index

February 21, 2022

1 Linear Regression - Cumulative Lab

1.1 Introduction

In this cumulative lab you'll perform a full linear regression analysis and report the findings of your final model, including both predictive model performance metrics and interpretation of fitted model parameters.

1.2 Objectives

You will be able to:

- Perform a full linear regression analysis with iterative model development
- Evaluate your final model and interpret its predictive performance metrics
- Apply an inferential lens to interpret relationships between variables identified by the model

1.3 Your Task: Develop a LEGO Pricing Algorithm

Photo by Xavi Cabrera on Unsplash

1.3.1 Business Understanding

You just got hired by LEGO! Your first project is going to be to develop a pricing algorithm to help set a target price for new LEGO sets that are released to market. The goal is to save the company some time and to help ensure consistency in pricing between new products and past products.

The main purpose of this algorithm is *predictive*, meaning that **your model should be able to take in attributes of a LEGO set that does not yet have a set price, and to predict a good price**. The effectiveness of your predictive model will be measured by how well it predicts prices in our test set, where we know what the actual prices were but the model does not.

The secondary purpose of this algorithm is *inferential*, meaning that **your model should be able to tell us something about the relationship between the attributes of a LEGO set and its price**. You will apply your knowledge of statistics to include appropriate caveats about these relationships.

1.3.2 Data Understanding

You have access to a dataset containing over 700 LEGO sets released in the past, including attributes of those sets as well as their prices. You can assume that the numeric attributes in this dataset have already been preprocessed appropriately for modeling (i.e. that there are no missing or invalid values), while the text attributes are simply there for your visual inspection and should not be used for modeling. Also, note that some of these attributes cannot be used in your analysis because they will be unavailable for future LEGO products or are otherwise irrelevant.

You do not need to worry about inflation or differences in currency; just predict the same kinds of prices as are present in the past data, which have already been converted to USD.

1.3.3 Loading the Data

In the cells below, we load both the train and test datasets for you. Remember, both of these datasets contain prices, but we are using the test set as a stand-in for future LEGO products where the price has not yet been determined. The model will be trained on just the train set, then we will compare its predictions on the test set to the actual prices on the test set.

[1]:
```
# Run this cell without changes
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

[3]:
```
# Run this cell without changes

train = pd.read_csv("data/lego_train.csv")
test = pd.read_csv("data/lego_test.csv")

X_train = train.drop("list_price", axis=1)
y_train = train["list_price"]
```

```python
X_test = test.drop("list_price", axis=1)
y_test = test["list_price"]

# X_train
```

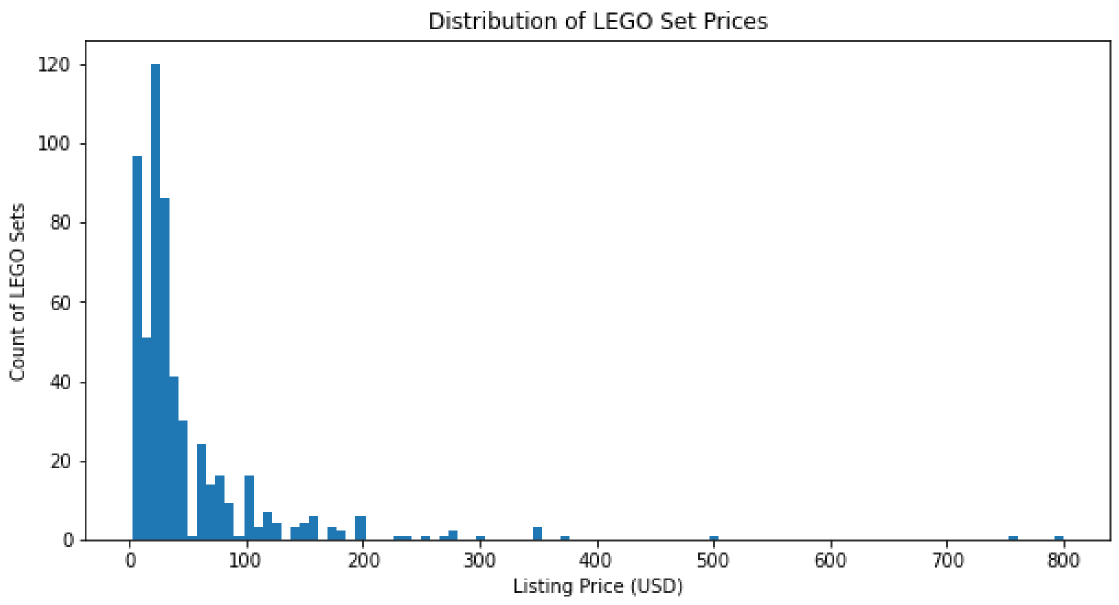
Some more information about the features of this dataset:

[4]:
```python
# Run this cell without changes
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 558 entries, 0 to 557
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   prod_id           558 non-null    int64  
 1   set_name          558 non-null    object 
 2   prod_desc         544 non-null    object 
 3   theme_name        558 non-null    object 
 4   piece_count       558 non-null    int64  
 5   min_age           558 non-null    float64
 6   max_age           558 non-null    float64
 7   difficulty_level  558 non-null    int64  
 8   num_reviews       490 non-null    float64
 9   star_rating       490 non-null    float64
dtypes: float64(4), int64(3), object(3)
memory usage: 43.7+ KB
```

A visualization of the distribution of the target variable:

[5]:
```python
# Run this cell without changes

fig, ax = plt.subplots(figsize=(10, 5))

ax.hist(y_train, bins=100)

