An Essential Guide to Exploratory Data Analysis and Data Cleaning

Nick Vega Follow
Mar 2 · 11 min read ★

It is common for data scientists to spend a majority of their time exploring and cleaning data, but approaching this as an opportunity to invest in your model (instead of viewing it as just another chore on your to-do list) will yield big dividends later on in the data science process.

Performing thorough exploratory data analysis (EDA) and cleaning the dataset are not only essential steps, but also a great opportunity to lay the foundation for a strong machine learning model.

I used the Ames housing data set to predict home sale prices given a diverse range of features. In this post, I will cover different techniques I applied to explore and clean the data and in a subsequent post, I will cover how I applied different modeling techniques to substantially improve my performance.

· · ·

Overview of the Data

The Ames Housing Dataset is publicly available on Kaggle and is a great dataset to develop your skills. The data set also includes a data dictionary that provides a high-level description of each column.

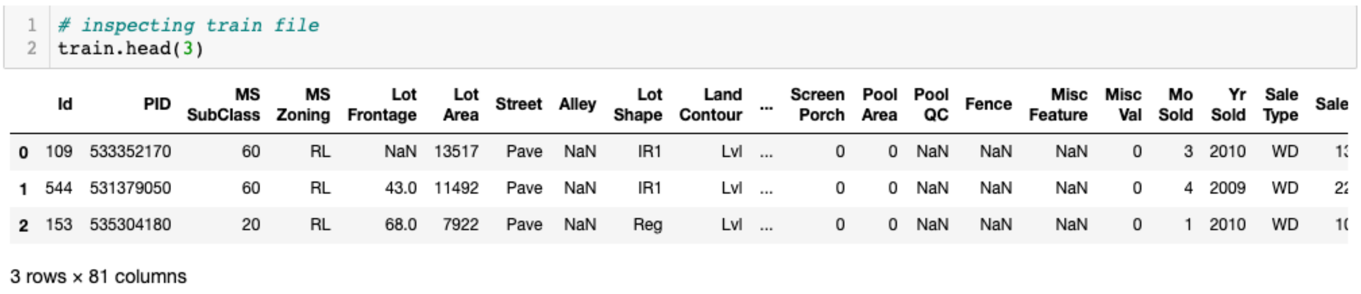
Exploratory Data Analysis (EDA)

An important step in any analysis is to define our target, or the variable we want to predict, in this case, the “sale price” column is the feature that I want to predict using the other features provided in the dataset.

EDA can reveal insights about our data that can be used to build a model as well as help the data scientist identify necessary data cleaning steps. As a first step, I imported the following libraries, which will be used to explore and visualize the data:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

Upon importing inspecting a few rows, I can see that there are 81 columns of data that include numerical and object type columns and from looking at the values for some observations, categorical data as well.



```
# inspecting train file
train.head(3)
```

	Id	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Screen Porch	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale
0	109	533352170	60	RL	NaN	13517	Pave	NaN	IR1	Lvl	...	0	0	NaN	NaN	NaN	0	3	2010	WD	1
1	544	531379050	60	RL	43.0	11492	Pave	NaN	IR1	Lvl	...	0	0	NaN	NaN	NaN	0	4	2009	WD	2
2	153	535304180	20	RL	68.0	7922	Pave	NaN	Reg	Lvl	...	0	0	NaN	NaN	NaN	0	1	2010	WD	1

3 rows × 81 columns

Modifying the column names:

I noted that column names are capitalized and contain spaces. I find it easier to analyze and perform subsequent modeling if feature names follow a certain format — lower cased and have spaces represented by an underscore.

Using the pandas rename method is a quick and simple way approach to do this, for example:

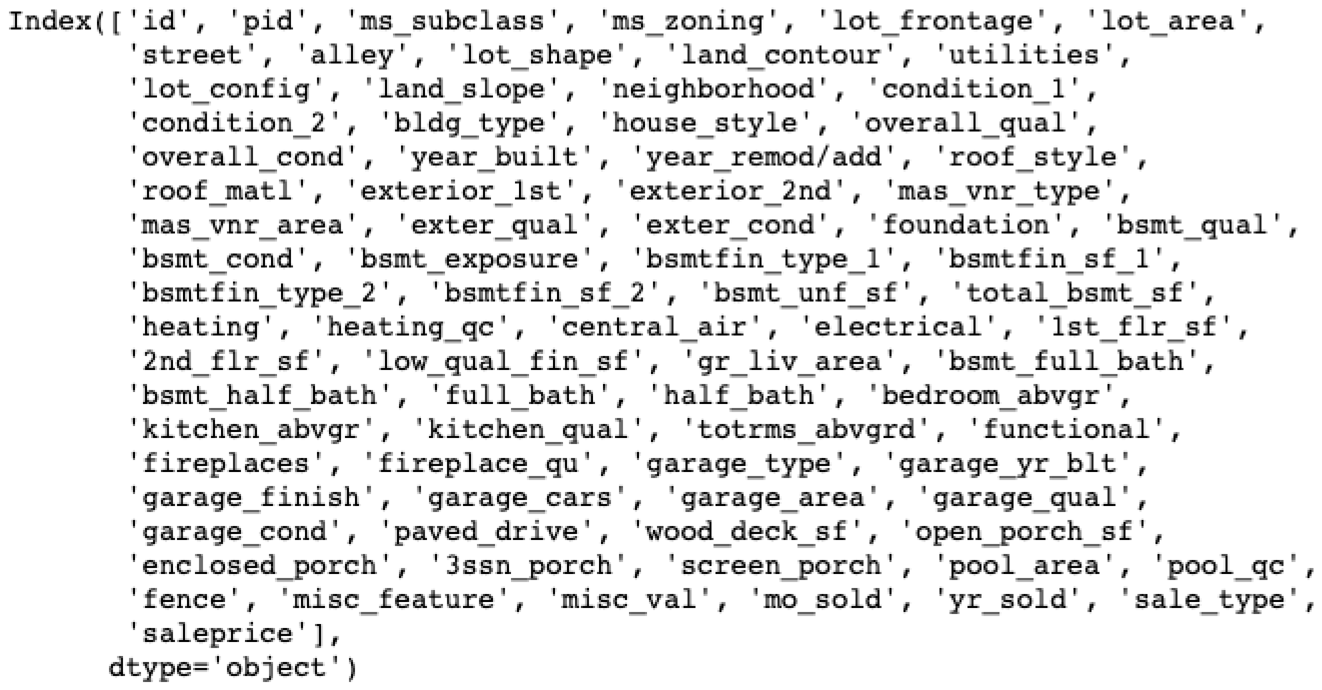
```
train.rename(columns={
    'Id': 'id',
    'PID': 'pid',
    'MS SubClass': 'ms_subclass',
}, inplace=True)
```

However, with so many columns in this data set, this method would be slow, inefficient and error prone. Fortunately, Python offers an efficient tool to fix this — the dictionary comprehension. For each column in my dataframe, the code below will replace spaces with an underscore and return lower case text.

```
train.columns = [i.replace(' ', '_').lower() for i in train.columns]
```

I can then inspect my columns and confirm that the change is as expected:

```
train.columns
```



```
Index(['id', 'pid', 'ms_subclass', 'ms_zoning', 'lot_frontage', 'lot_area',
       'street', 'alley', 'lot_shape', 'land_contour', 'utilities',
       'lot_config', 'land_slope', 'neighborhood', 'condition_1',
       'condition_2', 'bldg_type', 'house_style', 'overall_qual',
       'overall_cond', 'year_built', 'year_remod/add', 'roof_style',
       'roof_matl', 'exterior_1st', 'exterior_2nd', 'mas_vnr_type',
       'mas_vnr_area', 'exter_qual', 'exter_cond', 'foundation', 'bsmt_qual',
       'bsmt_cond', 'bsmt_exposure', 'bsmtfin_type_1', 'bsmtfin_sf_1',
       'bsmtfin_type_2', 'bsmtfin_sf_2', 'bsmt_unf_sf', 'total_bsmt_sf',
       'heating', 'heating_qc', 'central_air', 'electrical', '1st_flr_sf',
       '2nd_flr_sf', 'low_qual_fin_sf', 'gr_liv_area', 'bsmt_full_bath',
       'bsmt_half_bath', 'full_bath', 'half_bath', 'bedroom_abvgr',
       'kitchen_abvgr', 'kitchen_qual', 'totrms_abvgrd', 'functional',
       'fireplaces', 'fireplace_qu', 'garage_type', 'garage_yr_blt',
       'garage_finish', 'garage_cars', 'garage_area', 'garage_qual',
       'garage_cond', 'paved_drive', 'wood_deck_sf', 'open_porch_sf',
       'enclosed_porch', '3ssn_porch', 'screen_porch', 'pool_area', 'pool_qc',
       'fence', 'misc_feature', 'misc_val', 'mo_sold', 'yr_sold', 'sale_type',
       'saleprice'],
      dtype='object')
```

. . .

