specific model? This would slash your available data volume. Your problem task includes the contextual insight that the data are for resorts all belonging to the same market share. This suggests one might expect prices to be similar amongst them. You can look into this. A boxplot grouped by State is an ideal way to quickly compare prices. Another side note worth bringing up here is that, in reality, the best approach here definitely would include consulting with the client or other domain expert. They might know of good reasons for treating states equivalently or differently. The data scientist is rarely the final arbiter of such a decision. But here, you'll see if we can find any supporting evidence for treating states the same or differently.
```

```
#%% md
#### 2.6.3.5 Distribution Of Ticket Price By State<a id='2.6.3.5_Distribution_Of_Ticket_Price_By_State'></a>
```

```
#%% md
Our primary focus is our Big Mountain resort, in Montana. Does the state give you any clues to help decide what your primary target response feature should be (weekend or weekday ticket prices)?
```

```
#%% md
##### 2.6.3.5.1 Average weekend and weekday price by state<a id='2.6.3.5.1_Average_weekend_and_weekday_price_by_state'></a>
```

```
#%%
#Code task 14#
# Calculate average weekday and weekend price by
```

```
245 state and sort by the average of the two
246 # Hint: use the pattern dataframe.groupby(<grouping
    variable>)[<list of columns>].mean()
247 state_price_means = ski_data.groupby('state')[['
    AdultWeekday', 'AdultWeekend']].mean()
248 state_price_means.head()
249 #%%
250 # The next bit simply reorders the index by
    increasing average of weekday and weekend prices
251 # Compare the index order you get from
252 # state_price_means.index
253 # with
254 # state_price_means.mean(axis=1).sort_values(
    ascending=False).index
255 # See how this expression simply sits within the
    reindex()
256 (state_price_means.reindex(index=state_price_means.
    mean(axis=1)
257     .sort_values(ascending=False)
258     .index)
259     .plot(kind='barh', figsize=(10, 10), title='
    Average ticket price by State'))
260 plt.xlabel('Price ($)');
261 #%% md
262 The figure above represents a dataframe with two
    columns, one for the average prices of each kind of
    ticket. This tells you how the average ticket price
    varies from state to state. But can you get more
    insight into the difference in the distributions
    between states?
263 #%% md
264 ##### 2.6.3.5.2 Distribution of weekday and weekend
    price by state<a id='2.6.3.5.
    2_Distribution_of_weekday_and_weekend_price_by_state
    '></a>
265 #%% md
266 Next, you can transform the data into a single
    column for price with a new categorical column that
    represents the ticket type.
267 #%%
268 #Code task 15#
```

```
#Use the pd.melt function, pass in the ski_data columns 'state', 'AdultWeekday', and 'Adultweekend' only,
#specify 'state' for `id_vars`
#gather the ticket prices from the 'Adultweekday' and 'AdultWeekend' columns using the `value_vars` argument,
#call the resultant price column 'Price' via the `value_name` argument,
#name the weekday/weekend indicator column 'Ticket' via the `var_name` argument

ticket_prices = pd.melt(ski_data[['state', 'AdultWeekday', 'AdultWeekend']],
                        id_vars='state',
                        var_name='Ticket',
                        value_vars=['AdultWeekday', 'AdultWeekend'],
                        value_name='Price')
#%%
ticket_prices.head()
#%% md
This is now in a format we can pass to [seaborn](https://seaborn.pydata.org/)'s [boxplot](https://seaborn.pydata.org/generated/seaborn.boxplot.html) function to create boxplots of the ticket price distributions for each ticket type for each state.
#%%
#Code task 16#
#Create a seaborn boxplot of the ticket price dataframe we created above,
#with 'state' on the x-axis, 'Price' as the y-value, and a hue that indicates 'Ticket'
#This will use boxplot's x, y, hue, and data arguments.

order = ticket_prices.groupby('state')['Price'].mean().sort_values(ascending=False).index

plt.subplots(figsize=(12, 8))
sns.boxplot(x='state', y='Price', hue='Ticket', data
```

