Analyzing out-of-this world data

Using data collected from the Open Exoplanet Catalogue database:
https://github.com/OpenExoplanetCatalogue/open_exoplanet_catalogue/
(https://github.com/OpenExoplanetCatalogue/open_exoplanet_catalogue/)

Data License

Setup

In [1]:
```
%matplotlib inline

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

EDA

In [2]:
```
planets = pd.read_csv('data/planets.csv')
planets.head()
```

Out[2]:

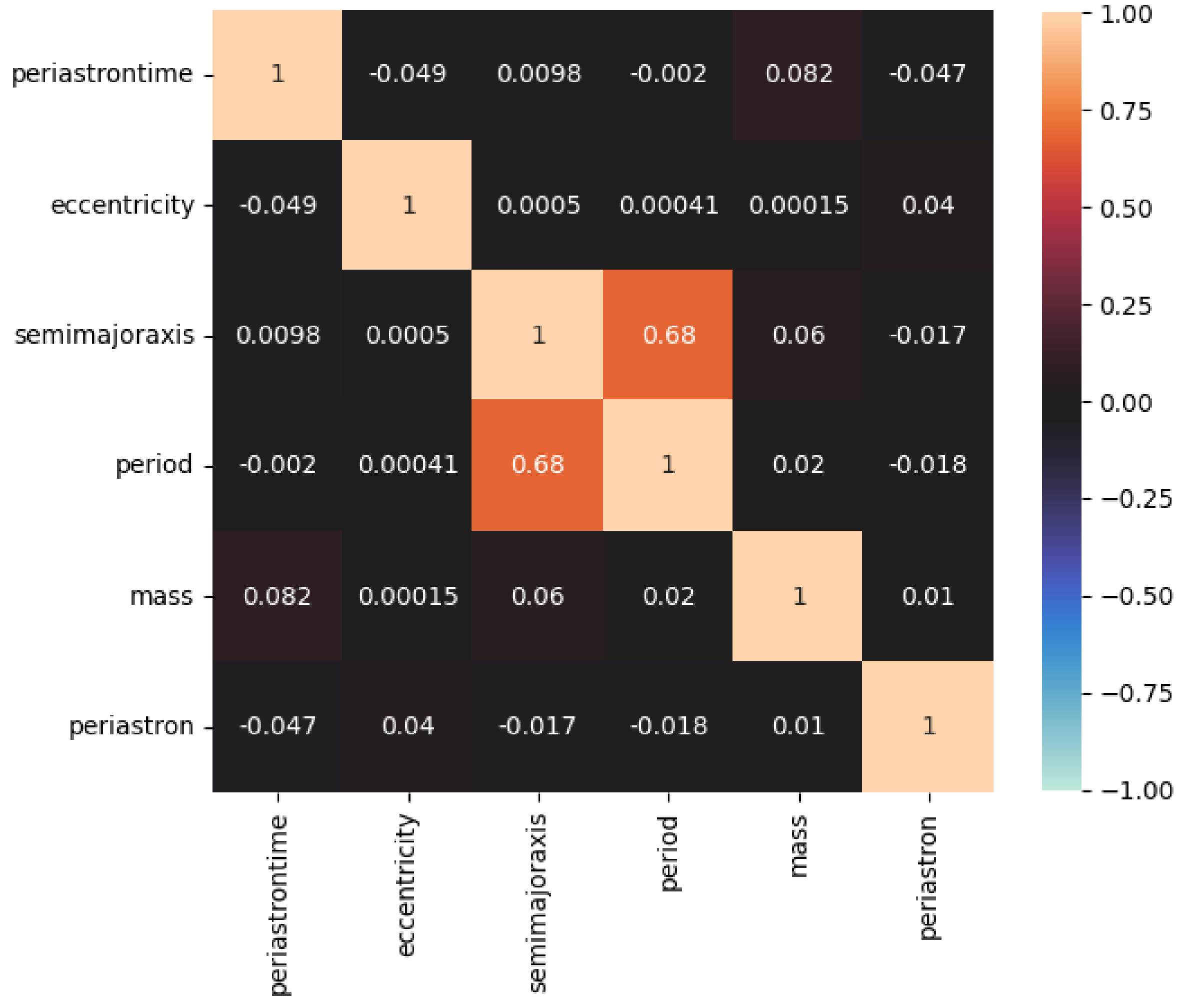
	discoverymethod	description	periastrontime	discoveryyear	eccentricity	semimajoraxis	per
0	RV	11 Com b is a brown dwarf-mass companion to th...	2452899.60	2008.0	0.231	1.290	326
1	RV	11 Ursae Minoris is a star located in the cons...	2452861.04	2009.0	0.080	1.540	516
2	RV	14 Andromedae is an evolved star in the conste...	2452861.40	2008.0	0.000	0.830	185
3	RV	The star 14 Herculis is only 59 light years aw...	NaN	2002.0	0.359	2.864	1766
4	RV	14 Her c is the second companion in the system...	NaN	2006.0	0.184	9.037	9886

Looking for correlated features

It's important to perform an in-depth exploration of the data before modeling. This includes consulting domain experts, looking for correlations between variables, examining distributions, etc. The visualizations covered in chapters 5 and 6 will prove indispensible for this process. One such visualization is the heatmap which we can use to look for correlated features:

In [8]:
```python
fig = plt.figure(figsize=(7, 7))
# create a heatmap
sns.heatmap(
    # correlation matrix of planets (without discovery year)
    planets.drop(columns='discoveryyear').corr(),
    # adjust size, make boxes square, add labels on boxes
    center=0, vmin=-1, vmax=1, square=True, annot=True,
    cbar_kws={'shrink': 0.8}
)
```

Out[8]: <AxesSubplot:>



Looking at Orbit shape

Eccentricity	Orbit Shape
0	Circular
(0, 1)	Elliptical
1	Parabolic
> 1	Hyperbolic