ax.set_xlabel("Listing Price (USD)")
ax.set_ylabel("Count of LEGO Sets")
ax.set_title("Distribution of LEGO Set Prices");
```



1.3.4 Requirements

1. Interpret a Correlation Heatmap to Build a Baseline Model You'll start modeling by choosing the feature that is most correlated with our target, and build and evaluate a linear regression model with just that feature.

2. Build a Model with All Relevant Numeric Features Now, add in the rest of the relevant numeric features of the training data, and compare that model's performance to the performance of the baseline model.

3. Select the Best Combination of Features Using statistical properties of the fitted model, the `sklearn.feature_selection` submodule, and some custom code, find the combination of relevant numeric features that produces the best scores.

4. Build and Evaluate a Final Predictive Model Using the best features selected in the previous step, create a final model, fit it on all rows of the training dataset, and evaluate it on all rows of the test dataset in terms of both r-squared and RMSE.

5. Interpret the Final Model Determine what, if any, understanding of the underlying relationship between variables can be determined with this model. This means you will need to interpret the model coefficients as well as checking whether the assumptions of linear regression have been met.

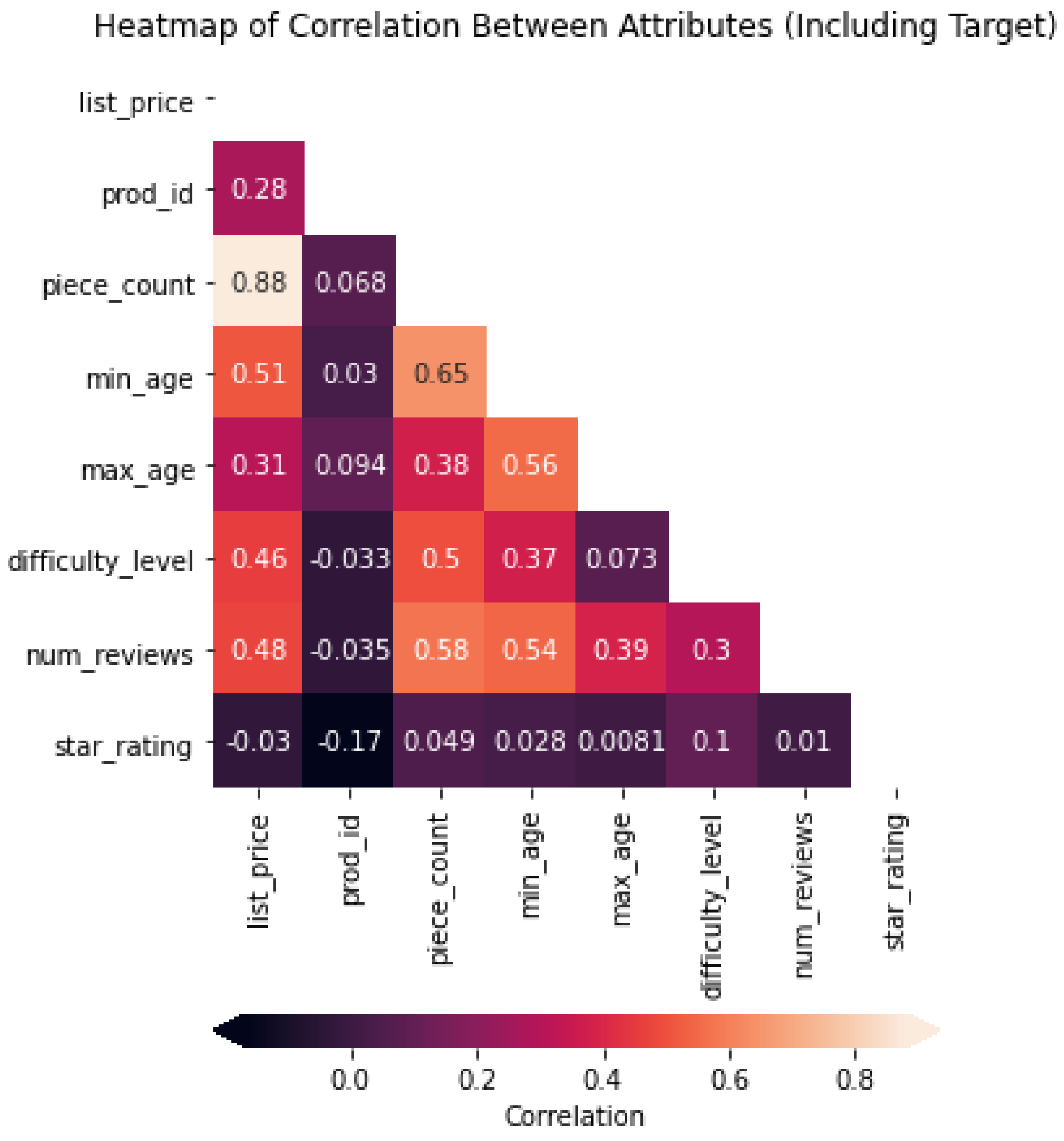
1.4 1. Interpret a Correlation Heatmap to Build a Baseline Model

1.4.1 Interpreting a Correlation Heatmap

The code below produces a heatmap showing the correlations between all of the numeric values in our training data. The x and y axis labels indicate the pair of values that are being compared, and then the color and the number are both representing the correlation. Color is used here to make it easier to find the largest/smallest numbers — you could perform this analysis with just `train.corr()` if all you wanted was the correlation values.

The very left column of the plot is the most important, since it shows correlations between the target (listing price) and other attributes.

[6]:
```
# Run this cell without changes

import seaborn as sns
import numpy as np

# Create a df with the target as the first column,
# then compute the correlation matrix
heatmap_data = pd.concat([y_train, X_train], axis=1)
corr = heatmap_data.corr()

# Set up figure and axes
fig, ax = plt.subplots(figsize=(5, 8))

# Plot a heatmap of the correlation matrix, with both
# numbers and colors indicating the correlations
sns.heatmap(
    # Specifies the data to be plotted
    data=corr,
    # The mask means we only show half the values,
    # instead of showing duplicates. It's optional.
    mask=np.triu(np.ones_like(corr, dtype=bool)),
    # Specifies that we should use the existing axes
    ax=ax,
    # Specifies that we want labels, not just colors
    annot=True,
    # Customizes colorbar appearance
    cbar_kws={"label": "Correlation", "orientation": "horizontal",
              "pad": .2, "extend": "both"}
)

# Customize the plot appearance
ax.set_title("Heatmap of Correlation Between Attributes (Including Target)");
```



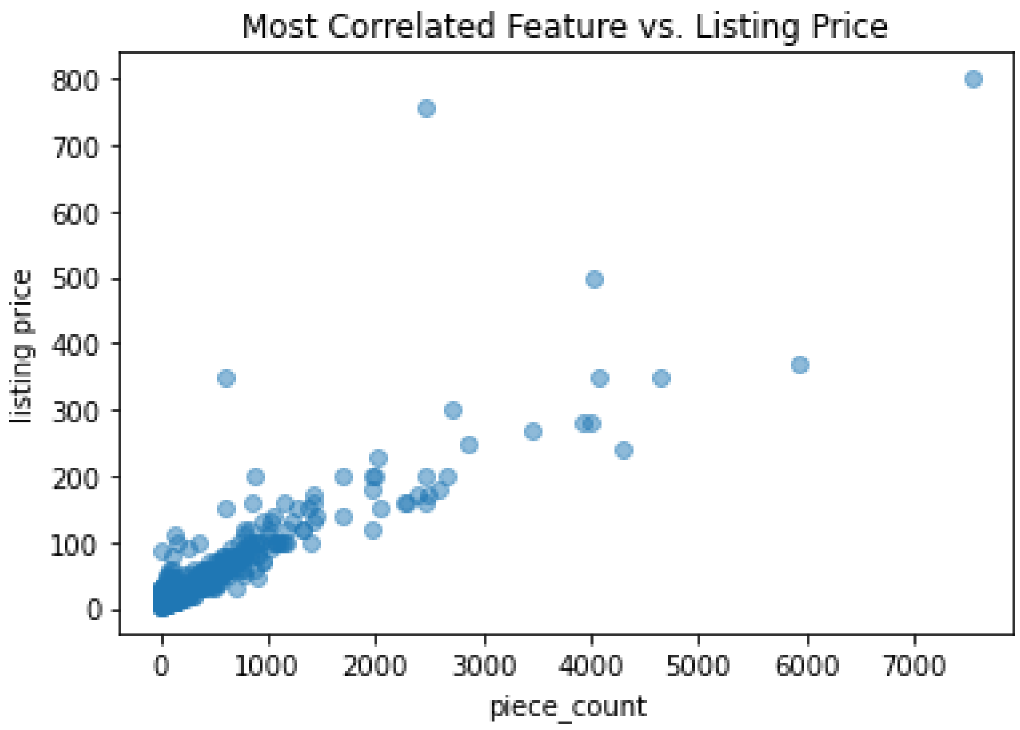
Based on the plot above, which feature is most strongly correlated with the target (`listing_price`)? In other words, which feature has the strongest positive or negative correlation — the correlation with the greatest magnitude?

[7]:
```
# Replace None with the name of the feature (a string)