```
=ticket_prices, order=order)
plt.xticks(rotation='vertical')
plt.ylabel('Price ($)')
plt.xlabel('State');
```

#%% md

Aside from some relatively expensive ticket prices in California, Colorado, and Utah, most prices appear to lie in a broad band from around 25 to over 100 dollars. Some States show more variability than others. Montana and South Dakota, for example, both show fairly small variability as well as matching weekend and weekday ticket prices. Nevada and Utah, on the other hand, show the most range in prices. Some States, notably North Carolina and Virginia, have weekend prices far higher than weekday prices. You could be inspired from this exploration to consider a few potential groupings of resorts, those with low spread, those with lower averages, and those that charge a premium for weekend tickets. However, you're told that you are taking all resorts to be part of the same market share, you could argue against further segment the resorts. Nevertheless, ways to consider using the State information in your modelling include:

* disregard State completely
* retain all State information
* retain State in the form of Montana vs not Montana, as our target resort is in Montana

You've also noted another effect above: some States show a marked difference between weekday and weekend ticket prices. It may make sense to allow a model to take into account not just State but also weekend vs weekday.

#%% md

Thus we currently have two main questions you want to resolve:

* What do you do about the two types of ticket price?

```
309 * What do you do about the state information?
310 #%% md
311 ### 2.6.4 Numeric Features<a id='2.6.
    4_Numeric_Features'></a>
312 #%% md
313 Having decided to reserve judgement on how exactly
    you utilize the State, turn your attention to
    cleaning the numeric features.
314 #%% md
315 #### 2.6.4.1 Numeric data summary<a id='2.6.4.
    1_Numeric_data_summary'></a>
316 #%%
317 #Code task 17#
318 #Call ski_data's `describe` method for a statistical
     summary of the numerical columns
319 #Hint: there are fewer summary stat columns than
    features, so displaying the transpose
320 #will be useful again
321 ski_data.describe().T
322 #%% md
323 Recall you're missing the ticket prices for some 16
    % of resorts. This is a fundamental problem that
    means you simply lack the required data for those
    resorts and will have to drop those records. But you
     may have a weekend price and not a weekday price,
    or vice versa. You want to keep any price you have.
324 #%%
325 missing_price = ski_data[['AdultWeekend', '
    AdultWeekday']].isnull().sum(axis=1)
326 missing_price.value_counts()/len(missing_price) *
    100
327 #%% md
328 Just over 82% of resorts have no missing ticket
    price, 3% are missing one value, and 14% are missing
     both. You will definitely want to drop the records
    for which you have no price information, however you
     will not do so just yet. There may still be useful
    information about the distributions of other
    features in that 14% of the data.
329 #%% md
330 #### 2.6.4.2 Distributions Of Feature Values<a id='2
```

```
.6.4.2_Distributions_Of_Feature_Values'></a>
#%% md
Note that, although we are still in the 'data wrangling and cleaning' phase rather than exploratory data analysis, looking at distributions of features is immensely useful in getting a feel for whether the values look sensible and whether there are any obvious outliers to investigate. Some exploratory data analysis belongs here, and data wrangling will inevitably occur later on. It's more a matter of emphasis. Here, we're interesting in focusing on whether distributions look plausible or wrong. Later on, we're more interested in relationships and patterns.
#%%
#Code task 18#
#Call ski_data's `hist` method to plot histograms of each of the numeric features
#Try passing it an argument figsize=(15,10)
#Try calling plt.subplots_adjust() with an argument hspace=0.5 to adjust the spacing
#It's important you create legible and easy-to-read plots
ski_data.hist(figsize=(15,10))
plt.subplots_adjust(hspace=0.5)
#Hint: notice how the terminating ';' "swallows" some messy output and leads to a tidier notebook
#%% md
What features do we have possible cause for concern about and why?