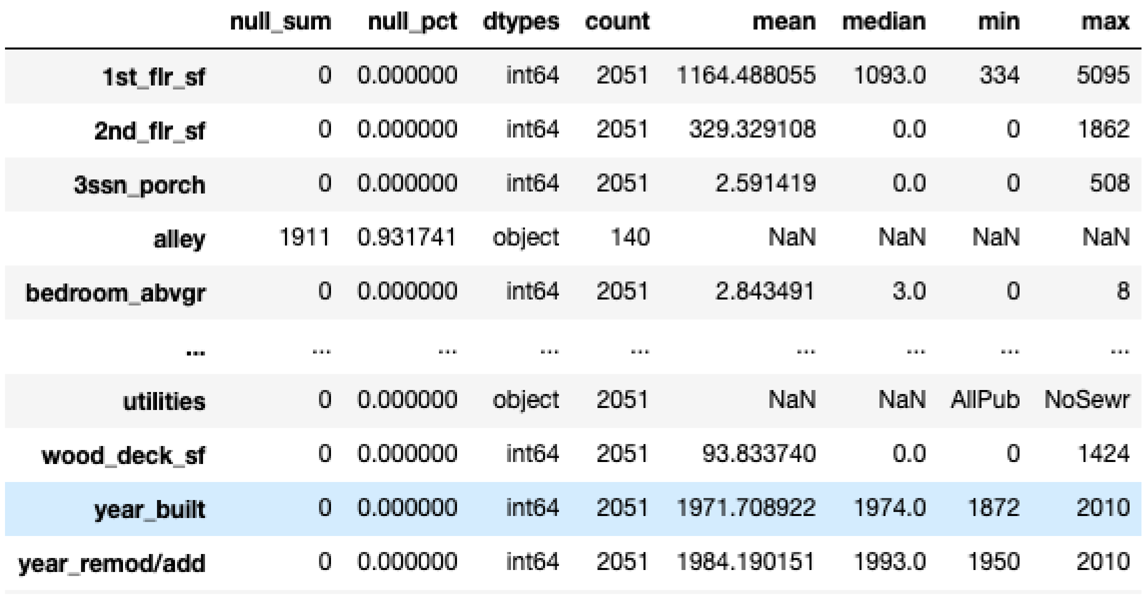
Efficient way to perform initial EDA

Next, I want to examine my data to inspect data types, identify null values, and inspect the summary statistics for each column available.

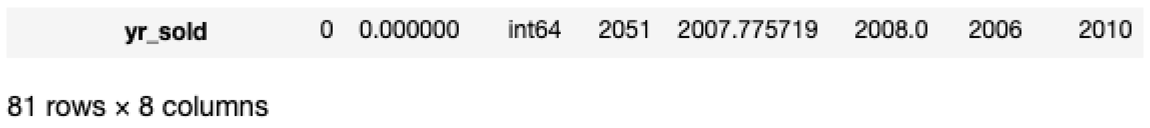
To achieve this, I will define a function, that takes my data as an input, and returns a data frame where each feature in my data set is now a row and the summary statistics are columns. The function will take a data frame as an input and calculate summary statistics to reveal insights about the data.

```
def ames_eda(df):
    eda_df = {}
    eda_df['null_sum'] = df.isnull().sum()
    eda_df['null_pct'] = df.isnull().mean()
    eda_df['dtypes'] = df.dtypes
    eda_df['count'] = df.count()
    eda_df['mean'] = df.mean()
    eda_df['median'] = df.median()
    eda_df['min'] = df.min()
    eda_df['max'] = df.max()

    return pd.DataFrame(eda_df)

ames_eda(train)
```

By passing my dataframe('train') into the function, I can get a quick and clean visual summary of my data, revealing summary statistics, data types and whether certain features have missing data and if so, what percentage of total data those values represent.



	null_sum	null_pct	dtypes	count	mean	median	min	max
1st_flr_sf	0	0.000000	int64	2051	1164.488055	1093.0	334	5095
2nd_flr_sf	0	0.000000	int64	2051	329.329108	0.0	0	1862
3ssn_porch	0	0.000000	int64	2051	2.591419	0.0	0	508
alley	1911	0.931741	object	140	NaN	NaN	NaN	NaN
bedroom_abvgr	0	0.000000	int64	2051	2.843491	3.0	0	8
...	...	...	...	...	...	...	...	...
utilities	0	0.000000	object	2051	NaN	NaN	AllPub	NoSewr
wood_deck_sf	0	0.000000	int64	2051	93.833740	0.0	0	1424
year_built	0	0.000000	int64	2051	1971.708922	1974.0	1872	2010
year_remod/add	0	0.000000	int64	2051	1984.190151	1993.0	1950	2010

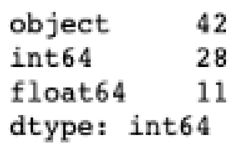


yr_sold	0	0.000000	int64	2051	2007.775719	2008.0	2006	2010

81 rows × 8 columns

From the dataframe above, I can see that there are object and integer columns. I want to inspect what all my column types and evaluate if there are any implications for EDA:

```
train.dtypes.value_counts()
```

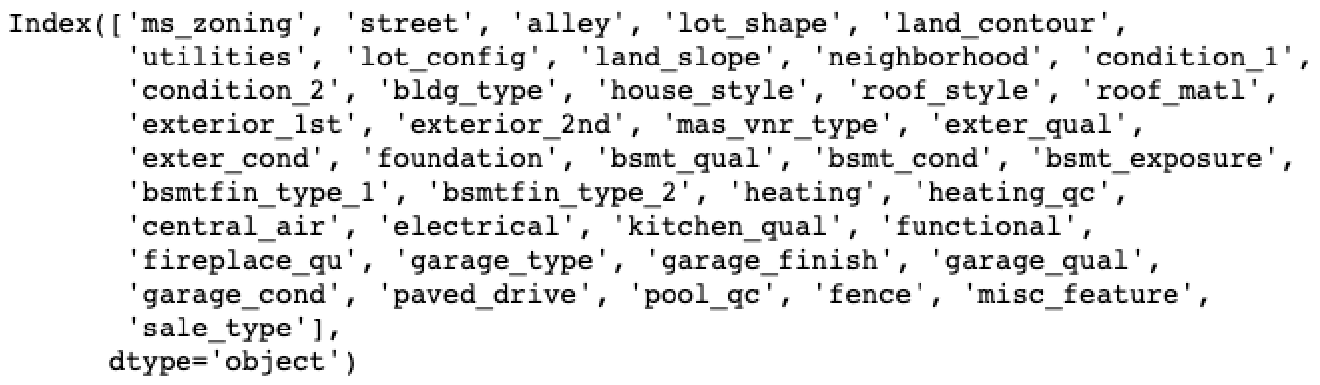


```
object     42
int64      28
float64    11
dtype: int64
```