most_correlated_feature = "piece_count"
```

(Make sure you run the cell above before proceeding.)

Let's create a scatter plot of that feature vs. listing price:

[9]:
```
# Run this cell without changes
fig, ax = plt.subplots()

ax.scatter(X_train[most_correlated_feature], y_train, alpha=0.5)
ax.set_xlabel(most_correlated_feature)
ax.set_ylabel("listing price")
ax.set_title("Most Correlated Feature vs. Listing Price");
```



Assuming you correctly identified `piece_count` (the number of pieces in the LEGO set) as the most correlated feature, you should have a scatter plot that shows a fairly clear linear relationship between that feature and the target. It looks like we are ready to proceed with making our baseline model without any additional transformation.

1.4.2 Building a Baseline Model

Now, we'll build a linear regression model using just that feature, which will serve as our baseline model:

[10]:
```
# Run this cell without changes

from sklearn.linear_model import LinearRegression

baseline_model = LinearRegression()
```

Then we evaluate the model using `cross_validate` and `ShuffleSplit`, which essentially means that we perform 3 separate train-test splits within our `X_train` and `y_train`, then we find both the train and the test scores for each.

[11]:
```python
# Run this cell without changes

from sklearn.model_selection import cross_validate, ShuffleSplit

splitter = ShuffleSplit(n_splits=3, test_size=0.25, random_state=0)

baseline_scores = cross_validate(
    estimator=baseline_model,
    X=X_train[[most_correlated_feature]],
    y=y_train,
    return_train_score=True,
    cv=splitter
)

print("Train score:     ", baseline_scores["train_score"].mean())
print("Validation score:", baseline_scores["test_score"].mean())
```

```
Train score:      0.7785726407224942
Validation score: 0.7793473618106956
```

Interpret these scores below. What are we measuring? What can we learn from this?

Hint: when you use `cross_validate`, it uses the `.score` method of the estimator by default. See documentation here for that method of `LinearRegression`.

[49]:
```python
# Replace None with appropriate text
"""
We are using .score method which gives back the coefficient of determination
of the prediction which means we find R2.
"""

### From GitHub Solution

"""
Because we are using the .score method of LinearRegression, these
are r-squared scores. That means that each of them represents the
amount of variance of the target (listing price) that is explained
by the model's features (currently just the number of pieces) and
parameters (intercept value and coefficient values for the features)

In general this seems like a fairly strong model already. It is
getting nearly identical performance on training subsets compared to
the validation subsets, explaining around 80% of the variance both
times
"""
```

[49]:
```
"\nBecause we are using the .score method of LinearRegression, these\nare
r-squared scores. That means that each of them represents the\namount of
variance of the target (listing price) that is explained\nby the model's
features (currently just the number of pieces) and\nparameters (intercept value
and coefficient values for the features)\n\nIn general this seems like a fairly
strong model already. It is\ngetting nearly identical performance on training
subsets compared to\nthe validation subsets, explaining around 80% of the
variance both\ntimes\n"
```

1.5 2. Build a Model with All Numeric Features

Now that we have established a baseline, it's time to move on to more complex models.

1.5.1 Numeric Feature Selection

One thing that you will almost always need to do in a modeling process is remove non-numeric data prior to modeling. While you could apply more-advanced techniques such as one-hot encoding or NLP in order to convert non-numeric columns into numbers, this time just create a dataframe `X_train_numeric` that is a copy of `X_train` that only contains numeric columns.

You can look at the `df.info()` printout above to do this manually, or there is a handy `.select_dtypes` method (documentation here).

[12]:
```python
# Replace None with appropriate code

X_train_numeric = X_train.select_dtypes(include=["float64", "int64"]).copy()

X_train_numeric

### From GitHub

# X_train_numeric = X_train.select_dtypes("number").copy()

# X_train_numeric
```

[12]:
```
     prod_id  piece_count  min_age  max_age  difficulty_level  num_reviews  \
0      60123          330      7.0     12.0                 1          3.0
1      71246           96      7.0     14.0                 1          3.0
2      10616           25      1.5      5.0                 1          3.0
3      31079          379      8.0     12.0                 1          5.0
4      42057          199      8.0     14.0                 1          9.0
..       …            …        …        …                 …            …
553    71343           56      7.0     14.0                 1          1.0
554    75114           81      7.0     14.0                 0         10.0
555    41597          708     10.0     99.0                 2         13.0
556    75116           98      8.0     14.0                 1          1.0
557    76097          406      7.0     14.0                 2          5.0
```

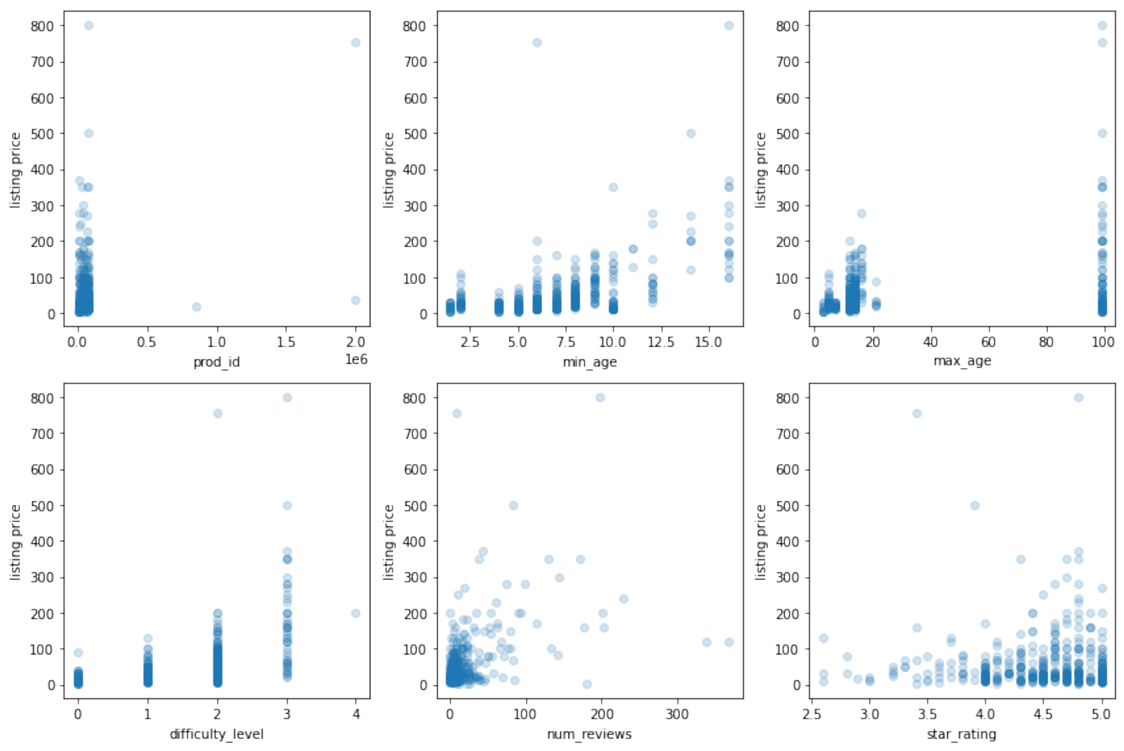
```
     star_rating
0            4.3
1            4.7
2            5.0
3            4.4
4            4.7
..           …
553          5.0
554          4.7
555          4.8
556          5.0
557          4.8