* SkiableTerrain_ac because values are clustered down the low end,
* Snow Making_ac for the same reason,
* fastEight because all but one value is 0 so it has very little variance, and half the values are missing,
* fastSixes raises an amber flag; it has more variability, but still mostly 0,
* trams also may get an amber flag for the same reason,
```

```
* yearsOpen because most values are low but it has a maximum of 2019, which strongly suggests someone recorded calendar year rather than number of years.
#%% md
##### 2.6.4.2.1 SkiableTerrain_ac<a id='2.6.4.2.1_SkiableTerrain_ac'></a>
#%%
#Code task 19#
#Filter the 'SkiableTerrain_ac' column to print the values greater than 10000
ski_data['SkiableTerrain_ac'][ski_data['SkiableTerrain_ac'] > 10000]
#%% md
**Q: 2** One resort has an incredibly large skiable terrain area! Which is it?
#%%
#Code task 20#
#Now you know there's only one, print the whole row to investigate all values, including seeing the resort name
#Hint: don't forget the transpose will be helpful here
ski_data[ski_data['SkiableTerrain_ac'] > 10000].T
#%% md
**A: 2** Silverton Mountain is listed as having a value of 26819.0
#%% md
But what can you do when you have one record that seems highly suspicious?
#%% md
You can see if your data are correct. Search for "silverton mountain skiable area". If you do this, you get some [useful information](https://www.google.com/search?q=silverton+mountain+skiable+area).
#%% md
![Silverton Mountain information](images/silverton_mountain_info.png)
#%% md
You can spot check data. You see your top and base elevation values agree, but the skiable area is very different. Your suspect value is 26819, but the
```

```
373 value you've just looked up is 1819. The last three
    digits agree. This sort of error could have occured
    in transmission or some editing or transcription
    stage. You could plausibly replace the suspect value
     with the one you've just obtained. Another
    cautionary note to make here is that although you're
     doing this in order to progress with your analysis
    , this is most definitely an issue that should have
    been raised and fed back to the client or data
    originator as a query. You should view this "data
    correction" step as a means to continue (documenting
     it carefully as you do in this notebook) rather
    than an ultimate decision as to what is correct.
374 #%%
375 #Code task 21#
376 #Use the .loc accessor to print the '
    SkiableTerrain_ac' value only for this resort
377 ski_data.loc[39, 'SkiableTerrain_ac']
378 #%%
379 #Code task 22#
380 #Use the .loc accessor again to modify this value
    with the correct value of 1819
381 ski_data.loc[39, 'SkiableTerrain_ac'] = 1819
382 #%%
383 #Code task 23#
384 #Use the .loc accessor a final time to verify that
    the value has been modified
385 ski_data.loc[39, 'SkiableTerrain_ac']
386 #%% md
387 **NB whilst you may become suspicious about your
    data quality, and you know you have missing values,
    you will not here dive down the rabbit hole of
    checking all values or web scraping to replace
    missing values.**
388 #%% md
389 What does the distribution of skiable area look like
     now?
390 #%%
391 ski_data.SkiableTerrain_ac.hist(bins=30)
392 plt.xlabel('SkiableTerrain_ac')
393 plt.ylabel('Count')
```

```
394 plt.title('Distribution of skiable area (acres)
    after replacing erroneous value');
395 #%% md
396 You now see a rather long tailed distribution. You
    may wonder about the now most extreme value that is
    above 8000, but similarly you may also wonder about
    the value around 7000. If you wanted to spend more
    time manually checking values you could, but leave
    this for now. The above distribution is plausible.
397 #%% md
398 ##### 2.6.4.2.2 Snow Making_ac<a id='2.6.4.2.
    2_Snow_Making_ac'></a>
399 #%%
400 ski_data['Snow Making_ac'][ski_data['Snow Making_ac'
    ] > 1000]
401 #%%
402 ski_data[ski_data['Snow Making_ac'] > 3000].T
403 #%% md
404 You can adopt a similar approach as for the suspect
    skiable area value and do some spot checking. To
    save time, here is a link to the website for [
    Heavenly Mountain Resort](https://www.skiheavenly.
    com/the-mountain/about-the-mountain/mountain-info.
    aspx). From this you can glean that you have values
    for skiable terrain that agree. Furthermore, you can
     read that snowmaking covers 60% of the trails.
405 #%% md
406 What, then, is your rough guess for the area covered
     by snowmaking?
407 #%%
408 .6 * 4800
409 #%% md
410 This is less than the value of 3379 in your data so
    you may have a judgement call to make. However,
    notice something else. You have no ticket pricing
    information at all for this resort. Any further
    effort spent worrying about values for this resort
    will be wasted. You'll simply be dropping the entire
     row!
411 #%% md
412 ##### 2.6.4.2.3 fastEight<a id='2.6.4.2.3_fastEight
```