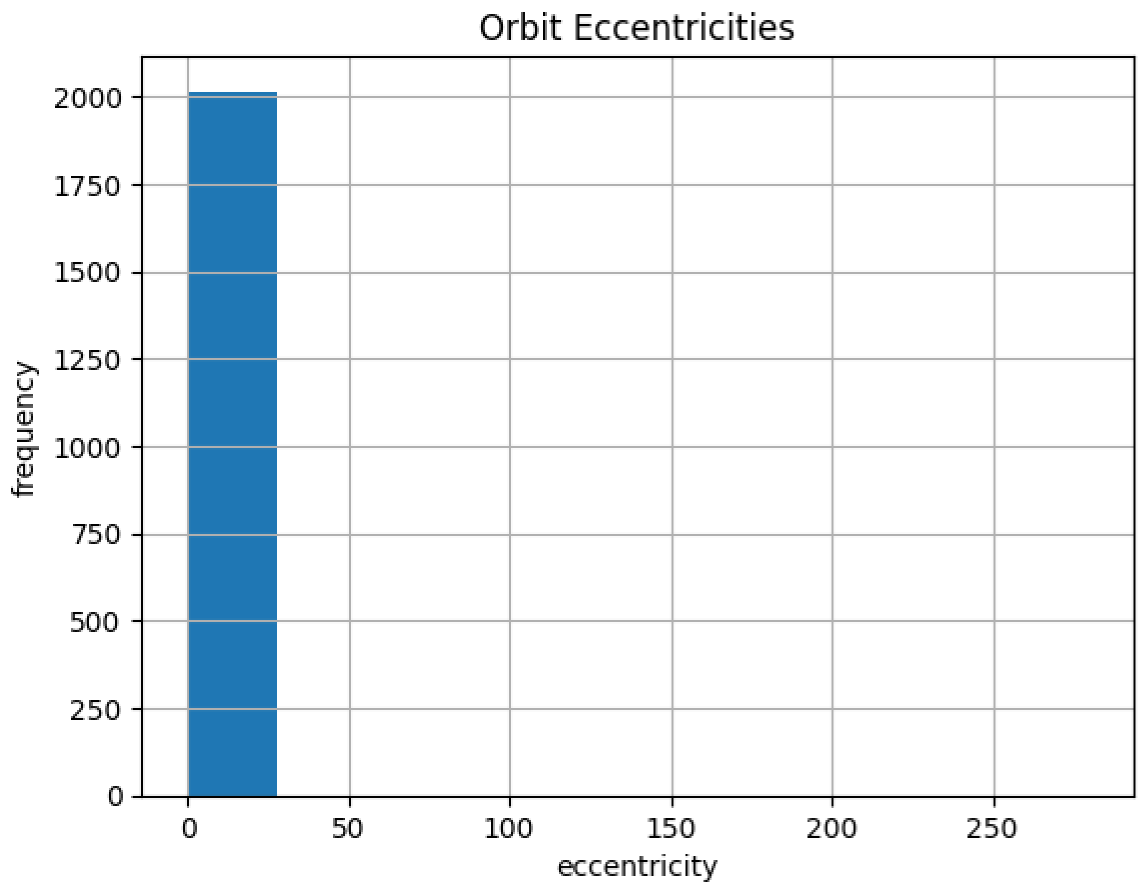
In [9]:
```
# highest and lowest eccentricity
planets.eccentricity.min(), planets.eccentricity.max()
```

Out[9]:
```
(-0.129287, 280.0)
```

All of the planets in the data have circular or elliptical orbits. Let's see the distribution:

In [10]:
```
# histogram of eccentricity with the labels and titles
planets.eccentricity.hist()
plt.xlabel('eccentricity')
plt.ylabel('frequency')
plt.title('Orbit Eccentricities')
```

Out[10]:
```
Text(0.5, 1.0, 'Orbit Eccentricities')
```

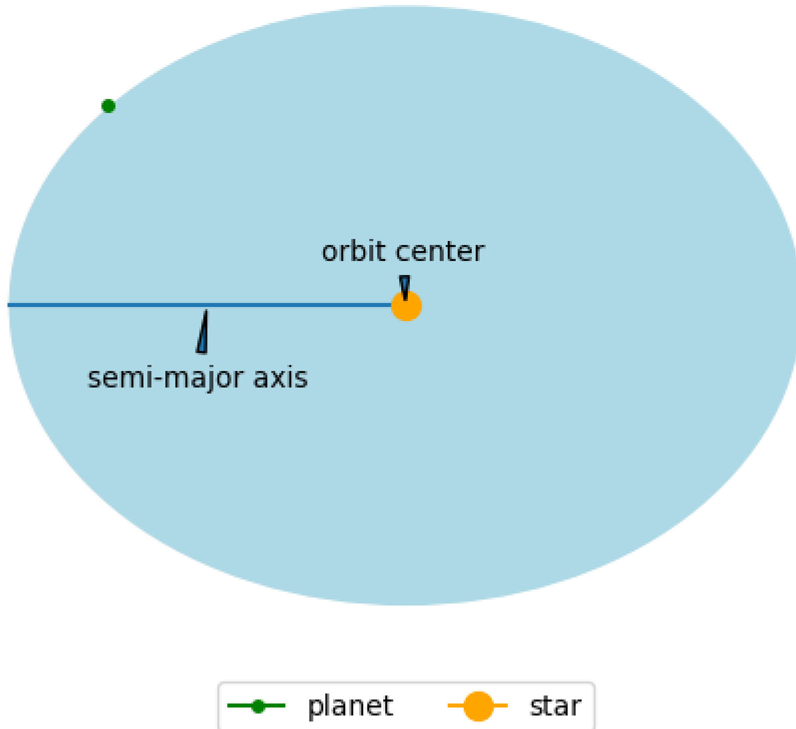


Understanding the semi-major axis

An ellipse, being an elongated circle, has 2 axes: **major** and **minor** for the longest and smallest ones, respectively. The *semi*-major axis is half the major axis. When compared to a circle, the axes are like the diameter crossing the entire shape and the semis are akin to the radius being half the diameter.

In [11]:
```
from visual_aids import misc_viz
misc_viz.elliptical_orbit()
```

Out[11]: <AxesSubplot:>



Checking data values

With just the variables of interest, we have a lot of missing data:

In [12]:
```
planets[['period', 'eccentricity', 'semimajoraxis', 'mass']].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5187 entries, 0 to 5186
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   period         4909 non-null   float64
 1   eccentricity   2015 non-null   float64
 2   semimajoraxis  2600 non-null   float64
 3   mass           2552 non-null   float64
dtypes: float64(4)
memory usage: 162.2 KB
```

If we drop it, we are left with about 30% of it:

In [13]:
```
planets[['period', 'eccentricity', 'semimajoraxis', 'mass']].dropna().shape
```

Out[13]: (1777, 4)

We use `describe()` to get a summary of the variables of interest:

In [14]:
```
planets[['period', 'eccentricity', 'semimajoraxis', 'mass']].describe()
```