This reveals that there are 42 object columns, 28 integer columns and 11 float columns in my dataset. Importantly, more than half my data consists of string or text values that will need to be explored and cleaned before modeling. As we will see below, just because the data does not have a numeric type out of the box does not mean it won't be important or useful to understand when modeling.

To see all the different object columns, I use the following code to return a list and inspect the data dictionary for a high level explanation of what these columns represent.

```
train.select_dtypes(include=['object']).columns
```



```
Index(['ms_zoning', 'street', 'alley', 'lot_shape', 'land_contour',
       'utilities', 'lot_config', 'land_slope', 'neighborhood', 'condition_1',
       'condition_2', 'bldg_type', 'house_style', 'roof_style', 'roof_matl',
       'exterior_1st', 'exterior_2nd', 'mas_vnr_type', 'exter_qual',
       'exter_cond', 'foundation', 'bsmt_qual', 'bsmt_cond', 'bsmt_exposure',
       'bsmtfin_type_1', 'bsmtfin_type_2', 'heating', 'heating_qc',
       'central_air', 'electrical', 'kitchen_qual', 'functional',
       'fireplace_qu', 'garage_type', 'garage_finish', 'garage_qual',
       'garage_cond', 'paved_drive', 'pool_qc', 'fence', 'misc_feature',
       'sale_type'],
      dtype='object')
```

Exploring the Object Columns:

By inspecting the data dictionary provided, I can see that many of these object columns, such as ‘central_air’ and ‘heating_qc’ are categorical or ordinal features that:

1. can be converted to numeric values through data cleaning, and
2. are intuitively related to the price of a house — a house with central air would logically have a higher sale price than one without, holding all else constant.

. . .

Exploring Relationships with our target:

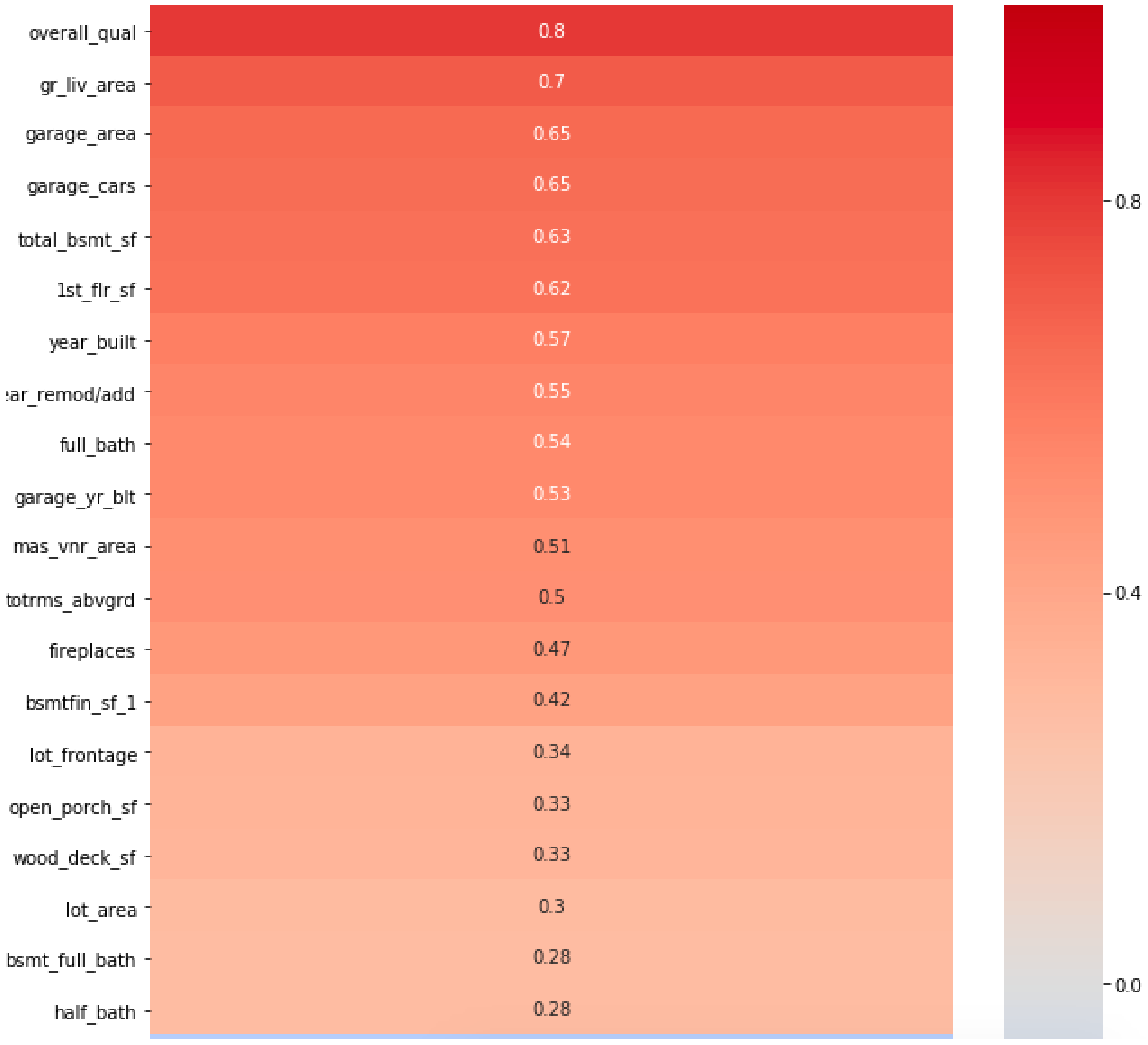
A key step in our EDA investment is to explore whether there is a relationship between our potential feature columns and our target, the home’s sale price.