```
'></a>
#%% md
Look at the different fastEight values more closely:
#%%
ski_data.fastEight.value_counts()
#%% md
Drop the fastEight column in its entirety; half the values are missing and all but the others are the value zero. There is essentially no information in this column.
#%%
#Code task 24#
#Drop the 'fastEight' column from ski_data. Use inplace=True
ski_data.drop(columns='fastEight', inplace=True)
#%% md
What about yearsOpen? How many resorts have purportedly been open for more than 100 years?
#%%
#Code task 25#
#Filter the 'yearsOpen' column for values greater than 100
ski_data['yearsOpen'][ski_data['yearsOpen'] > 100]
#%% md
Okay, one seems to have been open for 104 years. But beyond that, one is down as having been open for 2019 years. This is wrong! What shall you do about this?
#%% md
What does the distribution of yearsOpen look like if you exclude just the obviously wrong one?
#%%
#Code task 26#
#Call the hist method on 'yearsOpen' after filtering for values under 1000
#Pass the argument bins=30 to hist(), but feel free to explore other values
ski_data['yearsOpen'][ski_data['yearsOpen'] < 1000].hist(bins=30)
plt.xlabel('Years open')
plt.ylabel('Count')
```

```
440 plt.title('Distribution of years open excluding 2019
    ');
441 #%% md
442 The above distribution of years seems entirely
    plausible, including the 104 year value. You can
    certainly state that no resort will have been open
    for 2019 years! It likely means the resort opened in
     2019. It could also mean the resort is due to open
    in 2019. You don't know when these data were
    gathered!
443 #%% md
444 Let's review the summary statistics for the years
    under 1000.
445 #%%
446 ski_data.yearsOpen[ski_data.yearsOpen < 1000].
    describe()
447 #%% md
448 The smallest number of years open otherwise is 6.
    You can't be sure whether this resort in question
    has been open zero years or one year and even
    whether the numbers are projections or actual. In
    any case, you would be adding a new youngest resort
    so it feels best to simply drop this row.
449 #%%
450 ski_data = ski_data[ski_data.yearsOpen < 1000]
451 #%% md
452 ##### 2.6.4.2.4 fastSixes and Trams<a id='2.6.4.2.
    4_fastSixes_and_Trams'></a>
453 #%% md
454 The other features you had mild concern over, you
    will not investigate further. Perhaps take some care
     when using these features.
455 #%% md
456 ## 2.7 Derive State-wide Summary Statistics For Our
    Market Segment<a id='2.7_Derive_State-
    wide_Summary_Statistics_For_Our_Market_Segment'></a>
457 #%% md
458 You have, by this point removed one row, but it was
    for a resort that may not have opened yet, or
    perhaps in its first season. Using your business
    knowledge, you know that state-wide supply and
```

```
458 demand of certain skiing resources may well factor
    into pricing strategies. Does a resort dominate the
    available night skiing in a state? Or does it
    account for a large proportion of the total skiable
    terrain or days open?
459
460 If you want to add any features to your data that
    captures the state-wide market size, you should do
    this now, before dropping any more rows. In the next
     section, you'll drop rows with missing price
    information. Although you don't know what those
    resorts charge for their tickets, you do know the
    resorts exists and have been open for at least six
    years. Thus, you'll now calculate some state-wide
    summary statistics for later use.
461 #%% md
462 Many features in your data pertain to chairlifts,
    that is for getting people around each resort. These
     aren't relevant, nor are the features relating to
    altitudes. Features that you may be interested in
    are:
463
464 * TerrainParks
465 * SkiableTerrain_ac
466 * daysOpenLastYear
467 * NightSkiing_ac
468
469 When you think about it, these are features it makes
     sense to sum: the total number of terrain parks,
    the total skiable area, the total number of days
    open, and the total area available for night skiing
    . You might consider the total number of ski runs,
    but understand that the skiable area is more
    informative than just a number of runs.
470 #%% md
471 A fairly new groupby behaviour is [named aggregation
    ](https://pandas-docs.github.io/pandas-docs-travis/
    whatsnew/v0.25.0.html). This allows us to clearly
    perform the aggregations you want whilst also
    creating informative output column names.
472 #%%
```