Out[14]:

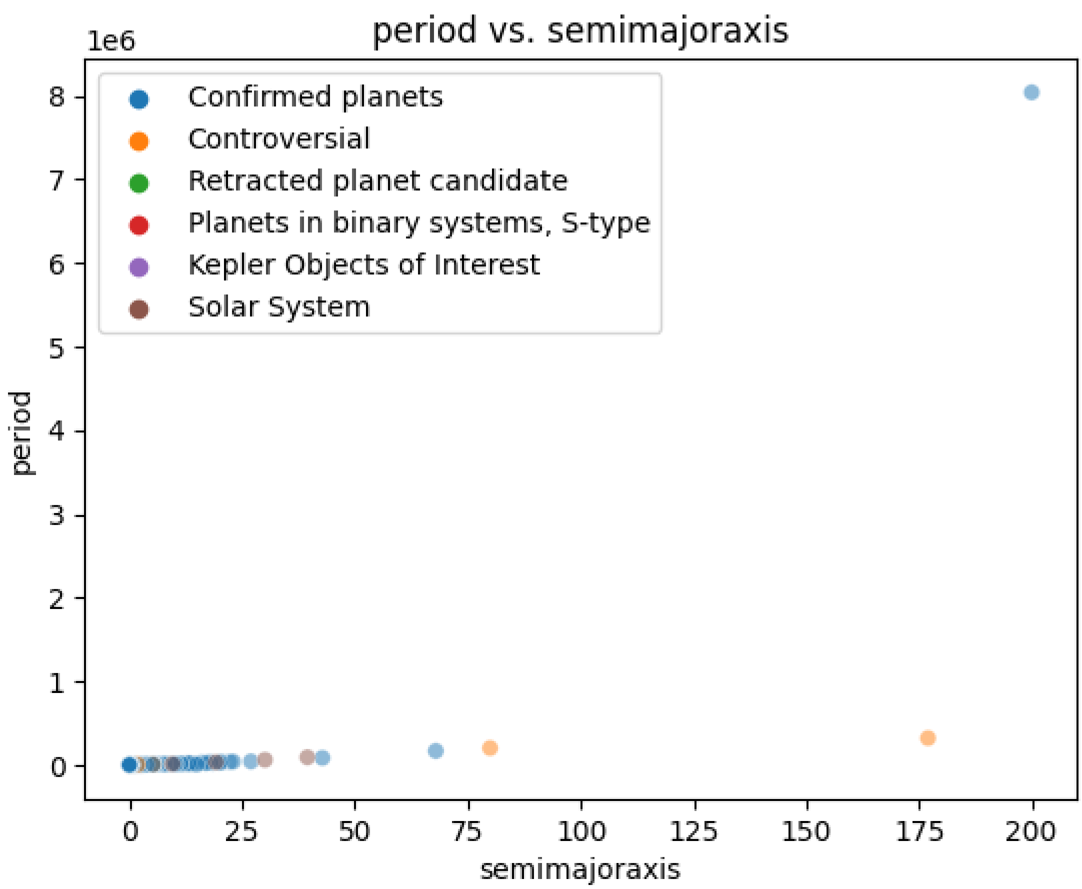
	period	eccentricity	semimajoraxis	mass
count	4.909000e+03	2015.000000	2600.000000	2552.000000
mean	2.189080e+03	0.286252	7.883031	2.292662
std	1.149292e+05	6.237088	159.148610	7.157556
min	6.511500e-02	-0.129287	0.004420	0.000008
25%	4.444480e+00	0.000000	0.050697	0.030950
50%	1.184900e+01	0.080000	0.118390	0.520000
75%	4.252159e+01	0.210000	1.050000	2.090000
max	8.040000e+06	280.000000	6471.000000	263.000000

Visualizing Year and Orbit Length

We have information on the planet list each planet belongs to. We may be wondering: are these planets are controversial because they are so far away?

In [17]:
```
sns.scatterplot(
    x=planets.semimajoraxis,
    y=planets.period,
    hue=planets.list,
    alpha=0.5
)
plt.title('period vs. semimajoraxis')
plt.legend(title='')
```

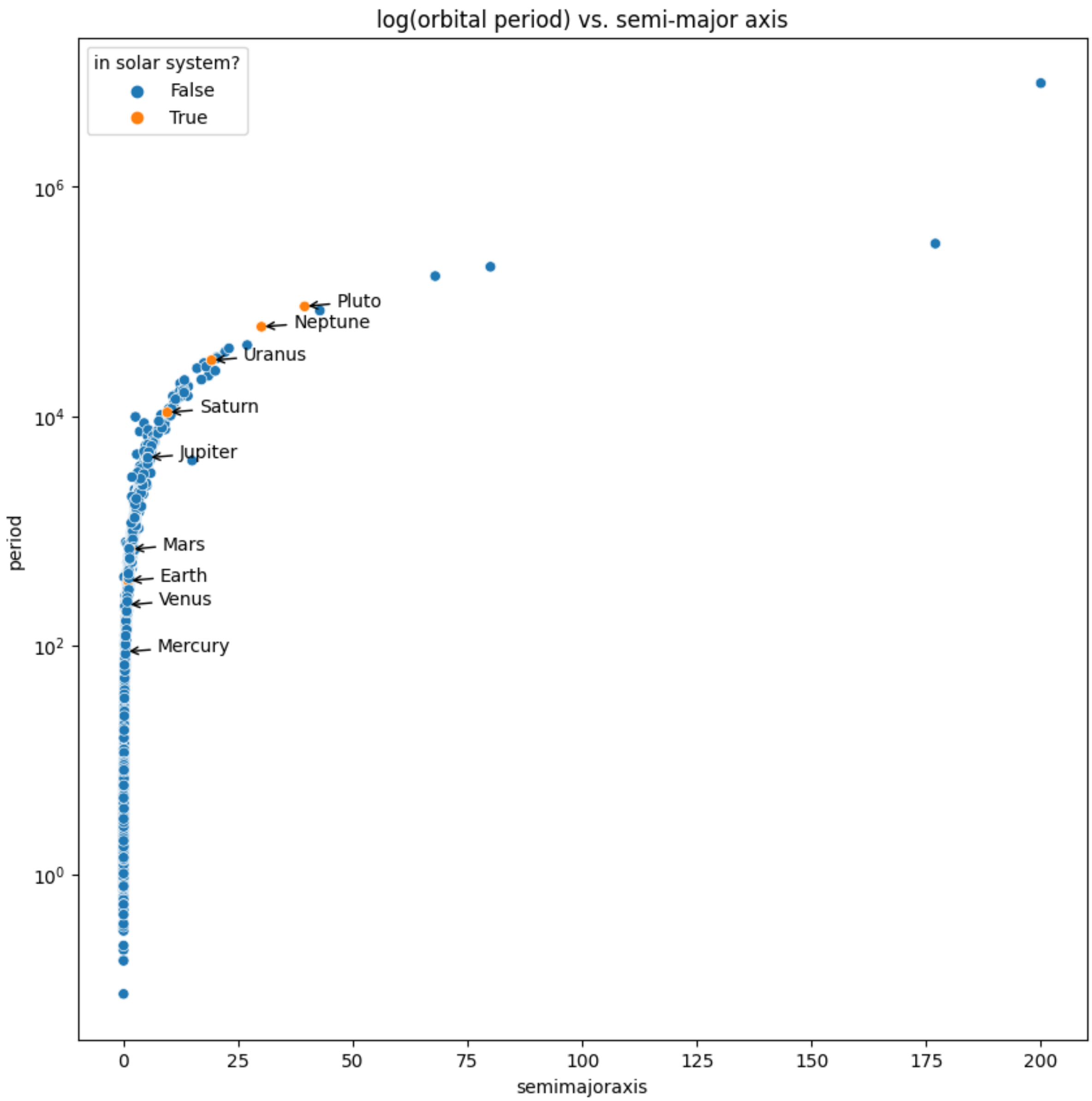
Out[17]: <matplotlib.legend.Legend at 0x130cfc05af0>



Since semi-major axis is highly correlated with period, let's see how the planets compare and label those in our solar system:

In [19]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
in_solar_system = (planets.list == 'Solar System').rename('in solar system?')
sns.scatterplot(
    x=planets.semimajoraxis,
    y=planets.period,
    hue=in_solar_system, # make planets in solar system a different color
    ax=ax
)
# make y axis log scale so its not all grouped up like in above cell
ax.set_yscale('log')
solar_system = planets[planets.list == 'Solar System']

# label each planet in solar system with arrow pointing to it
for planet in solar_system.name:
    data = solar_system.query(f'name == "{planet}"')
    ax.annotate(
        planet,
        (data.semimajoraxis, data.period),
        (7 + data.semimajoraxis, data.period),
        arrowprops=dict(arrowstyle='->')
    )
ax.set_title('log(orbital period) vs. semi-major axis')
```

