In this guided project, we'll work with a dataset of used cars from *eBay Kleinanzeigen*, a [classifieds](https://en.wikipedia.org/wiki/Classified_advertising) section of the German eBay website.

The dataset was originally [scraped](https://en.wikipedia.org/wiki/Web_scraping) and uploaded to Kaggle by user [orgesleka](https://www.kaggle.com/orgesleka).  
The original dataset isn't available on Kaggle anymore, but you can find it [here](https://data.world/data-society/used-cars-data).

We've made a few modifications from the original dataset:

* We sampled 50,000 data points from the full dataset, to ensure your code runs quickly in our hosted environment
* We dirtied the dataset a bit to more closely resemble what you would expect from a scraped dataset (the version uploaded to Kaggle was cleaned to be easier to work with)

The data dictionary provided with data is as follows:

* dateCrawled - When this ad was first crawled. All field-values are taken from this date.
* name - Name of the car.
* seller - Whether the seller is private or a dealer.
* offerType - The type of listing
* price - The price on the ad to sell the car.
* abtest - Whether the listing is included in an A/B test.
* vehicleType - The vehicle Type.
* yearOfRegistration - The year in which the car was first registered.
* gearbox - The transmission type.
* powerPS - The power of the car in PS.
* model - The car model name.
* kilometer - How many kilometers the car has driven.
* monthOfRegistration - The month in which the car was first registered.
* fuelType - What type of fuel the car uses.
* brand - The brand of the car.
* notRepairedDamage - If the car has a damage which is not yet repaired.
* dateCreated - The date on which the eBay listing was created.
* nrOfPictures - The number of pictures in the ad.
* postalCode - The postal code for the location of the vehicle.
* lastSeenOnline - When the crawler saw this ad last online.

The aim of this project is to clean the data and analyze the included used car listings. You'll also become familiar with some of the unique benefits jupyter notebook provides for pandas.

Let's start by importing the libraries we need and reading the dataset into pandas.

Instructions

* Start by writing a paragraph in a markdown cell introducing the project and the dataset.
* Import the pandas and NumPy libraries
* Read the autos.csv CSV file into pandas, and assign it to the variable name autos.
  + Try without specifying any encoding (which will default to UTF-8)
  + If you get an encoding error, try the next two most popular encodings (Latin-1 and Windows-1252) until you are able to read the file without error.
* Create a new cell with just the variable autos and run this cell.
  + A neat feature of jupyter notebook is its ability to render the first few and last few values of any pandas object.
* Use the DataFrame.info() and DataFrame.head() methods to print information about the autos dataframe, as well as the first few rows.
  + Write a markdown cell briefly describing your observations.

From the work we did in the last screen, we can make the following observations:

* The dataset contains 20 columns, most of which are strings.
* Some columns have null values, but none have more than ~20% null values.
* The column names use [camelcase](https://en.wikipedia.org/wiki/Camel_case) instead of Python's preferred [snakecase](https://en.wikipedia.org/wiki/Snake_case), which means we can't just replace spaces with underscores.

Let's convert the column names from camelcase to snakecase and reword some of the column names based on the data dictionary to be more descriptive.

Instructions

* Use the DataFrame.columns attribute to print an array of the existing column names.
* Copy that array and make the following edits to columns names:
  + yearOfRegistration to registration\_year
  + monthOfRegistration to registration\_month
  + notRepairedDamage to unrepaired\_damage
  + dateCreated to ad\_created
  + The rest of the column names from camelcase to snakecase.
* Assign the modified column names back to the DataFrame.columns attribute.
* Use DataFrame.head() to look at the current state of the autos dataframe.
* Write a markdown cell explaining the changes you made and why.

Now let's do some basic data exploration to determine what other cleaning tasks need to be done. Initially we will look for:

* Text columns where all or almost all values are the same. These can often be dropped as they don't have useful information for analysis.
* Examples of numeric data stored as text which can be cleaned and converted.

The following methods are helpful for exploring the data:

* DataFrame.describe() (with include='all' to get both categorical and numeric columns)
* Series.value\_counts() and Series.head() if any columns need a closer look.

Instructions

* Use DataFrame.describe() to look at descriptive statistics for all columns.
* Write a markdown cell noting:
  + Any columns that have mostly one value that are candidates to be dropped
  + Any columns that need more investigation.
  + Any examples of numeric data stored as text that needs to be cleaned.
* If you need to investigate any columns more, do so and write up any additional things you found.
* You likely found that the price and odometer columns are numeric values stored as text. For each column:
  + Remove any non-numeric characters.
  + Convert the column to a numeric dtype.
  + Use DataFrame.rename() to rename the column to odometer\_km.