Given the different data types of our features, we need to take a different approach to visually exploring the relationships.

For example, using our numeric columns, we can calculate the correlations between potential features and our target, but correlations won’t be calculated for non-numeric columns. This will be addressed further below.

Below, I’ve calculated the pairwise correlation between all of the numeric variables in the data frame and our target, sale price. Pandas’ corrwith() method will return a pair-wise correlation for each numeric variable with the target and ignore non-numeric columns.

```
correlations =
train.corrwith(train['saleprice']).iloc[:-1].to_frame()
correlations['abs'] = correlations[0].abs()
sorted_correlations = correlations.sort_values('abs',
ascending=False)[0]

fig, ax = plt.subplots(figsize=(10,20))
sns.heatmap(sorted_correlations.to_frame(), cmap='coolwarm',
annot=True, vmin=-1, vmax=1, ax=ax);
```



For ease of analysis, this visualization sorts the pair-wise correlations by absolute value.

Using seaborn, we can visualize these pair-wise correlations.

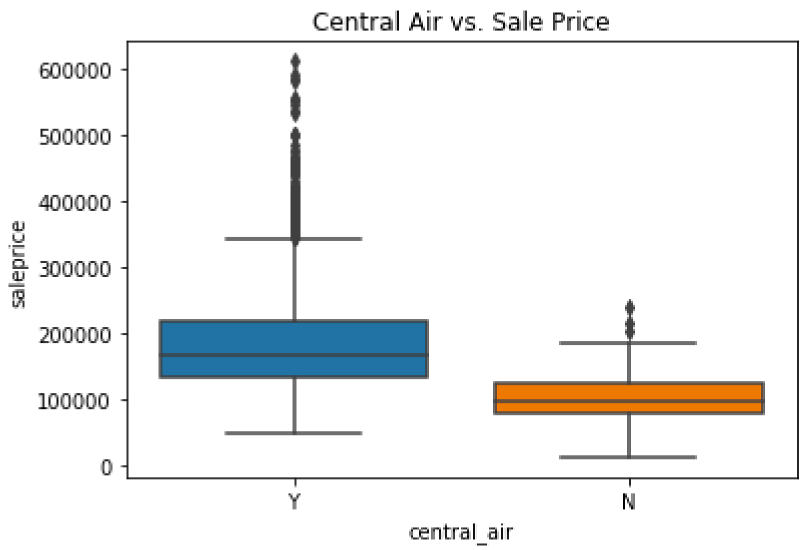
Unsurprisingly, features such as the overall quality of the home and the size of the living area have a strong relationship with the sale price.

But what about our non-numeric columns? Ordinal and categorical variables such as ‘exterior condition’ and ‘central air’ intuitively would have a relationship to sale price. It is critical that we visualize this!

A great way to achieve this is to generate a box plot that compares the values of an ordinal/categorical and visualize a relationship with sale price.

As we can see from the seaborn box plots below, there is a clear relationship between non-numeric values such the presence of central air or a home with “excellent” kitchen quality with sale price — relationships we want to capture in our model.

