Interactions

Introduction

In this section, you'll learn about interactions and how to account for them in regression models.

Objectives

You will be able to:

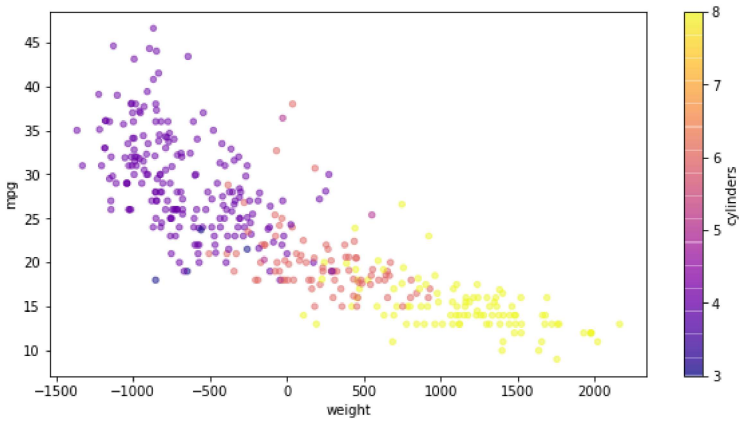
- Determine if an interaction term would be useful for a specific model or set of data
- Create interaction terms out of independent variables in linear regression
- Interpret coefficients of linear regression models that contain interaction terms

Why Create an Interaction Term?

The goal of creating an interaction term is to capture the *interaction* between multiple independent variables in your linear regression model. This applies to all combinations of numeric and categorical features, but we'll start with one numeric feature and one categorical feature because that tends to be the most straightforward to visualize.

For example, this graph shows two independent variables at the same time. First, `weight` is represented along the x-axis. Second, `cylinders` is represented by the color of the data point. We have also **centered** the weight and set 1970 = 0 for model year values to create more interpretable intercepts in our upcoming models.

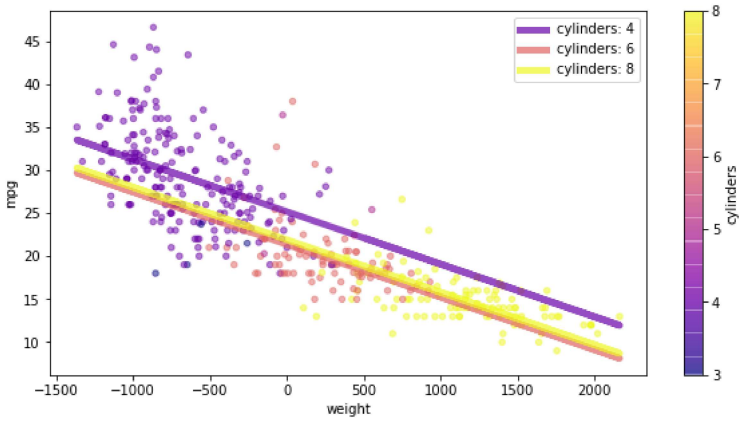
In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
%matplotlib nbagg
%matplotlib inline

data = pd.read_csv("auto-mpg.csv")
data["weight"] = data["weight"] - data["weight"].mean()
data["model year"] = data["model year"] - 70

fig, ax = plt.subplots(figsize=(10,5))
data.plot.scatter(x="weight", y="mpg", c="cylinders", cmap="plasma", alpha=0.5, ax=ax);
```



If we treat `cylinders` as a categorical variable and do not create any interaction terms, we are essentially saying that the relationship looks like this, with **parallel** slopes for each number of cylinders:

In [2]:
```
def plot_fit_lines(data, results, column, categories, interactions, cmap="plasma"):
    # Extract some shorter variable names for readability
    x = data["weight"]
    beta_0 = results.params["const"]
    beta_1 = results.params["weight"]

    # Set up scatter plot
    fig, ax = plt.subplots(figsize=(10,5))
    data.plot.scatter(x="weight", y="mpg", c=column, cmap=cmap, alpha=0.5, ax=ax)

    colors = cm.get_cmap(cmap)
    min_cat = data[column].min()
    max_cat = data[column].max()
    range_cat = max_cat - min_cat