```
473 #Code task 27#
474 #Add named aggregations for the sum of '
    daysOpenLastYear', 'TerrainParks', and '
    NightSkiing_ac'
475 #call them 'state_total_days_open', '
    state_total_terrain_parks', and '
    state_total_nightskiing_ac',
476 #respectively
477 #Finally, add a call to the reset_index() method (we
     recommend you experiment with and without this to
    see
478 #what it does)
479 state_summary = ski_data.groupby('state').agg(
480     resorts_per_state=pd.NamedAgg(column='Name',
    aggfunc='size'), #could pick any column here
481     state_total_skiable_area_ac=pd.NamedAgg(column='
    SkiableTerrain_ac', aggfunc='sum'),
482     state_total_days_open=pd.NamedAgg(column='
    daysOpenLastYear', aggfunc='sum'),
483     state_total_terrain_parks=pd.NamedAgg(column='
    TerrainParks', aggfunc='sum'),
484     state_total_nightskiing_ac=pd.NamedAgg(column='
    NightSkiing_ac', aggfunc='sum')
485 ).reset_index()
486 state_summary.head()
```

```
487 #%% md
488 ## 2.8 Drop Rows With No Price Data<a id='2.
    8_Drop_Rows_With_No_Price_Data'></a>
```

```
489 #%% md
490 You know there are two columns that refer to price
    : 'AdultWeekend' and 'AdultWeekday'. You can
    calculate the number of price values missing per row
    . This will obviously have to be either 0, 1, or 2,
    where 0 denotes no price values are missing and 2
    denotes that both are missing.
```

```
491 #%%
492 missing_price = ski_data[['AdultWeekend', '
    AdultWeekday']].isnull().sum(axis=1)
493 missing_price.value_counts()/len(missing_price) *
    100
```

```
494 #%% md
```

```
495 About 14% of the rows have no price data. As the
    price is your target, these rows are of no use. Time
     to lose them.
496 #%%
497 #Code task 28#
498 #Use `missing_price` to remove rows from ski_data
    where both price values are missing
499 ski_data = ski_data[missing_price != 2]
500 #%% md
501 ## 2.9 Review distributions<a id='2.
    9_Review_distributions'></a>
502 #%%
503 ski_data.hist(figsize=(15, 10))
504 plt.subplots_adjust(hspace=0.5);
505 #%% md
506 These distributions are much better. There are
    clearly some skewed distributions, so keep an eye on
     `fastQuads`, `fastSixes`, and perhaps `trams`.
    These lack much variance away from 0 and may have a
    small number of relatively extreme values.  Models
    failing to rate a feature as important when domain
    knowledge tells you it should be is an issue to look
     out for, as is a model being overly influenced by
    some extreme values. If you build a good machine
    learning pipeline, hopefully it will be robust to
    such issues, but you may also wish to consider
    nonlinear transformations of features.
507 #%% md
508 ## 2.10 Population data<a id='2.10_Population_data
    '></a>
509 #%% md
510 Population and area data for the US states can be
    obtained from [wikipedia](https://simple.wikipedia.
    org/wiki/List_of_U.S._states). Listen, you should
    have a healthy concern about using data you "found
    on the Internet". Make sure it comes from a
    reputable source. This table of data is useful
    because it allows you to easily pull and incorporate
     an external data set. It also allows you to proceed
     with an analysis that includes state sizes and
    populations for your 'first cut' model. Be explicit
```

```
510 about your source (we documented it here in this
    workflow) and ensure it is open to inspection. All
    steps are subject to review, and it may be that a
    client has a specific source of data they trust that
     you should use to rerun the analysis.
511 #%%
512 #Code task 29#
513 #Use pandas' `read_html` method to read the table
    from the URL below
514
515 #Downloaded to csv to avoid 404 errors
516
517 #import requests
518
519 #states_url = 'https://simple.wikipedia.org/w/index.
    php?title=List_of_U.S._states&oldid=7168473'
520 #headers = {"User-Agent": "Mozilla/5.0"}
521 #response = requests.get(states_url, headers=headers
    )
522
523 #usa_states_download = pd.read_html(response.text)
524 #%%
525 # save the data to a new csv file
526
527 #usa_states_download_df = usa_states_download[0]
528 #datapath = '../raw_data'
529 #save_file(usa_states_download_df, '
    usa_states_download.csv', datapath)
530 #%%
531 # load data from csv
532 usa_states = pd.read_csv('../raw_data/
    usa_states_download.csv')
533 #%%
534 type(usa_states)
535 #%%
536 len(usa_states)
537 #%%
538 #usa_states = usa_states[0]
539 usa_states.head()
540 #%% md
541 Note, in even the last year, the capability of `pd.
```