Out[19]: Text(0.5, 1.0, 'log(orbital period) vs. semi-major axis')



Finding Similar Planets with k-Means Clustering

Since we want to perform clustering to learn more about the data, we will build our pipeline standardizing the data before running k-means and fit it on the all the data:

In [21]:
```
from sklearn.cluster import KMeans
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# scale data to 0 mean
# kmeans clustering algo that splits data into 8 clusters
kmeans_pipeline = Pipeline([
    ('scale', StandardScaler()),
    ('kmeans', KMeans(8, random_state=0))
])
```

Grab the data and fit the model:

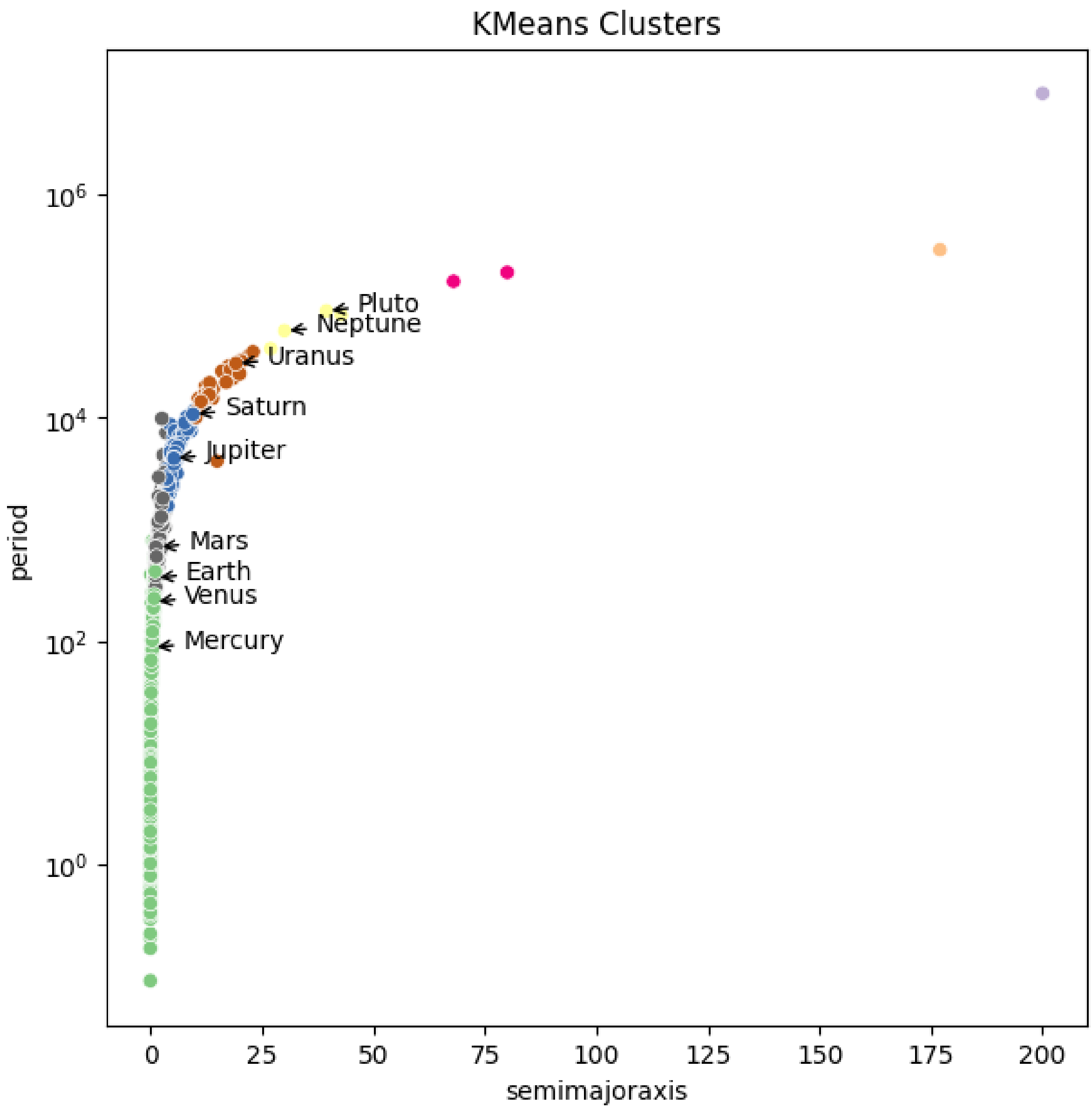
```
In [22]: # grab same features used in previous scatter plot
         kmeans_data = planets[['semimajoraxis', 'period']].dropna()
         kmeans_pipeline.fit(kmeans_data)
```

```
Out[22]: Pipeline(steps=[('scale', StandardScaler()),
                         ('kmeans', KMeans(random_state=0))])
```

We can recreate our plot from before and this time, color by the cluster k-means put each planet in:

In [23]:
```
fig, ax = plt.subplots(1, 1, figsize=(7, 7))

sns.scatterplot(
    x=kmeans_data.semimajoraxis,
    y=kmeans_data.period,
    hue=kmeans_pipeline.predict(kmeans_data),
    ax=ax, palette='Accent'
)

# rest is almost same as previous scatterplot
ax.set_yscale('log')

solar_system = planets[planets.list == 'Solar System']

for planet in solar_system.name:
    data = solar_system.query(f'name == "{planet}"')
    ax.annotate(
        planet,
        (data.semimajoraxis, data.period),
        (7 + data.semimajoraxis, data.period),
        arrowprops=dict(arrowstyle='->')
    )
ax.get_legend().remove()
ax.set_title('KMeans Clusters')
```