    # For each category, calculate and graph the fit line
    for cat in categories:
        if cat in interactions:
            fit_line = beta_0 + \
                (beta_1 + results.params[f"weight x {column}_{cat}"]) * x + \
                results.params[f"{column}_{cat}"]
        else:
            fit_line = beta_0 + beta_1 * x + results.params[f"{column}_{cat}"]

        ax.plot(
            x,
            fit_line,
            color=colors((cat - min_cat)/range_cat),
            label=f"{column}: {cat}",
            linewidth=5,
            alpha=0.7)

    ax.legend();
```

In [3]:
```
import statsmodels.api as sm

# Prepare data for modeling
y = data["mpg"]
X_no_interaction = data[["cylinders", "weight"]].copy()
X_no_interaction = pd.get_dummies(X_no_interaction, columns=["cylinders"], drop_first=True)

# Build model and get results
no_interaction_model = sm.OLS(y, sm.add_constant(X_no_interaction))
no_interaction_results = no_interaction_model.fit()

# Plot the lines generated by the model for each selected category
plot_fit_lines(data, no_interaction_results, "cylinders", [4, 6, 8], [])
```



(Note that there are actually vehicles with 3, 4, 5, 6, or 8 cylinders, but since there are very few samples for 3 and 5 cylinders we'll just focus on those with 4, 6, or 8 cylinders.)

In formula form, we are saying:

$$ \large{ \text{mpg} = \beta_0 + \beta_1 \times \text{weight} + \begin{cases} \beta_\text{(4 cylinders)} & \text{if 4 cylinders}, or\\ \beta_\text{(6 cylinders)} & \text{if 6 cylinders}, or\\ \beta_\text{(8 cylinders)} & \text{if 8 cylinders}. \end{cases}} $$

If you had drawn the best-fit lines yourself, would you have drawn ones that look like this?

In particular, if you focus only on the cars with **4 cylinders**, does that line actually make sense with the scattered dots of that color? No, it seems like there should be a line with a steeper slope for 4 cylinders compared to 6 cylinders or 8 cylinders.

Adding an ***interaction term*** means that these slopes can differ depending on the number of cylinders, instead of just changing the intercepts! In other words, interaction terms mean that **the lines do not need to be parallel**.

Adding an Interaction Term

Let's say that we are specifically concerned about the slope for cars with 4 cylinders. To do this, we'll add an interaction term that **multiplies** `weight` by `cylinders_4` .

Currently the data looks like this:

In [4]:
```
X_no_interaction
```

Out[4]:

	weight	cylinders_4	cylinders_5	cylinders_6	cylinders_8
0	526.415816	0	0	0	1
1	715.415816	0	0	0	1
2	458.415816	0	0	0	1
3	455.415816	0	0	0	1
4	471.415816	0	0	0	1
...	...	...	...	...	...
387	-187.584184	1	0	0	0
388	-847.584184	1	0	0	0
389	-682.584184	1	0	0	0
390	-352.584184	1	0	0	0
391	-257.584184	1	0	0	0

392 rows × 5 columns

We can add an interaction term between `weight` and `cylinders_4` like this:

In [5]:
```
X_interaction = X_no_interaction.copy()
X_interaction["weight x cylinders_4"] = X_interaction["weight"] * X_interaction["cylinders_4"]
X_interaction
```

Out[5]:

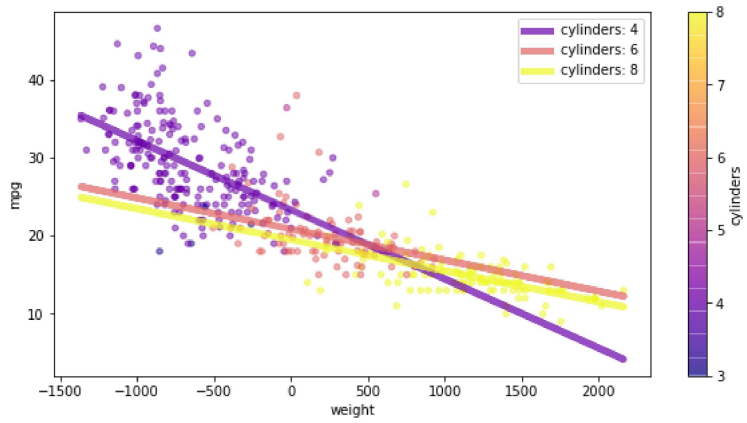
	weight	cylinders_4	cylinders_5	cylinders_6	cylinders_8	weight x cylinders_4
0	526.415816	0	0	0	1	0.000000
1	715.415816	0	0	0	1	0.000000
2	458.415816	0	0	0	1	0.000000