[558 rows x 7 columns]
```

The following code checks that your answer was correct:

[13]:
```
# Run this cell without changes

# X_train_numeric should be a dataframe
assert type(X_train_numeric) == pd.DataFrame

# Check for correct shape
assert X_train_numeric.shape == (558, 7)
```

Now we can look at scatter plots of all numeric features compared to the target (skipping `piece_count` since we already looked at that earlier):

[14]:
```
# Run this cell without changes

scatterplot_data = X_train_numeric.drop("piece_count", axis=1)

fig, axes = plt.subplots(ncols=3, nrows=2, figsize=(12, 8))
fig.set_tight_layout(True)

for index, col in enumerate(scatterplot_data.columns):
    ax = axes[index//3][index%3]
    ax.scatter(X_train_numeric[col], y_train, alpha=0.2)
    ax.set_xlabel(col)
    ax.set_ylabel("listing price")
```



1.5.2 Feature Selection Using Domain Understanding

Ok, now all of the remaining features can technically go into a model with scikit-learn. But do they make sense?

Some reasons you might not want to include a given numeric column include:

1. The column represents a unique identifier, not an actual numeric feature
2. The column is something that will not be available when making future predictions

Recall that the business purpose here is creating an algorithm to set the price for a newly-released LEGO set. Which columns should we drop because of the issues above?

[53]:
```
# Replace None with appropriate text
"""
First issue is prod_id, which is the name of the product and it is different
from the numerical values in the mathematical sense because these numbers just
represent the name of the product possibly coming from merging different
data frames.

The second and third issues are num_reviews and star_rating because we have not␣
 ↪released
the product yet.
"""
```

```
#### From GitHub

"""
The first issue aligns with the first feature we have, prod_id

While it is possible that there is some useful information
encoded in that number, it seems like it is not really a numeric
feature in the traditional sence

The scatter plot supports this idea, since it shows almost all
prices being represented by a narrow range of ID values

The second issue aligns with num_reviews and star_rating. Although
these might be useful features in some modeling context, they are
not useful for this algorithm because we won't know the number of
reviews or the star rating until after the LEGO set is released.
"""
```

[53]:
```
"\nThe first issue aligns with the first feature we have, prod_id\n\nWhile it is
possible that there is some useful information \nencoded in that number, it
seems like it is not really a numeric\nfeature in the traditional sence\n\nThe
scatter plot supports this idea, since it shows almost all\nprices being
represented by a narrow range of ID values\n\nThe second issue aligns with
num_reviews and star_rating. Although\nthese might be useful features in some
modeling context, they are\nnot useful for this algorithm because we won't know
the number of\nreviews or the star rating until after the LEGO set is
released.\n"
```

Now, create a variable `X_train_second_model`, which is a copy of `X_train_numeric` where those irrelevant columns have been removed:

[54]:
```
# Replace None with appropriate code
to_drop = ["prod_id", "num_reviews", "star_rating"]

X_train_second_model = X_train_numeric.drop(columns = to_drop, axis = 1).copy()

X_train_second_model
```

[54]:
```
     piece_count  min_age  max_age  difficulty_level
0            330      7.0     12.0                 1
1             96      7.0     14.0                 1
2             25      1.5      5.0                 1
3            379      8.0     12.0                 1
4            199      8.0     14.0                 1
..           …        …        …                 …
553           56      7.0     14.0                 1
554           81      7.0     14.0                 0
```

```
555            708      10.0      99.0                     2
556             98       8.0      14.0                     1
557            406       7.0      14.0                     2

[558 rows x 4 columns]
```

1.5.3 Building and Evaluating the Second Model

In the cell below, we use the same process to evaluate a model using `X_train_second_model` rather than using just `piece_count`.

[55]:
```
# Run this cell without changes

second_model = LinearRegression()

second_model_scores = cross_validate(
    estimator=second_model,
    X=X_train_second_model,
    y=y_train,
    return_train_score=True,
    cv=splitter
)

print("Current Model")
print("Train score:     ", second_model_scores["train_score"].mean())
print("Validation score:", second_model_scores["test_score"].mean())
print()
print("Baseline Model")
print("Train score:     ", baseline_scores["train_score"].mean())
print("Validation score:", baseline_scores["test_score"].mean())
```

```
Current Model
Train score:      0.7884552982196166
Validation score: 0.755820363666055

Baseline Model
Train score:      0.7785726407224942
Validation score: 0.7793473618106956
```

Interpret these results. Did our second model perform better than the baseline? Any ideas about why or why not?

Hint: because the purpose of this model is to set future prices that have not been determined yet, the most important metric for evaluating model performance is the validation score, not the train score.

[56]:
```
# Replace None with appropriate text
"""
It seems that the selected features increased the R2 score for the training set
```

```
while it reduced the R2 score for the test set. This means that our
model's ability is reduced becasue the R2 score for the test set is important
since they are not part of the training set and we use them to check the model's
ability of predicting future data. Therefore, we can conclude that adding these
features incread the overfitting of the model. So, we need to add some other
features to the model and changed the existing features.
"""



#### From GitHub

"""
Our second model got slightly better scores on the training
data, but worse scores on the validation data. This means that
it is a worse model overall, since what we care about is the
ability to generate prices for future LEGO sets, not the
ability to fit well to the known LEGO sets' features