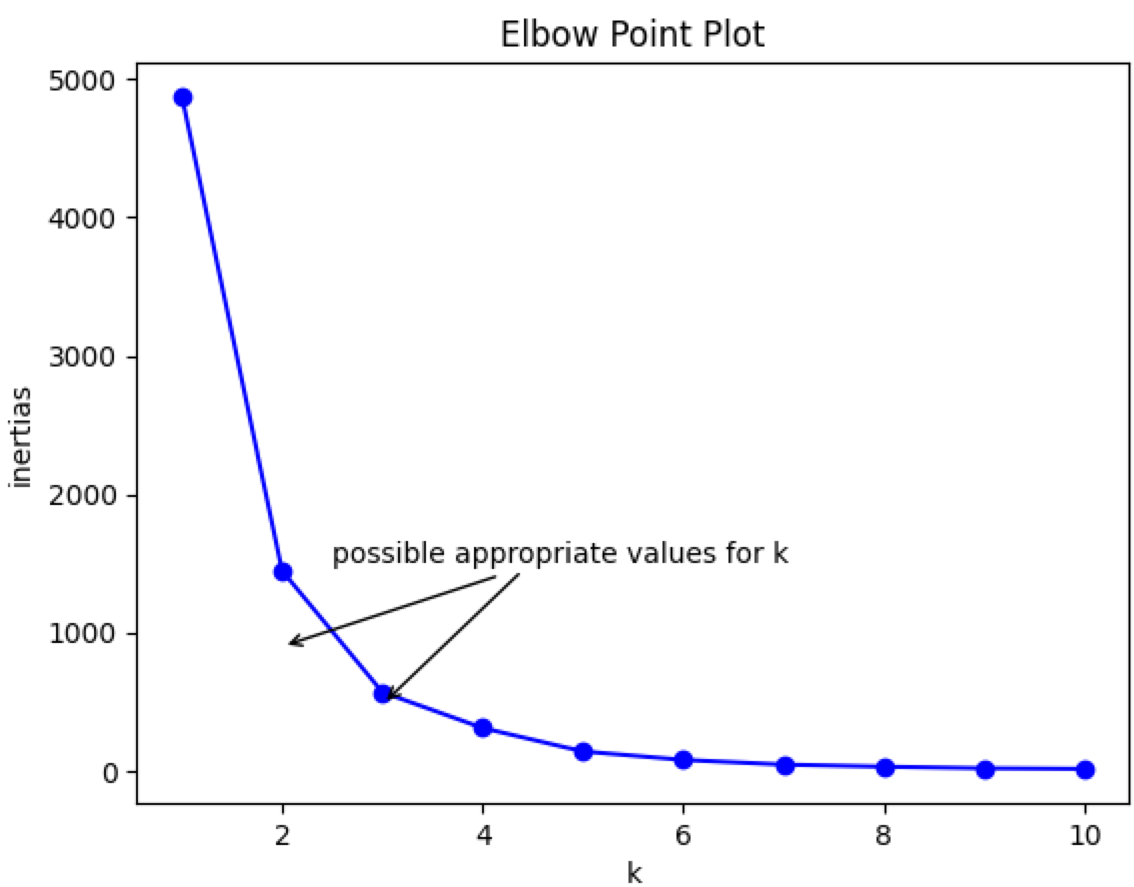
```
Out[23]: Text(0.5, 1.0, 'KMeans Clusters')
```



The elbow point method can be used to pick a good value for `k`. This value will be were we begin to see diminishing returns in the reduction of the value of the objective function:

In [25]:
```
from ml_utils.elbow_point import elbow_point

# elbow will help determine the optimal number of clusters (k) for kmeans clus
tering
ax = elbow_point(
    kmeans_data,
    Pipeline([
        ('scale', StandardScaler()),
        ('kmeans', KMeans(random_state=0))
    ])
)
ax.annotate(
    'possible appropriate values for k', xy=(2, 900), xytext=(2.5, 1500),
    arrowprops=dict(arrowstyle='->')
)
ax.annotate(
    '', xy=(3, 480), xytext=(4.4, 1450), arrowprops=dict(arrowstyle='->')
)
```

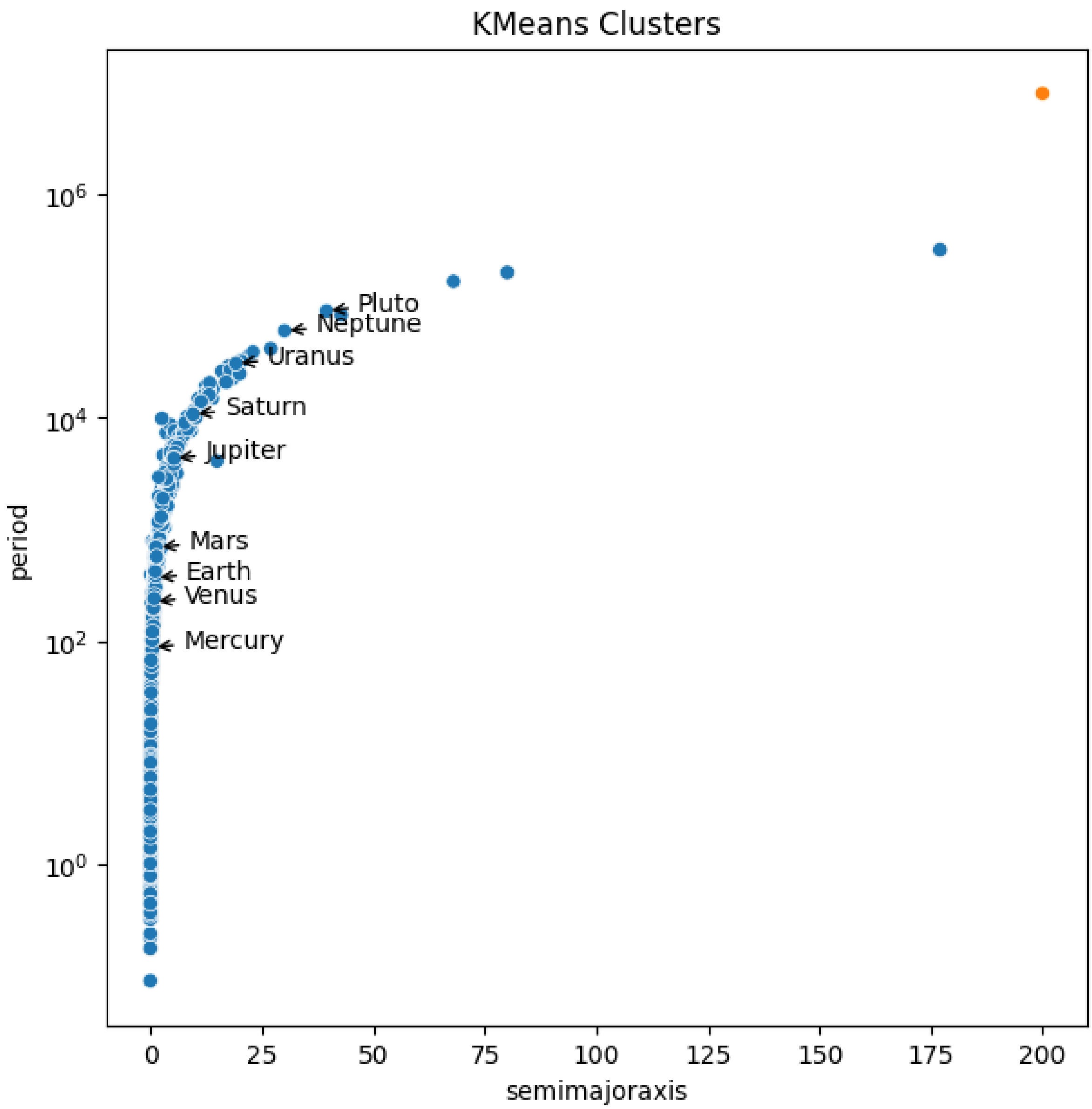
Out[25]:
```
Text(4.4, 1450, '')
```



k-means with the "optimal" k of 2

In [26]:
```
kmeans_pipeline_2 = Pipeline([
    ('scale', StandardScaler()),
    ('kmeans', KMeans(2, random_state=0))
]).fit(kmeans_data)

fig, ax = plt.subplots(1, 1, figsize=(7, 7))
sns.scatterplot(
    x=kmeans_data.semimajoraxis,
    y=kmeans_data.period,
    hue=kmeans_pipeline_2.predict(kmeans_data),
    ax=ax
)
ax.set_yscale('log')
solar_system = planets[planets.list == 'Solar System']
for planet in solar_system.name:
    data = solar_system.query(f'name == "{planet}"')
    ax.annotate(
        planet,
        (data.semimajoraxis, data.period),
        (7 + data.semimajoraxis, data.period),
        arrowprops=dict(arrowstyle='->')
    )
ax.get_legend().remove()
ax.set_title('KMeans Clusters')
```