```
read_html()` has improved. The merged cells you see in the web table are now handled much more conveniently, with 'Phoenix' now being duplicated so the subsequent columns remain aligned. But check this anyway. If you extract the established date column, you should just get dates. Recall previously you used the `.loc` accessor, because you were using labels. Now you want to refer to a column by its index position and so use `.iloc`. For a discussion on the difference use cases of `.loc` and `.iloc` refer to the [pandas documentation](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html).
#%%
#Code task 30#
#Use the iloc accessor to get the pandas Series for column number 4 from `usa_states`
#It should be a column of dates
established = usa_states.iloc[:, 4]
#%%
established
#%% md
Extract the state name, population, and total area (square miles) columns.
#%%
#Code task 31#
#Now use the iloc accessor again to extract columns 0, 5, and 6 and the dataframe's `copy()` method
#Set the names of these extracted columns to 'state', 'state_population', and 'state_area_sq_miles',
#respectively.
usa_states_sub = usa_states.iloc[:, [0, 5, 6]].copy()
usa_states_sub.columns = ['state', 'state_population', 'state_area_sq_miles']
usa_states_sub.head()
#%% md
Do you have all the ski data states accounted for?
#%%
#Code task 32#
#Find the states in `state_summary` that are not in
```

```
563  `usa_states_sub`
564 #Hint: set(list1) - set(list2) is an easy way to get
     items in list1 that are not in list2
565 missing_states = set(state_summary.state) - set(
    usa_states_sub.state)
566 missing_states
```

```
567 #%% md
568 No??
```

```
569 #%% md
570 If you look at the table on the web, you can perhaps
     start to guess what the problem is. You can confirm
     your suspicion by pulling out state names that
    _contain_ 'Massachusetts', 'Pennsylvania', or '
    Virginia' from usa_states_sub:
```

```
571 #%%
572 usa_states_sub.state[usa_states_sub.state.str.
    contains('Massachusetts|Pennsylvania|Rhode Island|
    Virginia')]
```

```
573 #%% md
574 Delete square brackets and their contents and try
    again:
```

```
575 #%%
576 #Code task 33#
577 #Use pandas' Series' `replace()` method to replace
    anything within square brackets (including the
    brackets)
578 #with the empty string. Do this inplace, so you need
     to specify the arguments:
579 #to_replace='\[.*\]' #literal square bracket
    followed by anything or nothing followed by literal
    closing bracket
580 #value='' #empty string as replacement
581 #regex=True #we used a regex in our `to_replace`
    argument
582 #inplace=True #Do this "in place"
583 usa_states_sub.state.replace(to_replace='\[.*\]',
    value='', regex=True, inplace=True)
584 usa_states_sub.state[usa_states_sub.state.str.
    contains('Massachusetts|Pennsylvania|Rhode Island|
    Virginia')]
```

```
585 #%%
```

```
#Code task 34#
#And now verify none of our states are missing by checking that there are no states in
#state_summary that are not in usa_states_sub (as earlier using `set()`)
missing_states = set(state_summary.state) - set(usa_states_sub.state)
missing_states
```

```
#%% md
Better! You have an empty set for missing states now. You can confidently add the population and state area columns to the ski resort data.
```

```
#%%
#Code task 35#
#Use 'state_summary's `merge()` method to combine our new data in 'usa_states_sub'
#specify the arguments how='left' and on='state'
state_summary = state_summary.merge(usa_states_sub, how='left', on='state')
state_summary.head()
```

```
#%% md
Having created this data frame of summary statistics for various states, it would seem obvious to join this with the ski resort data to augment it with this additional data. You will do this, but not now. In the next notebook you will be exploring the data, including the relationships between the states. For that you want a separate row for each state, as you have here, and joining the data this soon means you'd need to separate and eliminate redundances in the state data when you wanted it.
```