It seems like adding in these other features is actually just
causing overfitting, rather than improving the model's ability
to understand the underlying patterns in the data
"""
```

[56]:
```
"\nOur second model got slightly better scores on the training\ndata, but worse
scores on the validation data. This means that\nit is a worse model overall,
since what we care about is the\nability to generate prices for future LEGO
sets, not the\nability to fit well to the known LEGO sets' features\n\nIt seems
like adding in these other features is actually just\ncausing overfitting,
rather than improving the model's ability\nto understand the underlying patterns
in the data\n"
```

[57]:
```
### For Myself,
### I would like to know the correlation between maximum/minimum ages with the
### level of dificulty.

to_check = X_train_second_model.copy()
data_corr = to_check.corr().abs().stack().reset_index().sort_values(0,␣
 ↪ascending=False)
data_corr['pairs'] = list(zip(data_corr.level_0, data_corr.level_1))
corr_1_ind = data_corr.iloc[0:len(X_train_second_model.columns)].index
data_corr.drop(index = corr_1_ind, inplace = True)
data_corr.drop(columns = ["level_1", "level_0"], inplace = True)
```

[58]:
```
data_corr.reset_index(inplace = True, drop = True)
data_corr
```

```
[58]:           0                             pairs
      0   0.647647            (piece_count, min_age)
      1   0.647647            (min_age, piece_count)
      2   0.557013                (min_age, max_age)
      3   0.557013                (max_age, min_age)
      4   0.497373  (piece_count, difficulty_level)
      5   0.497373  (difficulty_level, piece_count)
      6   0.377158            (piece_count, max_age)
      7   0.377158            (max_age, piece_count)
      8   0.374841      (min_age, difficulty_level)
      9   0.374841      (difficulty_level, min_age)
      10  0.073348      (max_age, difficulty_level)
      11  0.073348      (difficulty_level, max_age)
```

```
[59]: data_corr.iloc[list(range(0, len(data_corr), 2))]
```

```
[59]:           0                             pairs
      0   0.647647            (piece_count, min_age)
      2   0.557013                (min_age, max_age)
      4   0.497373  (piece_count, difficulty_level)
      6   0.377158            (piece_count, max_age)
      8   0.374841      (min_age, difficulty_level)
      10  0.073348      (max_age, difficulty_level)
```

1.6 3. Select the Best Combination of Features

As you likely noted above, adding all relevant numeric features did not actually improve the model performance. Instead, it led to overfitting.

1.6.1 Investigating Multicollinearity

This potentially indicates that our model is performing poorly because these features violate the independence assumption (i.e. there is strong multicollinearity). In other words, maybe the minimum age, maximum age, and difficulty level are not really providing different information than the number of pieces in the LEGO set, and instead are just adding noise. Then the model is using that noise to get a slightly better score on the training data, but but a worse score on the validation data.

While `LinearRegression` from scikit-learn has a lot of nice functionality for working with a predictive framing (e.g. compatibility with the `cross_validate` function), it doesn't have anything built in to detect strong multicollinearity. Fortunately the same linear regression model is also available from StatsModels (documentation here), where it is called `OLS` (for "ordinary least squares"). Models in StatsModels, including `OLS`, are not really designed for predictive model validation, but they do give us a lot more statistical information.

In the cell below, we use StatsModels to fit and evaluate a linear regression model on the same features used in the second model. Note that we will only see one r-squared value (not train and validation r-squared values) because it is using the full `X_train` dataset instead of using cross-validation.

```
[60]: # Run this cell without changes

      import statsmodels.api as sm

      sm.OLS(y_train, sm.add_constant(X_train_second_model)).fit().summary()
```

```
[60]: <class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:             list_price   R-squared:                       0.786
Model:                            OLS   Adj. R-squared:                  0.784
Method:                 Least Squares   F-statistic:                     506.8
Date:                Fri, 04 Feb 2022   Prob (F-statistic):         2.40e-183
Time:                        16:14:14   Log-Likelihood:                -2741.4
No. Observations:                 558   AIC:                             5493.
Df Residuals:                     553   BIC:                             5514.
Df Model:                           4
Covariance Type:            nonrobust
================================================================================
====
                       coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
----
const               22.0991      4.550      4.857      0.000      13.162
31.037
piece_count          0.0906      0.003     33.407      0.000       0.085
0.096
min_age             -2.7627      0.696     -3.971      0.000      -4.129
-1.396
max_age              0.0501      0.051      0.981      0.327      -0.050
0.150
difficulty_level     3.3626      2.064      1.629      0.104      -0.691
7.416
==============================================================================
Omnibus:                      838.141   Durbin-Watson:                   1.976
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           331324.405
Skew:                           8.064   Prob(JB):                         0.00
Kurtosis:                     121.281   Cond. No.                     2.85e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 2.85e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

```
"""
```

A condition number of 10-30 indicates multicollinearity, and a condition number above 30 indicates strong multicollinearity. This print-out shows a condition number of `2.77e+03`, i.e. 2770, which is well above 30.

In a predictive context (we are currently trying to build a model to assign prices to future LEGO sets, not a model primarily intended for understanding the relationship between prices and attributes of past LEGO sets), we do not *always* need to be worried when we identify strong multicollinearity. Sometimes there are features that are highly collinear but they also are individually communicating useful information to the model. In this case, however, it seems like strong multicollinearity might be what is causing our second model to have worse performance than the first model.

1.6.2 Selecting Features Based on p-values

Given that we suspect our model's issues are related to multicollinearity, let's try to narrow down those features. In this case, let's use the p-values assigned to the coefficients of the model.

Looking at the model summary above, ***which features are statistically significant, with p-values above 0.05***? (P-values are labeled **P>|t|** in a StatsModels summary.)

[ ]:
```
# Replace None with appropriate text
"""
"min_age" and "piece_count" and "const"
"""
```

Important note: There are many limitations to using coefficient p-values to select features. See this StackExchange answer with examples in R for more details. The suggested alternative in that answer, `glmnet`, is a form of *regularization*, which you will learn about later. Another related technique is *dimensionality reduction*, which will also be covered later. However for now you can proceed using just the p-values technique until the more-advanced techniques have been covered.

In the cell below, create a list `significant_features` that contains the names of the columns whose features have statistically significant coefficient p-values. You should not include `"const"` in that list because `LinearRegression` from scikit-learn automatically adds a constant term and there is no column of `X_train` called `"const"`.

(You do not need to extract this information programmatically, just write them out like `"column_name_1", "column_name_2"` etc.)