Out[26]: Text(0.5, 1.0, 'KMeans Clusters')



Visualizing the cluster space

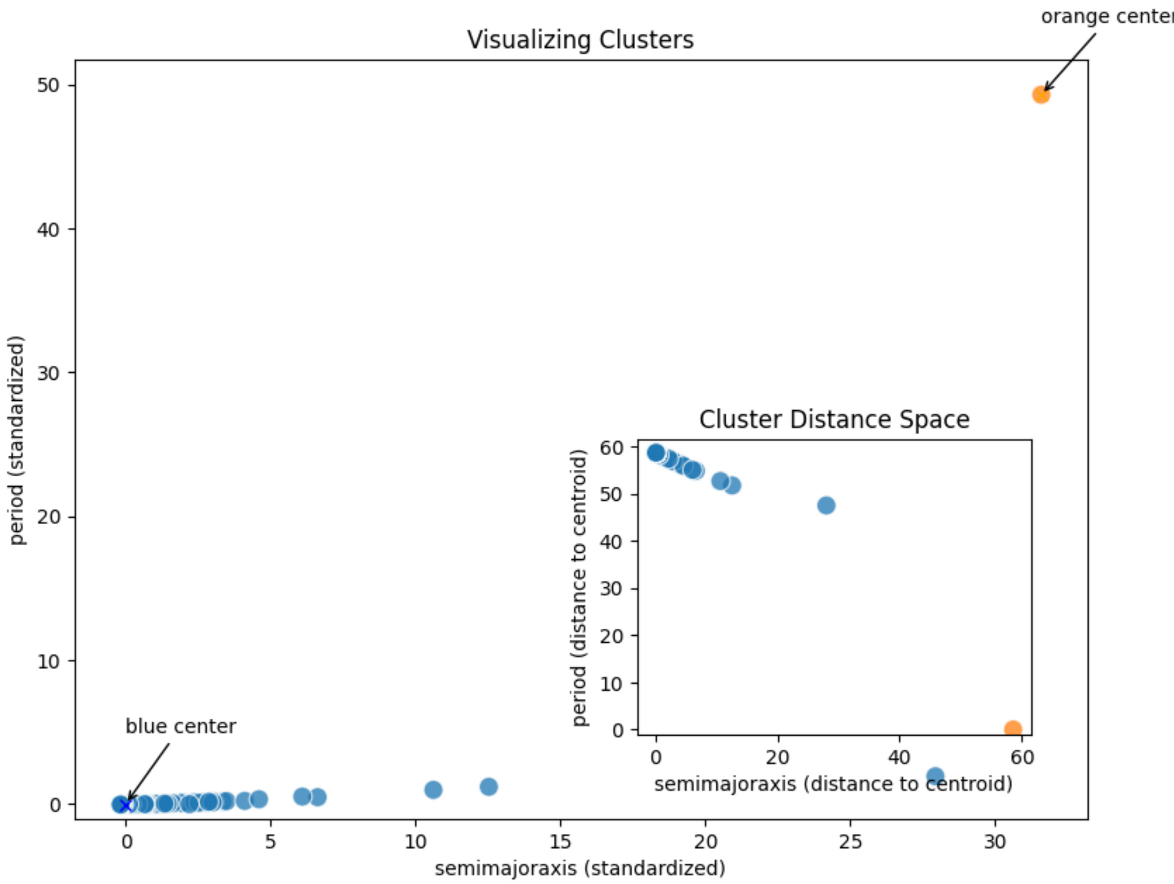
Since we standardized the data, looking at the centers tells us the second cluster contains "outliers" for period and semi-major axis:

In [27]:
```
kmeans_pipeline_2.named_steps['kmeans'].cluster_centers_
```

Out[27]:
```
array([[-1.29632815e-02, -2.02245973e-02],
       [ 3.15915171e+01,  4.92873436e+01]])
```

We can also visualize the clusters:

In [28]:
```
# set up layout
fig = plt.figure(figsize=(8, 6))
outside = fig.add_axes([0.1, 0.1, 0.9, 0.9])
inside = fig.add_axes([0.6, 0.2, 0.35, 0.35])

# scaled data and cluster distance data
scaled = kmeans_pipeline_2.named_steps['scale']\
    .fit_transform(kmeans_data)
cluster_distances = kmeans_pipeline_2\
    .fit_transform(kmeans_data)

for ax, data, title, axes_labels in zip(
    [outside, inside], [scaled, cluster_distances],
    ['Visualizing Clusters', 'Cluster Distance Space'],
    ['standardized', 'distance to centroid']
):
    sns.scatterplot(
        x=data[:,0], y=data[:,1], ax=ax, alpha=0.75, s=100,
        hue=kmeans_pipeline_2.named_steps['kmeans'].labels_
    )

    ax.get_legend().remove()
    ax.set_title(title)
    ax.set_xlabel(f'semimajoraxis ({axes_labels})')
    ax.set_ylabel(f'period ({axes_labels})')
    ax.set_ylim(-1, None)

# add the centroids to the outside plot
cluster_centers = kmeans_pipeline_2.named_steps['kmeans'].cluster_centers_
for color, centroid in zip(['blue', 'orange'], cluster_centers):
    outside.plot(*centroid, color=color, marker='x')
    outside.annotate(
        f'{color} center', xy=centroid, xytext=centroid + [0, 5],
        arrowprops=dict(arrowstyle='->')
    )
```



Notes on the `scikit-learn` API

Method	Action	Used when...
`fit()`	Train the model or preprocessor	Modeling, preprocessing
`transform()`	Transform the data into the new space	Clustering, preprocessing
`fit_transform()`	Run `fit()` , followed by `transform()`	Clustering, preprocessing
`score()`	Evaluate the model using the default scoring method	Modeling
`predict()`	Use model to predict output values for given inputs	Modeling
`fit_predict()`	Run `fit()` , followed by `predict()`	Modeling
`predict_proba()`	Like `predict()` , but returns the probability of belonging to each class	Classification

Evaluation of model

There are many metrics to choose from, but since we don't know the true labels of our data, we can only use unsupervised ones. We will use a few different metrics to get a more well-rounded view of our performance:

Silhouette Score

- true labels not known
- higher = better defined (more separated) clusters
- -1 is worst, 1 is best, near 0 indicates overlapping clusters

In [29]:
```
from sklearn.metrics import silhouette_score
silhouette_score(kmeans_data, kmeans_pipeline.predict(kmeans_data))
```

Out[29]:
```
0.7809782696744539
```

Davies-Bouldin Score

- true labels not known
- ratio of within-cluster distances to between-cluster distances
- zero is the best partition

In [30]:
```
from sklearn.metrics import davies_bouldin_score
davies_bouldin_score(kmeans_data, kmeans_pipeline.predict(kmeans_data))
```

Out[30]:
```
0.40214928555175533
```

Calinski and Harabasz Score

- true labels not known
- higher = better defined (more separated) clusters

In [31]:
```
from sklearn.metrics import calinski_harabasz_score
calinski_harabasz_score(kmeans_data, kmeans_pipeline.predict(kmeans_data))
```

Out[31]:
```
4671613.5913595
```

Predicting Length of Year in Earth Days (Period)

1. separate x and y data, dropping nulls
2. create the training and testing sets
3. train a linear regression model (no preprocessing since we want to interpret the coefficients)
4. isolate the coefficients from the model
5. evaluate the model

Step 1:

In [32]:
```
data = planets[
    ['semimajoraxis', 'period', 'mass', 'eccentricity']
].dropna()
X = data[['semimajoraxis', 'mass', 'eccentricity']]
y = data.period
```

Step 2:

In [33]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=0
)
```

Linear Regression

Step 3:

In [34]:
```
from sklearn.linear_model import LinearRegression
lm = LinearRegression().fit(X_train, y_train)
```

Get equation

Step 4:

In [35]:
```
# get intercept
lm.intercept_
```

Out[35]:
```
-1193.3700241298125
```

In [36]:
```
# get coefficients
[(col, coef) for col, coef in zip(X_train.columns, lm.coef_)]
```

Out[36]:
```
[('semimajoraxis', 2023.8043609058575),
 ('mass', -46.10431511684035),
 ('eccentricity', 1.68975045278567)]
```

Evaluation of model

Step 5

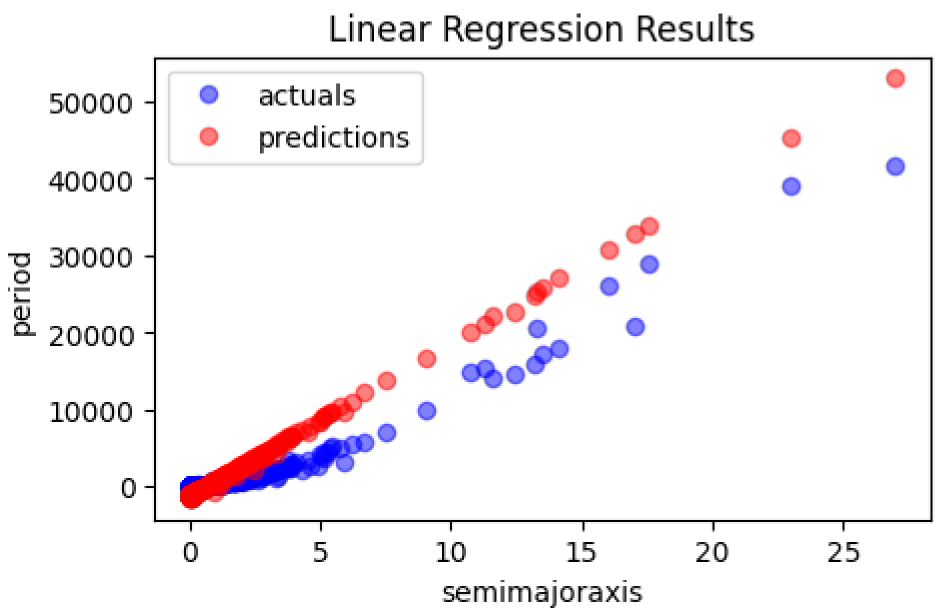
In order to evaluate our model's predictions against the actual values, we need to make predictions for the test set:

In [37]:
```
preds = lm.predict(X_test)
```

We can then plot the predictions and actual values:

In [38]:
```
fig, axes = plt.subplots(1, 1, figsize=(5, 3))
axes.plot(X_test.semimajoraxis, y_test, 'ob', label='actuals', alpha=0.5)
axes.plot(X_test.semimajoraxis, preds, 'or', label='predictions', alpha=0.5)
axes.set(xlabel='semimajoraxis', ylabel='period')
axes.legend()
axes.set_title('Linear Regression Results')
```

Out[38]:
```
Text(0.5, 1.0, 'Linear Regression Results')
```



The correlation between the predictions and the actual values tells us they trend together, but we need to look at other metrics to quantify the errors our model makes:

In [39]:
```
np.corrcoef(y_test, preds)[0][1]
```

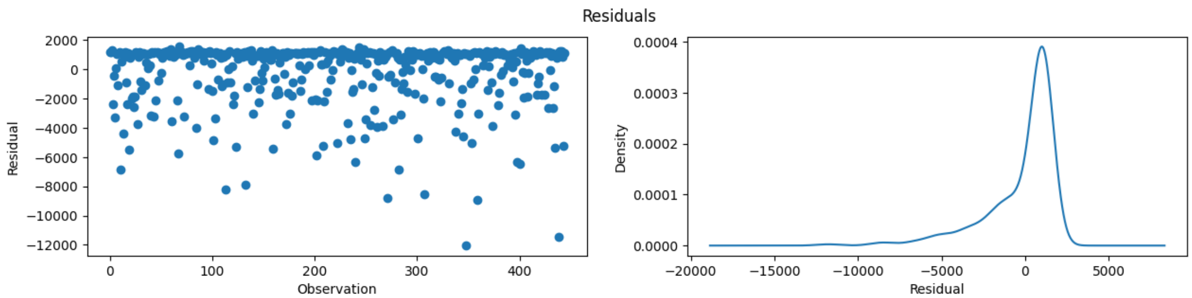
Out[39]:
```
0.9663385303645294
```

Residuals

Our residuals have no pattern (left subplot); however, the distribution has some negative skew, and the residuals aren't quite centered around zero (right subplot):

In [40]:
```
from ml_utils.regression import plot_residuals