3	455.415816	0	0	0	1	0.000000
4	471.415816	0	0	0	1	0.000000
...	...	...	...	...	...	...
387	-187.584184	1	0	0	0	-187.584184
388	-847.584184	1	0	0	0	-847.584184
389	-682.584184	1	0	0	0	-682.584184
390	-352.584184	1	0	0	0	-352.584184
391	-257.584184	1	0	0	0	-257.584184

392 rows × 6 columns

Then build a model and repeat the process of graphing the slopes:

In [6]:
```
# Build model and get results
interaction_model = sm.OLS(y, sm.add_constant(X_interaction))
interaction_results = interaction_model.fit()
```

In [7]:
```
# Plot fit lines, this time with an interaction term
plot_fit_lines(data, interaction_results, "cylinders", [4, 6, 8], [4])
```



In formula form, we are now saying:

$$ \large{ \text{mpg} = \beta_0 + \begin{cases} \beta_1 \times \text{weight} \times \beta_\text{interaction} + \beta_\text{(4 cylinders)} & \text{if 4 cylinders}, or\\ \beta_1 \times \text{weight} + \beta_\text{(6 cylinders)} & \text{if 6 cylinders}, or\\ \beta_1 \times \text{weight} + \beta_\text{(8 cylinders)} & \text{if 8 cylinders}. \end{cases}} $$

Note that βs in this formula are not the same as in the previous formula. This is clear from the fact that the intercept for the 4 cylinders line $\beta_\text{(4 cylinders)}$ is higher in this graph than in the previous graph. Just like adding an additional independent variable to a multiple regression, adding an interaction fundamentally changes the model and the other coefficients will adjust accordingly.

Evaluating the Addition of the Interaction Term

As with any modification to the independent variables of a regression model, we want to ensure that this actually improves the model. Does the interaction term (`weight` times `cylinders_4`) contribute different information to the model, compared to `weight` and `cylinders_4` individually?

The fact that this line *looks* more correct is a good indicator that this interaction term was a good choice. The other things we'll want to check are:

1. Whether the interaction term coefficient is statistically significant
2. Whether this improved the model metrics overall

Checking Coefficient Statistical Significance

We can check the p-values for all model coefficients using the code below:

In [8]:
```
pvalues_df = pd.DataFrame(interaction_results.pvalues, columns=["p-value"])
pvalues_df["p < 0.05"] = pvalues_df["p-value"] < 0.05
pvalues_df
```

Out[8]:

	p-value	p < 0.05
const	2.832364e-17	True
weight	1.029030e-07	True
cylinders_4	1.866970e-02	True
cylinders_5	2.172974e-03	True
cylinders_6	2.216976e-01	False
cylinders_8	6.045203e-01	False
weight x cylinders_4	1.478591e-05	True

The `weight` , `cylinders_4` , **and** `weight x cylinders_4` coefficients are all statistically significant at an alpha of 0.05. This is an indicator that adding the interaction term was a good choice.

Checking Model Metrics

Our original model had this adjusted R-Squared value:

In [9]:
```
no_interaction_results.rsquared_adj
```

Out[9]:
```
0.7213409417018843
```

And our model with the interaction term had this adjusted R-Squared value:

In [10]:
```
interaction_results.rsquared_adj
```

Out[10]:
```
0.733924573658242
```

Once again it appears that adding the interaction term improved the model!

Interpreting a Numeric x Categorical Interaction Term