[61]:
```
# Replace None with appropriate code
significant_features = ["min_age", "piece_count"]
```

Now let's build a model using those significant features only:

[62]:
```
# Run this cell without changes
third_model = LinearRegression()
X_train_third_model = X_train[significant_features]
```

```python
third_model_scores = cross_validate(
    estimator=third_model,
    X=X_train_third_model,
    y=y_train,
    return_train_score=True,
    cv=splitter
)

print("Current Model")
print("Train score:     ", third_model_scores["train_score"].mean())
print("Validation score:", third_model_scores["test_score"].mean())
print()
print("Second Model")
print("Train score:     ", second_model_scores["train_score"].mean())
print("Validation score:", second_model_scores["test_score"].mean())
print()
print("Baseline Model")
print("Train score:     ", baseline_scores["train_score"].mean())
print("Validation score:", baseline_scores["test_score"].mean())
```

```
Current Model
Train score:      0.7869252233899847
Validation score: 0.7638761794341221

Second Model
Train score:      0.7884552982196166
Validation score: 0.755820363666055

Baseline Model
Train score:      0.7785726407224942
Validation score: 0.7793473618106956
```

Interpret the results below. What happened when we removed the features with high p-values?

```
[ ]: # Replace None with appropriate text
     """
     R2 score for the test set is improved compared to the two previous models
     and R2 score of train set of our current model is almost the same as the
     R2 score of the second model and both of these models are better than the
     R2 score of the Baseline Model. So removing "max_age" and "level_difficulty"
     has improved our model.
     """
```

1.6.3 Selecting Features with `sklearn.feature_selection`

Let's try a different approach. Scikit-learn has a submodule called `feature_selection` that includes tools to help reduce the feature set.

We'll use `RFECV` (documentation here). "RFE" stands for "recursive feature elimination", meaning that it repeatedly scores the model, finds and removes the feature with the lowest "importance", then scores the model again. If the new score is better than the previous score, it continues removing features until the minimum is reached. "CV" stands for "cross validation" here, and we can use the same splitter we have been using to test our data so far.

[63]:
```
# Run this cell without changes

from sklearn.feature_selection import RFECV
from sklearn.preprocessing import StandardScaler

# Importances are based on coefficient magnitude, so
# we need to scale the data to normalize the coefficients
X_train_for_RFECV = StandardScaler().fit_transform(X_train_second_model)

model_for_RFECV = LinearRegression()

# Instantiate and fit the selector
selector = RFECV(model_for_RFECV, cv=splitter)
selector.fit(X_train_for_RFECV, y_train)

# Print the results
print("Was the column selected?")
for index, col in enumerate(X_train_second_model.columns):
    print(f"{col}: {selector.support_[index]}")
```

```
Was the column selected?
piece_count: True
min_age: False
max_age: False
difficulty_level: False
```

Interesting. So, this algorithm is saying that our baseline model, with `piece_count` as the only feature, is the best one it could find.

However, note that this is based on the "importances" of the features, which means the coefficients in the context of a linear regression. It is possible that we can still get a better model by including multiple features, if we try removing columns using a different strategy.

1.6.4 A Brute Force Approach

Given that we have only four columns and only a few hundred rows, one other option we have is something more computationally expensive: write custom code that goes over multiple different permutations of the columns, to see if we can find something better than the p-values approach or the `RFECV` approach.

The code below assumes that we want to keep the `piece_count` column, then attempts a linear regression with all possible combinations of 1-2 additional features. Don't worry too much if you don't understand everything that is happening here — an approach like this should be a last resort and you may not ever need to use it!

[64]:
```
# Run this cell without changes

from itertools import combinations

features = ["piece_count", "min_age", "max_age", "difficulty_level"]

# Make a dataframe to hold the results (not strictly necessary
# but it makes the output easier to read)
results_df = pd.DataFrame(columns=features)

# Selecting just piece_count
results_df = results_df.append({
    "train_score": baseline_scores["train_score"].mean(),
    "val_score": baseline_scores["test_score"].mean()
}, ignore_index=True)

# Selecting 1 additional feature
for feature in features[1:]:
    scores = cross_validate(
        estimator=second_model,
        X=X_train[["piece_count", feature]],
        y=y_train,
        return_train_score=True,
        cv=splitter
    )
    # Note: this technique of appending to a df is quite inefficient
    # Here it works because it's only happening 6 times, but avoid
    # doing this for a whole dataset
    results_df = results_df.append({
        feature: "Yes",
        "train_score": scores["train_score"].mean(),
        "val_score": scores["test_score"].mean()
    }, ignore_index=True)

# Selecting 2 additional features
for (feature1, feature2) in list(combinations(features[1:], 2)):
    scores = cross_validate(
        estimator=second_model,
        X=X_train[["piece_count", feature1, feature2]],
        y=y_train,
        return_train_score=True,
        cv=splitter
    )
    results_df = results_df.append({
        feature1: "Yes",
        feature2: "Yes",
        "train_score": scores["train_score"].mean(),
        "val_score": scores["test_score"].mean()
    }, ignore_index=True)

# Including all 3 additional features
results_df = results_df.append({
    "min_age": "Yes", "max_age": "Yes", "difficulty_level": "Yes",
    "train_score": second_model_scores["train_score"].mean(),
    "val_score": second_model_scores["test_score"].mean()
}, ignore_index=True)

# Fill in remaining values where appropriate
results_df["piece_count"] = "Yes"
results_df.fillna("No", inplace=True)

results_df
```

[64]:

```
  piece_count min_age max_age difficulty_level  train_score  val_score
0         Yes      No      No               No     0.778573   0.779347
1         Yes     Yes      No               No     0.786925   0.763876
2         Yes      No     Yes               No     0.778837   0.780949
3         Yes      No      No              Yes     0.778669   0.780662
4         Yes     Yes     Yes               No     0.788011   0.751768
5         Yes     Yes      No              Yes     0.787145   0.767399
6         Yes      No     Yes              Yes     0.778920   0.781578
7         Yes     Yes     Yes              Yes     0.788455   0.755820
```

Interpret the table above. It shows both training and validation scores for `piece_count` as well as all combinations of 0, 1, 2, or 3 other features.

Which features make the best model? Which make the worst? How does this align with the previous discussion of multicollinearity? And how much does feature selection seem to matter in general for this dataset + model algorithm, once we have identified the most correlated feature for the baseline?

[ ]:

```
# Replace None with appropriate text
"""
the best model has
"piece_count", "difficulty_level", and "max_age" because the R2 score of the
validation is 0.78.

It seems that there are some collinarity between the features leading our model
to overfitting the train sets and in turn increasing the R2 score of train set
and decreasing the R2 score of test set.
"""

#### From gitHub

"""
```

```
The best model uses piece_count, max_age, and difficulty_level
It has a validation score of 0.781578

The worst model uses piece_count, min_age, and max_age
It has a validation score of 0.751768

This makes sense if we think that min_age and max_age are
mostly providing the same information, and that the difference
is mainly noise (leading to overfitting), that the best model
would only have one of them

Overall, feature selection does not seem to matter very much
for this dataset + linear regression. So long as we use our
most correlated feature (piece_count), the validation score
doesn't change very much, regardless of which other features
are inc
```

1.7 4. Build and Evaluate a Final Predictive Model

In the cell below, create a list `best_features` which contains the names of the best model features based on the findings of the previous step:

[65]:
```
# Replace None with appropriate code
best_features = ["piece_count", "difficulty_level", "max_age"]
```

Now, we prepare the data for modeling:

[66]:
```
# Run this cell without changes
X_train_final = X_train[best_features]
X_test_final = X_test[best_features]
```

In the cell below, instantiate a `LinearRegression` model called `final_model`, then fit it on the training data and score it on the test data.