plot_residuals(y_test, preds)
```

Out[40]:
```
array([<AxesSubplot:xlabel='Observation', ylabel='Residual'>,
       <AxesSubplot:xlabel='Residual', ylabel='Density'>], dtype=object)
```



R^2

By default, the `score()` method of the `LinearRegression` object will give us the R^2:

In [41]:
```
lm.score(X_test, y_test)
```

Out[41]:
```
0.7222350988053549
```

If not, we can use the `r2_score()` function from `sklearn.metrics` :

In [42]:
```
from sklearn.metrics import r2_score
r2_score(y_test, preds)
```

Out[42]:
```
0.7222350988053549
```

Adjusted R^2

R^2 increases when we add regressors whether or not they actually improve the model. Adjusted R^2 penalizes additional regressors to address this:

In [43]:
```
from ml_utils.regression import adjusted_r2
adjusted_r2(lm, X_test, y_test)
```

Out[43]:
```
0.7203455416543709
```

Problems with R^2

R^2 doesn't tell us about the prediction errors or if we specified the model correctly. Consider Anscombe's quartet from chapter 1:

Anscombe's Quartet

All four data sets have the same summary statistics (mean, standard deviation, correlation coefficient), despite having different data:

In [44]:
```
anscombe = sns.load_dataset('anscombe').groupby('dataset')
anscombe.describe()
```

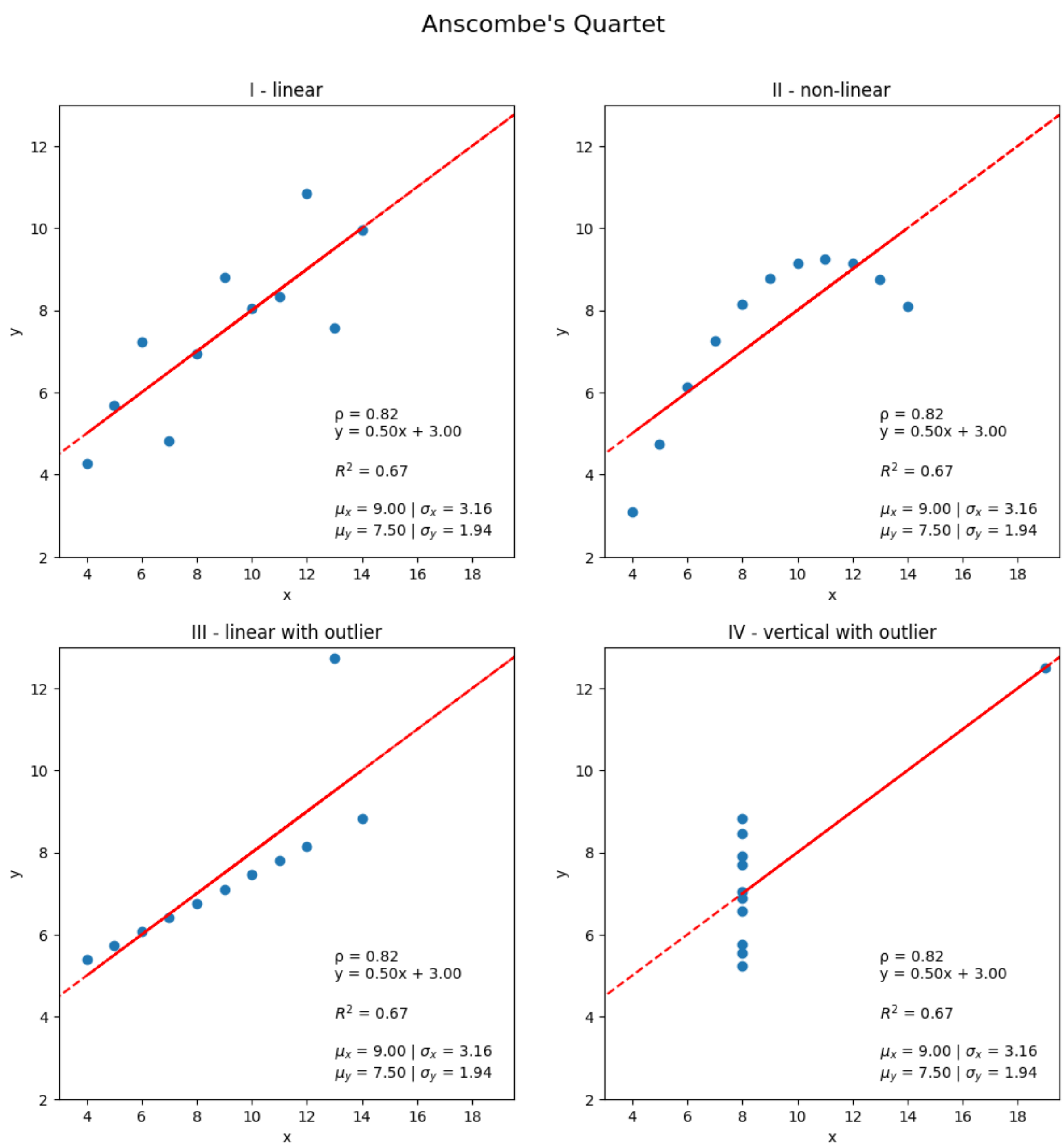
Out[44]:

	x								y				
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25
dataset													
I	11.0	9.0	3.316625	4.0	6.5	9.0	11.5	14.0	11.0	7.500909	2.031568	4.26	6.3
II	11.0	9.0	3.316625	4.0	6.5	9.0	11.5	14.0	11.0	7.500909	2.031657	3.10	6.6
III	11.0	9.0	3.316625	4.0	6.5	9.0	11.5	14.0	11.0	7.500000	2.030424	5.39	6.2
IV	11.0	9.0	3.316625	8.0	8.0	8.0	8.0	19.0	11.0	7.500909	2.030579	5.25	6.1

When fitted with a regression line, they all have the same R^2 despite some of them not indicating a linear relationship between x and y:

In [45]:
```python
from visual_aids import stats_viz
stats_viz.anscombes_quartet(r_squared=True)
```

Out[45]:
```
array([<AxesSubplot:title={'center':'I - linear'}, xlabel='x', ylabel='y'>,
       <AxesSubplot:title={'center':'II - non-linear'}, xlabel='x', ylabel
='y'>,
       <AxesSubplot:title={'center':'III - linear with outlier'}, xlabel='x',
ylabel='y'>,
       <AxesSubplot:title={'center':'IV - vertical with outlier'}, xlabel
='x', ylabel='y'>],
      dtype=object)
```



Explained Variance

The percentage of the variance in the data is explained by our model:

In [46]:
```
from sklearn.metrics import explained_variance_score
explained_variance_score(y_test, preds)
```

Out[46]: 0.7248170039385958

Mean Absolute Error (MAE)

This gives us an idea of how far off our predictions are on average (in Earth days):

In [47]:
```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, preds)
```

Out[47]: 1510.3287900809842

Root Mean Squared Error (RMSE)

We can use this to punish large errors more:

In [48]:
```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(y_test, preds))
```

Out[48]: 2154.8786141301657

Median Absolute Error

We can also look at the median absolute error to ignore any outliers in prediction errors and get a better picture of our error:

In [49]:
```
from sklearn.metrics import median_absolute_error
median_absolute_error(y_test, preds)
```

Out[49]: 1104.6059146033715

← Chapter 8 (../../ch_08/anomaly_detection.ipynb) Planet Data Collection (./planet_data_collection.ipynb) Preprocessing (./preprocessing.ipynb) Red Wine (./red_wine.ipynb) Red + White Wine (./wine.ipynb)
</div> Solutions (../../solutions/ch_09/exercise_1.ipynb) Chapter 10 → (../ch_10/red_wine.ipynb)