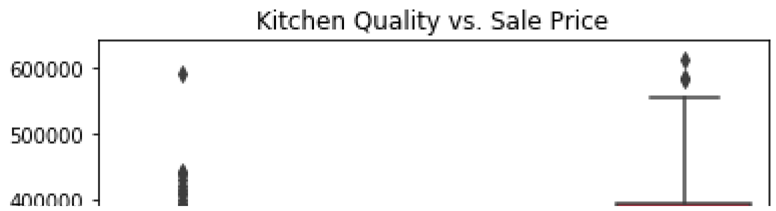
```
sns.boxplot(train['central_air'],
        train['saleprice']).set_title('Central Air vs. Sale Price');
```

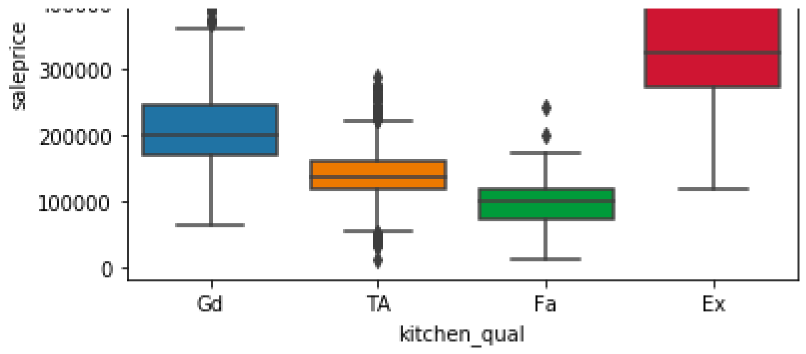


Based on this visualization, we will need to convert these columns to represent numeric values when we clean our data before modeling.

We can also use the box plots to look at features that have categorical values. For example, kitchen quality is ranked on a scale from poor to excellent and again we can visualize a relationship to price:

```
sns.boxplot(train['kitchen_qual'],
            train['saleprice']).set_title('Kitchen Quality vs. Sale
Price');
```

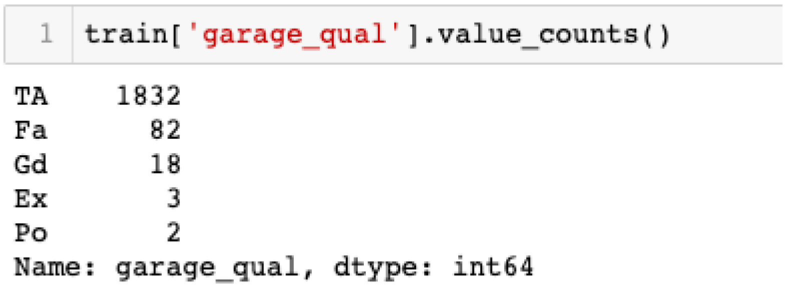




So we want to use the string columns, but how do we clean them?

One option to clean our categorical data is to define a function and apply it to our data such as in the example below to convert the garage quality from its categorical labels to numeric.

First, we want to identify the range of values that a certain feature may contain. Based on the values identified, we can create a function to overwrite each value with numerical values.



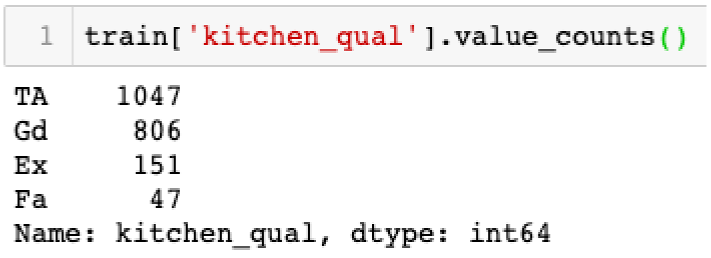
```
train['garage_qual'].value_counts()
```

```
TA    1832
Fa      82
Gd      18
Ex       3
Po       2
Name: garage_qual, dtype: int64
```

```
def garage_qual_cleaner(cell):
    if cell == 'Ex':
        return 5
    elif cell == 'Gd':
        return 4
    elif cell == 'TA':
        return 3
    elif cell == 'Fa':
        return 2
    elif cell == 'Po':
        return 1
    else:
        return 0
```

The function above would take a pandas series as an input and convert the string to a numeric value. We can then apply the function to the series of interest.

Alternatively, we could map a dictionary to overwrite values without creating a function.



```
train['kitchen_qual'].value_counts()
```

```
TA    1047
Gd     806
Ex     151
Fa      47
Name: kitchen_qual, dtype: int64
```

For example, given the values for kitchen quality we could use map a dictionary using the below code to convert the string values to a numeric data type:

```
train['kitchen_qual'].map({'Ex': 5, 'Gd': 4, 'TA': 3, 'Fa': 2, 'Po':
1})
```

However, given that there are 42 object columns that need cleaning, writing different sets of code to handle all these situations individually is inefficient.

Instead we can invest in defining one master function that can clean and organize the data:

```
def data_cleaner(df):
    # map numeric values onto all the quality columns using a quality
dictionary
    qual_dict = {'Po':0,'Fa':1,'TA':2,'Gd':3,'Ex':4}
    # create a list of ordinal column names
    ordinal_col_names = [col for col in df.columns if (col[-4:] in
['qual', 'cond']) and col[:3] != 'ove'] # last section ignores
"overall quality columns which will be addressed below
    # creating a new feature called age
    df['age'] = df.apply(lambda row: row['yr_sold'] -
max(row['year_built'], row['year_remod/add']), axis=1)
    # dummify the date sold column
    df['date_sold'] = df.apply(lambda row: str(row['mo_sold'])+ '-' +
str(row['yr_sold']), axis=1)
    df.loc[:,df.dtypes!= 'object'] = df.loc[:, df.dtypes !=
'object'].apply(lambda col: col.fillna(col.mean()))
```

```
    # transforming columns
    df[ordinal_col_names] = df[ordinal_col_names].applymap(lambda
cell: 2 if pd.isnull(cell) else qual_dict[cell])

    return df

# applying the function to train data
train = clean_data(train)
```