From the last screen, we learned that there are a number of text columns where almost all of the values are the same (seller and offer\_type). We also converted the price and odometer columns to numeric types and renamed odometer to odometer\_km.

Let's continue exploring the data, specifically looking for data that doesn't look right. We'll start by analyzing the odometer\_km and price columns. Here's the steps we'll take:

* Analyze the columns using minimum and maximum values and look for any values that look unrealistically high or low (outliers) that we might want to remove.
* We'll use:
  + Series.unique().shape to see how many unique values
  + Series.describe() to view min/max/median/mean etc
  + Series.value\_counts(), with some variations:
    - chained to .head() if there are lots of values.
    - Because Series.value\_counts() returns a series, we can use Series.sort\_index()with ascending= True or False to view the highest and lowest values with their counts (can also chain to head() here).
  + When removing outliers, we can do df[(df["col"] >= x ) & (df["col"] <= y )], but it's more readable to use df[df["col"].between(x,y)]

Instructions

* For each of the odometer\_km and price columns:
  + Use the techniques above to explore the data
  + If you find there are outliers, remove them and write a markdown paragraph explaining your decision.
  + After you have removed the outliers, make some observations about the remaining values.

Let's now move on to the date columns and understand the date range the data covers.

There are 5 columns that should represent date values. Some of these columns were created by the crawler, some came from the website itself. We can differentiate by referring to the data dictionary:

- `date\_crawled`: added by the crawler

- `last\_seen`: added by the crawler

- `ad\_created`: from the website

- `registration\_month`: from the website

- `registration\_year`: from the website

Right now, the date\_crawled, last\_seen, and ad\_created columns are all identified as string values by pandas. Because these three columns are represented as strings, we need to convert the data into a numerical representation so we can understand it quantitatively. The other two columns are represented as numeric values, so we can use methods like Series.describe() to understand the distribution without any extra data processing.

Let's first understand how the values in the three string columns are formatted. These columns all represent full timestamp values, like so:

autos[['date\_crawled','ad\_created','last\_seen']][0:5]

|  |  |  |  |
| --- | --- | --- | --- |
|  | **date\_crawled** | **ad\_created** | **last\_seen** |
| 0 | 2016-03-26 17:47:46 | 2016-03-26 00:00:00 | 2016-04-06 06:45:54 |
| 1 | 2016-04-04 13:38:56 | 2016-04-04 00:00:00 | 2016-04-06 14:45:08 |
| 2 | 2016-03-26 18:57:24 | 2016-03-26 00:00:00 | 2016-04-06 20:15:37 |
| 3 | 2016-03-12 16:58:10 | 2016-03-12 00:00:00 | 2016-03-15 03:16:28 |
| 4 | 2016-04-01 14:38:50 | 2016-04-01 00:00:00 | 2016-04-01 14:38:50 |

You'll notice that the first 10 characters represent the day (e.g. 2016-03-12). To understand the date range, we can extract just the date values, use Series.value\_counts() to generate a distribution, and then sort by the index.

To select the first 10 characters in each column, we can use Series.str[:10]:

print(autos['date\_crawled'].str[:10])

0       2016-03-26

1       2016-04-04

2       2016-03-26

3       2016-03-12

...

Instructions

* Use the workflow we just described to calculate the distribution of values in the date\_crawled, ad\_created, and last\_seen columns (all string columns) as percentages.
  + To include missing values in the distribution and to use percentages instead of counts, chain the Series.value\_counts(normalize=True, dropna=False) method.
  + To rank by date in ascending order (earliest to latest), chain the Series.sort\_index()method.
  + Write a markdown cell after each column exploration to explain your observations.
* Use Series.describe() to understand the distribution of registration\_year.
  + Write a markdown cell explaining your observations.

One thing that stands out from the exploration we did in the last screen is that the registration\_yearcolumn contains some odd values:

* The minimum value is 1000, before cars were invented
* The maximum value is 9999, many years into the future

Because a car can't be first registered after the listing was seen, any vehicle with a registration year above 2016 is definitely inaccurate. Determining the earliest valid year is more difficult. Realistically, it could be somewhere in the first few decades of the 1900s.

Let's count the number of listings with cars that fall outside the 1900 - 2016 interval and see if it's safe to remove those rows entirely, or if we need more custom logic.