[67]:
```
# Replace None with appropriate code

final_model = LinearRegression()

# Fit the model on X_train_final and y_train
final_model.fit(X_train_final, y_train)

# Score the model on X_test_final and y_test
# (use the built-in .score method)
final_model.score(X_test_final, y_test)
```

[67]:
```
0.6542913715071491
```

1.7.1 User-Friendly Metrics

The score above is an r-squared score. Let's compute the RMSE as well, since this would be more applicable to a business audience.

[68]:
```python
# Run this cell without changes
from sklearn.metrics import mean_squared_error

mean_squared_error(y_test, final_model.predict(X_test_final), squared=False)
```

[68]:
```
47.40368797433301
```

What does this value mean in the current business context?

[ ]:
```python
# Replace None with appropriate text
"""
None
"""


#### From GitHub

"""
This means that for an average LEGO set, this algorithm will
be off by about $47. Given that most LEGO sets sell for less
than $100, we would definitely want to have a human double-check
and adjust these prices rather than just allowing the algorithm
to set them
"""
```

1.8 5. Interpret the Final Model

Below, we display the coefficients and intercept for the final model:

[69]:
```python
# Run this cell without changes
print(pd.Series(final_model.coef_, index=X_train_final.columns,␣
 ↪name="Coefficients"))
print()
print("Intercept:", final_model.intercept_)
```

```
piece_count         0.085633
difficulty_level    2.044057
max_age            -0.043271
Name: Coefficients, dtype: float64

Intercept: 9.680845111984297
```

Interpret these values below. What is the pricing algorithm you have developed?

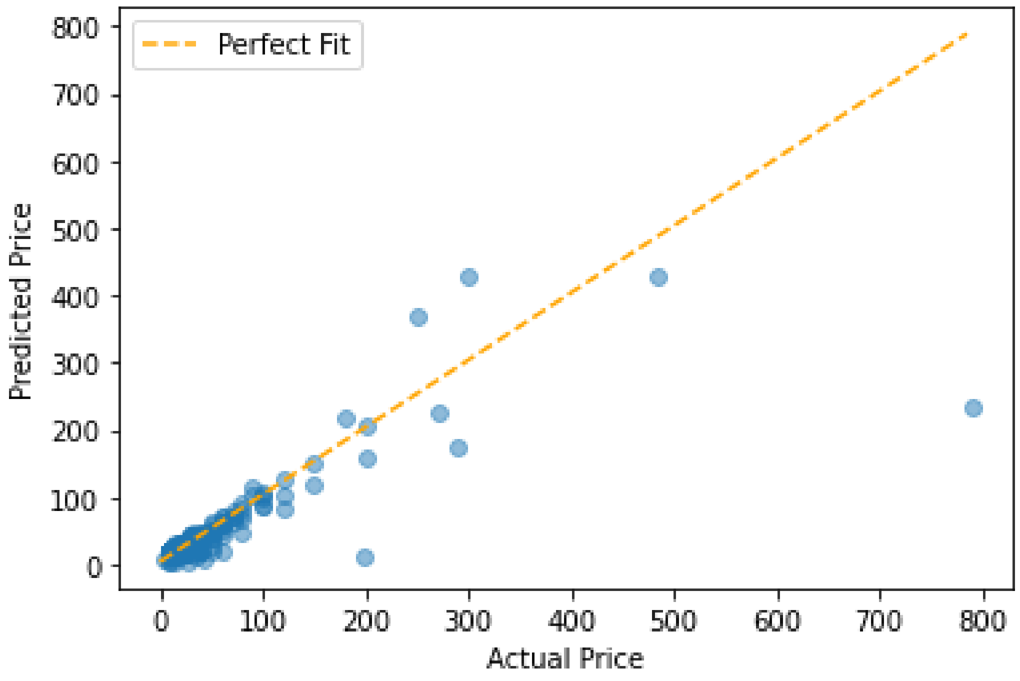
[ ]:
```
# Replace None with appropriate text
"""
None
"""



#### From GitHub

"""
According to our model, the base price for a LEGO set (the
model intercept) is about $9.68. Then for each additional
LEGO piece in the set, the price goes up by $0.09 per piece.
For every year higher that the maximum age is, the price
goes down by about $0.04. Then finally for every increase
of 1 in the difficulty level, the price goes up by about $2.04.
"""
```

Before assuming that these coefficients give us inferential insight into past pricing decisions, we should investigate each of the assumptions of linear regression, in order to understand how much our model violates them.

1.8.1 Investigating Linearity

First, let's check whether the linearity assumption holds.

[70]:
```
# Run this cell without changes

preds = final_model.predict(X_test_final)
fig, ax = plt.subplots()

perfect_line = np.arange(y_test.min(), y_test.max())
ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
ax.scatter(y_test, preds, alpha=0.5)
ax.set_xlabel("Actual Price")
ax.set_ylabel("Predicted Price")
ax.legend();
```



Are we violating the linearity assumption?

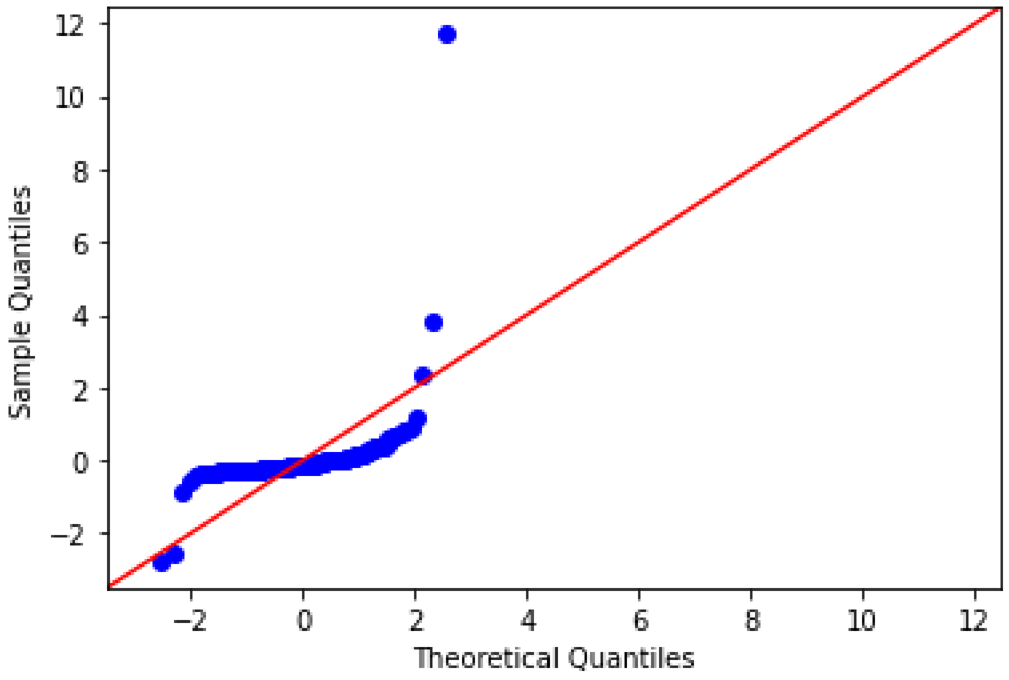
[ ]:
```
# Replace None with appropriate text
"""
It can be seen that there are some cases that are far away from the perfect line
meaning that there are some outliers which will affect our model.
"""

#### From GitHub

"""
We have some outliers that are all over the
place, but in general it looks like we have
a linear relationship (not violating this
assumption)
"""
```