```
#%% md
## 2.11 Target Feature<a id='2.11_Target_Feature'></a>
```

```
#%% md
Finally, what will your target be when modelling ticket price? What relationship is there between weekday and weekend prices?
```

```
#%%
#Code task 36#
#Use ski_data's `plot()` method to create a
```

```
scatterplot (kind='scatter') with 'AdultWeekday' on the x-axis and
#'AdultWeekend' on the y-axis
ski_data.plot(x='AdultWeekday', y='AdultWeekend', kind='scatter');
```

```
#%% md
A couple of observations can be made. Firstly, there is a clear line where weekend and weekday prices are equal. Weekend prices being higher than weekday prices seem restricted to sub $100 resorts. Recall from the boxplot earlier that the distribution for weekday and weekend prices in Montana seemed equal. Is this confirmed in the actual data for each resort? Big Mountain resort is in Montana, so the relationship between these quantities in this state are particularly relevant.
```

```
#%%
#Code task 37#
#Use the loc accessor on ski_data to print the 'AdultWeekend' and 'AdultWeekday' columns for Montana only
ski_data.loc[ski_data.state == "Montana", ["AdultWeekend", "AdultWeekday"]]
```

```
#%% md
Is there any reason to prefer weekend or weekday prices? Which is missing the least?
```

```
#%%
ski_data[['AdultWeekend', 'AdultWeekday']].isnull().sum()
```

```
#%% md
Weekend prices have the least missing values of the two, so drop the weekday prices and then keep just the rows that have weekend price.
```

```
#%%
ski_data.drop(columns='AdultWeekday', inplace=True)
ski_data.dropna(subset=['AdultWeekend'], inplace=True)
```

```
#%%
ski_data.shape
```

```
#%% md
Perform a final quick check on the data.
```

```
#%% md
### 2.11.1 Number Of Missing Values By Row - Resort<a id='2.11.1_Number_Of_Missing_Values_By_Row_-_Resort'></a>
#%% md
Having dropped rows missing the desired target ticket price, what degree of missingness do you have for the remaining rows?
#%%
missing = pd.concat([ski_data.isnull().sum(axis=1), 100 * ski_data.isnull().mean(axis=1)], axis=1)
missing.columns=['count', '%']
missing.sort_values(by='count', ascending=False).head(10)
#%% md
These seem possibly curiously quantized...
#%%
missing['%'].unique()
#%% md
Yes, the percentage of missing values per row appear in multiples of 4.
#%%
missing['%'].value_counts()
#%% md
This is almost as if values have been removed artificially... Nevertheless, what you don't know is how useful the missing features are in predicting ticket price. You shouldn't just drop rows that are missing several useless features.
#%%
ski_data.info()
#%% md
There are still some missing values, and it's good to be aware of this, but leave them as is for now.
#%% md
## 2.12 Save data<a id='2.12_Save_data'></a>
#%%
ski_data.shape
#%% md
Save this to your data directory, separately. Note that you were provided with the data in `raw_data`
```

```
and you should saving derived data in a separate location. This guards against overwriting our original data.
#%%
# save the data to a new csv file
datapath = '../data'
save_file(ski_data, 'ski_data_cleaned.csv', datapath)
#%%
# save the state_summary separately.
datapath = '../data'
save_file(state_summary, 'state_summary.csv', datapath)
#%% md
## 2.13 Summary<a id='2.13_Summary'></a>
#%% md
**Q: 3** Write a summary statement that highlights the key processes and findings from this notebook. This should include information such as the original number of rows in the data, whether our own resort was actually present etc. What columns, if any, have been removed? Any rows? Summarise the reasons why. Were any other issues found? What remedial actions did you take? State where you are in the project. Can you confirm what the target feature is for your desire to predict ticket price? How many rows were left in the data? Hint: this is a great opportunity to reread your notebook, check all cells have been executed in order and from a "blank slate" (restarting the kernel will do this), and that your workflow makes sense and follows a logical pattern. As you do this you can pull out salient information for inclusion in this summary. Thus, this section will provide an important overview of "what" and "why" without having to dive into the "how" or any unproductive or inconclusive steps along the way.
#%% md
**A: 3** Your answer here
```