Because the `const`, `cylinders_4`, `weight`, and `weight x cylinders_4` coefficients are all statistically significant, we can also interpret their meanings in the context of the model.

Intercept Coefficients

In [11]:
```
interaction_results.params["const"]
```

Out[11]:
```
18.243772819630173
```

The value shown above corresponds with β_0 in the formula shown previously. This means that ***for the reference category number of cylinders (3 in this case) and average vehicle weight, we expect to see a fuel economy of about 18 miles per gallon***.

In [12]:
```
interaction_results.params["cylinders_4"]
```

Out[12]:
```
5.082206606031144
```

This value corresponds with $\beta_\text{(4 cylinders)}$. One way we could interpret it is that ***having 4 cylinders (as opposed to the reference category) is associated with an increase of about 5 mpg for a vehicle with average weight***. We also could add β_0 with $\beta_\text{(4 cylinders)}$:

In [13]:
```
interaction_results.params["const"] + interaction_results.params["cylinders_4"]
```

Out[13]:
```
23.325979425661316
```

This means that ***for vehicles with 4 cylinders and average vehicle weight, we expect to see a fuel economy of about 23 miles per gallon***.

Multiplier Coefficients

In [14]:
```
interaction_results.params["weight"]
```

Out[14]:
```
-0.003982542168115807
```

The value shown above corresponds with β_1 in the formula shown previously. This means that ***for any number of cylinders other than 4*** (i.e. 3, 5, 6, or 8 cylinders) ***there is an associated decrease of about 0.004 miles per gallon for each additional pound of vehicle weight***.

In [15]:
```
interaction_results.params["weight x cylinders_4"]
```

Out[15]:
```
-0.0048771958178466385
```

This means that ***for vehicles with 4 cylinders, there is an additional decrease of about 0.005 in miles per gallon for each additional pound of vehicle weight, above and beyond the decrease represented by β_1***.

We can also add the interaction coefficient to the other coefficient (`weight`):

In [16]:
```
interaction_results.params["weight x cylinders_4"] + interaction_results.params["weight"]
```

Out[16]:
```
-0.008859737985962447
```

This means that ***for vehicles with 4 cylinders, there is an associated decrease of about 0.009 miles per gallon for each additional pound of vehicle weight***.

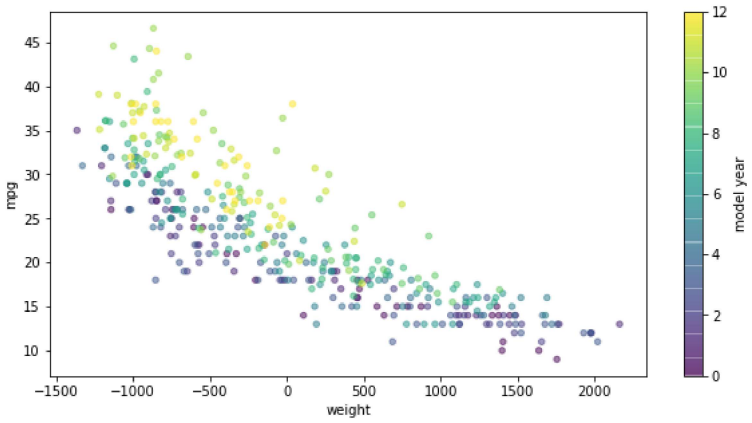
Note that the larger decrease (0.009 compared to 0.004) corresponds to the steeper downward slope shown in the graph.

Interactions with Numeric Features

It is also possible to create an interaction term between two numeric features rather than one numeric feature (e.g. `weight`) and one categorical feature (e.g. `cylinders_4`).

Let's take a look at `weight` vs. `model year`:

In [17]:
```
fig, ax = plt.subplots(figsize=(10,5))
data.plot.scatter(x="weight", y="mpg", c="model year", cmap="viridis", alpha=0.5, ax=ax);
```



Just for the purposes of EDA, we could try binning the years to encode them as categorical variables, then plotting their fit lines.

In [18]:
```
X_many_interactions = data[["weight", "model year"]].copy()
X_many_interactions = pd.get_dummies(X_many_interactions, columns=["model year"], drop_first=True)