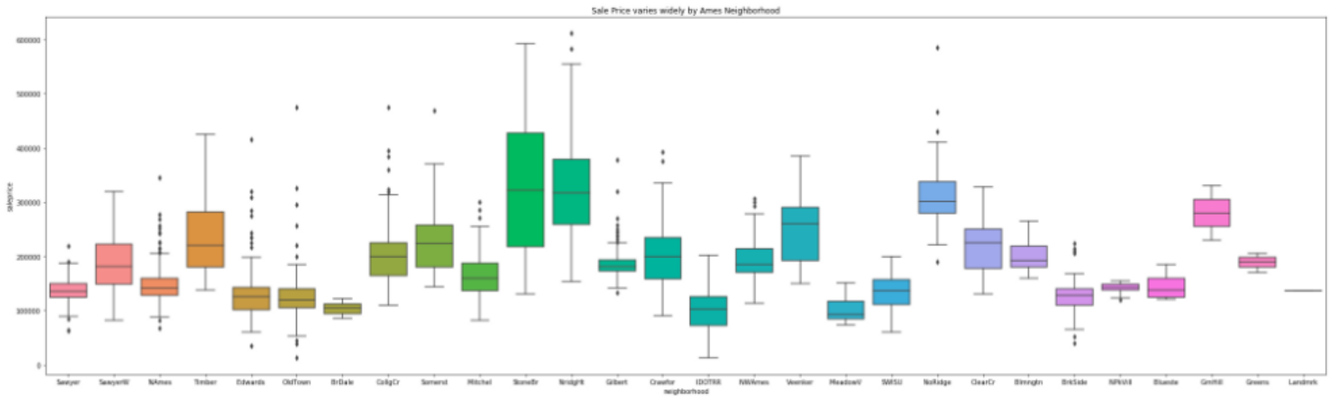
. . .

But what about variables without a clear, ordered relationship?

For example, the ‘neighborhood’ column contains string values detailing what neighborhood the home is located in. Without knowing the intimate details of the Ames real estate market, it is difficult to assign numeric values to the variable. We can’t simply assign random numbers or number them alphabetically as python will read neighborhood B (if we assign it as ‘2’) as being more valuable than neighborhood A (if we assign it as ‘1’), but less than neighborhood C — but this may not be the actual case and doing so would skew our model! So what do we do?

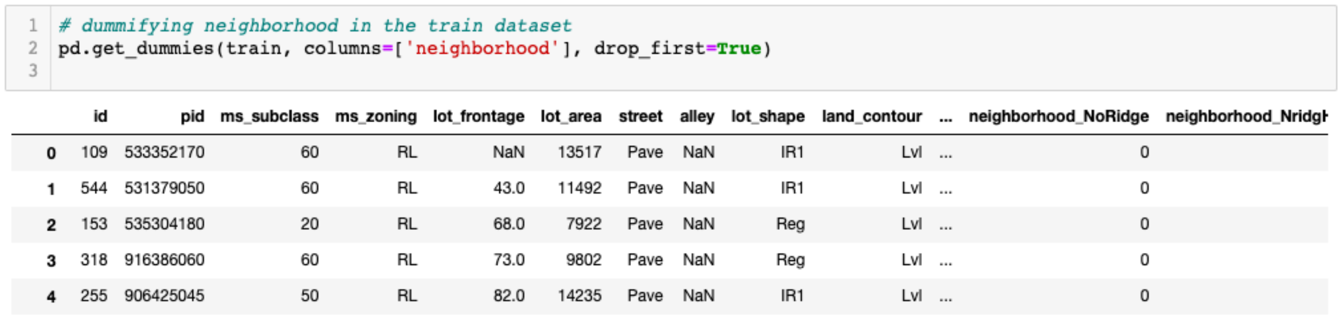
First, we want to visualize the relationship with our target:

```
plt.figure(figsize=(35,10)) # adjust the fig size to see everything
sns.boxplot(train['neighborhood'],
train['saleprice']).set_title('Sale Price varies widely by Ames
Neighborhood');
```



Some neighborhoods clearly have higher sale prices have than others — a relationship that we want to capture in our model.