Instructions

* Decide which the highest and lowest acceptable values are for the registration\_year column.
  + Write a markdown cell explaining your decision and why.
* Remove the values outside those upper and lower bounds and calculate the distribution of the remaining values using Series.value\_counts(normalize=True).
  + Write a markdown cell explaining your observations.

One of the analysis techniques we learned in this course is aggregation. When working with data on cars, it's natural to explore variations across different car brands. We can use aggregation to understand the brand column.

If you recall in an earlier mission, we explored how to use loops to perform aggregation. Here's what the process looks like:

* Identify the unique values we want to aggregate by
* Create an empty dictionary to store our aggregate data
* Loop over the unique values, and for each: - Subset the dataframe by the unique values - Calculate the mean of whichever column we're interested in - Assign the val/mean to the dict as k/v.

Instructions

* Explore the unique values in the brand column, and decide on which brands you want to aggregate by.
  + You might want to select the top 20, or you might want to select those that have over a certain percentage of the total values (e.g. > 5%).
  + Remember that Series.value\_counts() produces a series with index labels, so you can use Series.index attribute to access the labels, should you wish.
* Write a short paragraph describing the brand data, and explaining which brands you've chosen to aggregate on.
* Create an empty dictionary to hold your aggregate data.
  + Loop over your selected brands, and assign the mean price to the dictionary, with the brand name as the key.
  + Print your dictionary of aggregate data, and write a paragraph analyzing the results.

In the last screen, we aggregated across brands to understand mean price. We observed that in the top 6 brands, there's a distinct price gap.

* Audi, BMW and Mercedes Benz are more expensive
* Ford and Opel are less expensive
* Volkswagen is in between

For the top 6 brands, let's use aggregation to understand the average mileage for those cars and if there's any visible link with mean price. While our natural instinct may be to display both aggregated series objects and visually compare them, this has a few limitations:

* it's difficult to compare more than two aggregate series objects if we want to extend to more columns
* we can't compare more than a few rows from each series object
* we can only sort by the index (brand name) of both series objects so we can easily make visual comparisons

Instead, we can combine the data from both series objects into a single dataframe (with a shared index) and display the dataframe directly. To do this, we'll need to learn two pandas methods:

* [pandas series constructor](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html)
* [pandas dataframe constructor](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html)

Here's an example of the series constructor that uses the brand\_mean\_prices dictionary:

bmp\_series = pd.Series(brand\_mean\_prices)

print(bmp\_series)

volkswagen       5402

bmw             8332

opel             2975

mercedes\_benz   8628

audi             9336

ford             3749

dtype: int64

The keys in the dictionary became the index in the series object. We can then create a single-column dataframe from this series object. We need to use the columns parameter when calling the dataframe constructor (which accepts an array-like object) to specify the column name (or the column name will be set to 0 by default):

df = pd.DataFrame(bmp\_series, columns=['mean\_price'])

df

|  |  |
| --- | --- |
|  | **mean\_price** |
| volkswagen | 5402 |
| bmw | 8332 |
| opel | 2975 |
| mercedes\_benz | 8628 |
| audi | 9336 |
| ford | 3749 |

Instructions

* Use the loop method from the last screen to calculate the mean mileage and mean price for each of the top brands, storing the results in a dictionary.
* Convert both dictionaries to series objects, using the series constructor.
* Create a dataframe from the first series object using the dataframe constructor.
* Assign the other series as a new column in this dataframe.
* Pretty print the dataframe, and write a paragraph analyzing the aggregate data.

In this guided project, we practiced applying a variety of pandas methods to explore and understand a data set on car listings. Here are some next steps for you to consider:

* Data cleaning next steps:
  + Identify categorical data that uses german words, translate them and map the values to their english counterparts
  + Convert the dates to be uniform numeric data, so "2016-03-21" becomes the integer 20160321.
  + See if there are particular keywords in the name column that you can extract as new columns
* Analysis next steps:
  + Find the most common brand/model combinations
  + Split the odometer\_km into groups, and use aggregation to see if average prices follows any patterns based on the mileage.
  + How much cheaper are cars with damage than their non-damaged counterparts?

Curious to see what other students have done on this project? [Head over to our Community to check them out](https://community.dataquest.io/tags/c/social/share/49/294). While you are there, please remember to show some love and give your own feedback!

And of course, we welcome you to share your own project and show off your hard work. Head over to our Community to [share your finished Guided Project](https://community.dataquest.io/tags/c/social/share/49/294)!