1.8.2 Investigating Normality

Now let's check whether the normality assumption holds for our model.

[71]:
```
# Run this code without changes
import scipy.stats as stats

residuals = (y_test - preds)
```

```
sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True);
```



Are we violating the normality assumption?

[ ]:
```
# Replace None with appropriate text
"""
QQ Plot is not a line so the residual distribution is not normal
"""

#### From GitHub

"""
Our outliers are again causing problems. This
is bad enough that we can probably say that we
are violating the normality assumption
"""
```

1.8.3 Investigating Multicollinearity (Independence Assumption)

Another way to measure multicollinearity is with variance inflation factor (StatsModels documentation here). A "rule of thumb" for VIF is that 5 is too high (i.e. strong multicollinearity).

Run the code below to find the VIF for each feature.

```
[73]: # Run this cell without changes
      from statsmodels.stats.outliers_influence import variance_inflation_factor
      vif = [
          variance_inflation_factor(X_train_final.values, i)
          for i in
             range(X_train_final.shape[1])
            ]
      pd.Series(vif, index=X_train_final.columns, name="Variance Inflation Factor")
```

```
[73]: piece_count          1.923641
      difficulty_level     1.965106
      max_age              1.689177
      Name: Variance Inflation Factor, dtype: float64
```

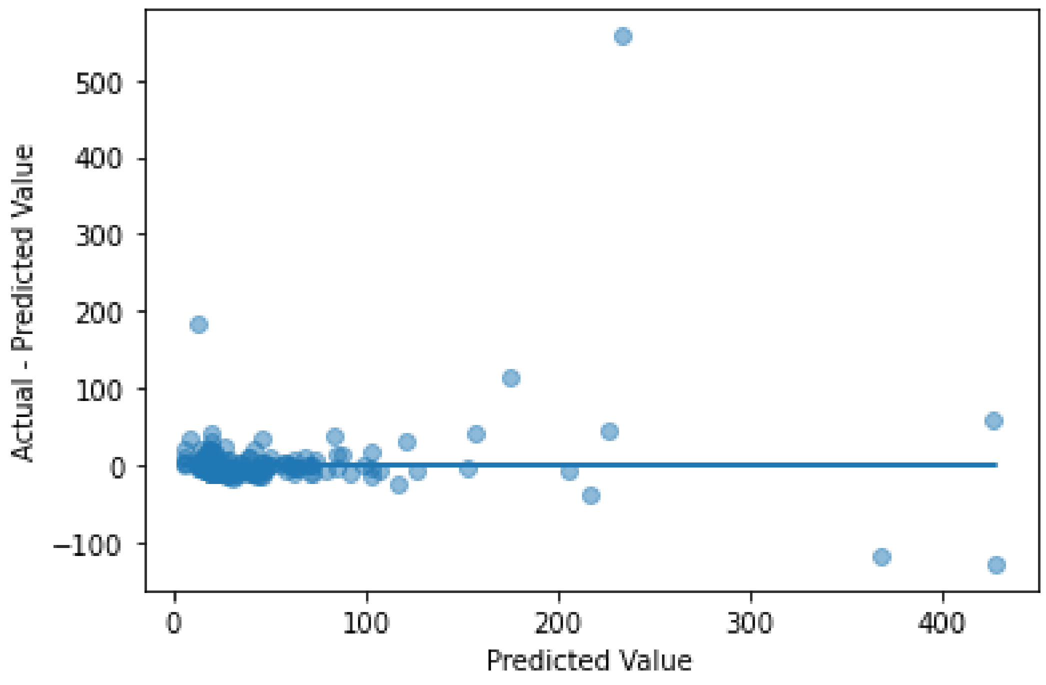
Do we have too high of multicollinearity?

```
[ ]: # Replace None with appropriate text
     """
     Since the values are all below 5, we can say that we do not have
     very high multicolinearity.
     """

     ### From GitHub

     """
     We are below 5 for all features in the final model,
     so we don't have too high of multicollinearity
     """
```

1.8.4 Investigating Homoscedasticity

Now let's check whether the model's errors are indeed homoscedastic or if they violate this principle and display heteroscedasticity.

```
[74]: # Run this cell without changes
      fig, ax = plt.subplots()

      ax.scatter(preds, residuals, alpha=0.5)
      ax.plot(preds, [0 for i in range(len(X_test))])
      ax.set_xlabel("Predicted Value")
      ax.set_ylabel("Actual - Predicted Value");
```



Are we violating the homoscedasticity assumption?

[ ]:
```
# Replace None with appropriate text
"""
the residual scatter plot does not show homoscedasiticity
"""


### From GitHub

"""
This is not the worst "funnel" shape, although
the residuals do seem to differ some based on
the predicted price. We are probably violating
a strict definition of homoscedasticity.
"""
```

1.8.5 Linear Regression Assumptions Conclusion

Given your answers above, how should we interpret our model's coefficients? Do we have a model that can be used for inferential as well as predictive purposes? What might your next steps be?

[ ]:
```
# Replace None with appropriate text
"""
Residuals are not showing homoscedasiticity and their distribution is not normal
meaning that the assumptions of linear regression are not satisfied.
```

```
"""

### From GitHub

"""
Our confidence in these coefficients should not be too high, since
we are violating or close to violating more than one of the
assumptions of linear regression. This really only should be used
for predictive purposes.

A good next step here would be to start trying to figure out why
our outliers behave the way they do. Maybe there is some information
we could extract from the text features that are currently not part
of the model
"""
```

1.9 Summary

Well done! As you can see, regression can be a challenging task that requires you to make decisions along the way, try alternative approaches, and make ongoing refinements.