Even string variables that do not have take on ordinal values such as neighborhood can be easily converted to a numerical amount by dummifying. Pandas provides a method to get dummify the variables — for each value (in this case neighborhood) a new feature will be created and the row will have a value of 0 or 1 for that column — a 1 signifying that in the original string column, a row contained the value that is now in the column name.



```
# dummifying neighborhood in the train dataset
pd.get_dummies(train, columns=['neighborhood'], drop_first=True)

```

	id	pid	ms_subclass	ms_zoning	lot_frontage	lot_area	street	alley	lot_shape	land_contour	...	neighborhood_NoRidge	neighborhood_Nridg
0	109	533352170	60	RL	NaN	13517	Pave	NaN	IR1	Lvl	...	0	
1	544	531379050	60	RL	43.0	11492	Pave	NaN	IR1	Lvl	...	0	
2	153	535304180	20	RL	68.0	7922	Pave	NaN	Reg	Lvl	...	0	
3	318	916386060	60	RL	73.0	9802	Pave	NaN	Reg	Lvl	...	0	
4	255	906425045	50	RL	82.0	14235	Pave	NaN	IR1	Lvl	...	0	

As shown above, will use the pandas get dummies method to convert these to numeric values. It is important that we call the 'drop_first' argument and set it as 'True.' This will dummify all variables after dropping the first one. We do this because it is important that a categorical variable of K categories, or levels, usually enters a regression as a sequence of K-1 dummy variables and the dropped variable will serve as our reference category. If a row has a value of 0 for all categories, we know that that observation belonged to the dropped column.

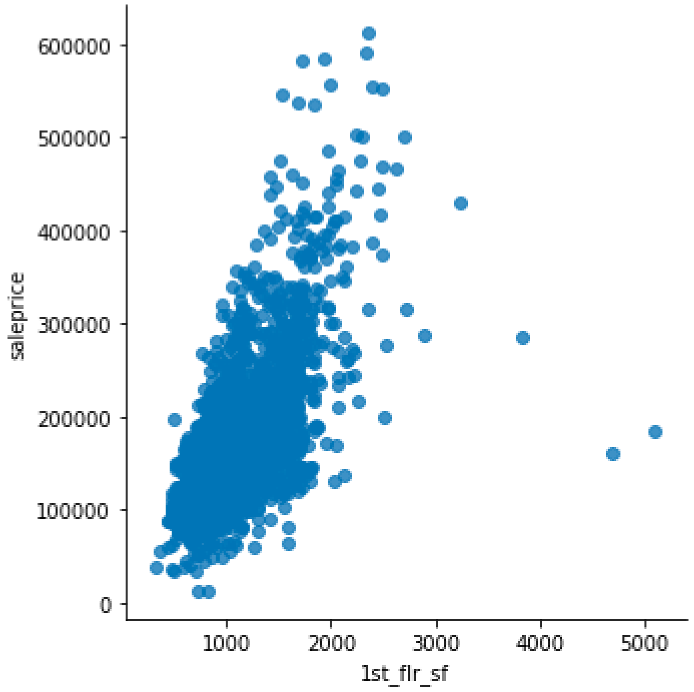
. . .

Now that we have cleaned and explored our data, we want to run a final check for outliers:

A quick way to check for outliers is to build a scatter plot between the target and a variable we would expect to be linearly related to our target. For example, it is not unreasonable to assume a linear relationship between sale price and square feet (as the size of the home increases, it logically follows that the sale price would as well).

To inspect this, I created a seaborn scatter plot between the target and the first floor square footage in a home:

```python
sns.lmplot(x='1st_flr_sf', y='saleprice', data=train, fit_reg=False);
```



We would expect to see a linear relationship between a variable such as first floor square footage and sale price (the larger the house, the higher the price). However, based on the scatterplot above, there are some outliers — large homes that have 4000 square feet on just one floor, but are relatively lower priced. We need to determine how best to handle these outliers to train our model.

Using pandas, I filtered the dataset below to inspect the outliers in greater detail:

```python
train.loc[train['1st_flr_sf'] > 3800]
```

	id	pid	ms_subclass	ms_zoning	lot_frontage	lot_area	street	alley	lot_shape	land_contour	...	pool_qc	fence	misc_feature	misc_val
616	1498	908154080	20	RL	123.0	47007	Pave	NaN	IR1	Lvl	...	NaN	NaN	NaN	0
960	1499	908154235	60	RL	313.0	63887	Pave	NaN	IR3	Bnk	...	Gd	NaN	NaN	0
1885	2181	908154195	20	RL	128.0	39290	Pave	NaN	IR1	Bnk	...	NaN	NaN	Elev	17000

3 rows × 83 columns

Based on the filtered data, rows 616, 960 and 1885 are our outlier columns — I will drop these from the dataset so that they do not skew the model. Using the for loop below, we can drop the outlier observations and then by re-examining our scatter plot, our data now appears less influenced by outliers.

```
rows_to_drop = [616, 960, 1885]

for row in rows_to_drop:
    train.drop(row, inplace=True)
```

. . .

Conclusion

As shown in this post, conducting exploratory data analysis and performing cleaning steps on the data can be an in depth process. But rather than viewing it as time consuming, it is important that we view this an important investment in the data science process. By investing our time to explore and clean our data, we have identified important outliers and extracted valuable information from string columns that will be critical to building a powerful predictive model.

In a subsequent post, I will detail how I used this data to build a linear regression model to predict the sale price of a home and use additional modeling techniques to improve upon the model.

Finally, you can check out my github for the full set of code I used to build this project.

Python Data Science Machine Learning Exploratory Data Analysis Data Cleaning