# skip first year value because this was dropped in one-hot encoding
years = data["model year"].unique()[1:]

for year in years:
    X_many_interactions[f"weight x model year_{year}"] = X_many_interactions["weight"] * \
                                                         X_many_interactions[f"model year_{year}"]

X_many_interactions
```

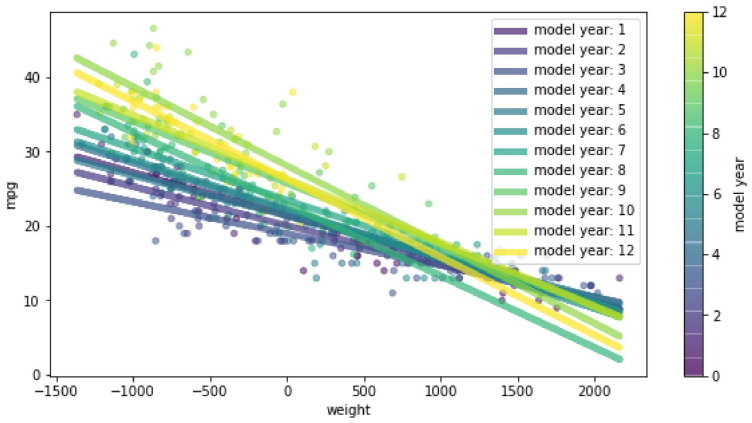
Out[18]:

	weight	model year_1	model year_2	model year_3	model year_4	model year_5	model year_6	model year_7	model year_8	model year_9	...	weight x model year_3	weight x model year_4	weight x model year_5	weight x model year_6	weight x model year_7	weight x model year_8	weight x model year_9	weight x model year_10
0	526.415816	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	715.415816	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	458.415816	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	455.415816	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	471.415816	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
387	-187.584184	0	0	0	0	0	0	0	0	0	...	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0
388	-847.584184	0	0	0	0	0	0	0	0	0	...	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0
389	-682.584184	0	0	0	0	0	0	0	0	0	...	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0
390	-352.584184	0	0	0	0	0	0	0	0	0	...	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0
391	-257.584184	0	0	0	0	0	0	0	0	0	...	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0

392 rows × 25 columns

In [19]:
```
# Build model and get results
many_interactions_model = sm.OLS(y, sm.add_constant(X_many_interactions))
many_interactions_results = many_interactions_model.fit()
```

In [20]:
```
# Plot many fit lines
# (Remember, a model year of 1 means 1971)
plot_fit_lines(data, many_interactions_results, "model year", years, years, cmap="viridis")
```



Looking at the plot above, we can see that there is a trend of steeper slopes for later model years. So rather than treating each of these model years as a separate category, let's treat `model year` as a numeric variable and create a single interaction term `weight x model year`:

In [21]:
```
# Prepare data with a single interaction term of weight x model year
X_numeric_interaction = data[["weight", "model year"]].copy()
X_numeric_interaction["weight x model_year"] = X_numeric_interaction["weight"] * X_numeric_interaction["model year"]
X_numeric_interaction
```

Out[21]:

	weight	model year	weight x model_year
0	526.415816	0	0.000000
1	715.415816	0	0.000000
2	458.415816	0	0.000000
3	455.415816	0	0.000000
4	471.415816	0	0.000000
...	...	...	...
387	-187.584184	12	-2251.010204
388	-847.584184	12	-10171.010204
389	-682.584184	12	-8191.010204
390	-352.584184	12	-4231.010204
391	-257.584184	12	-3091.010204

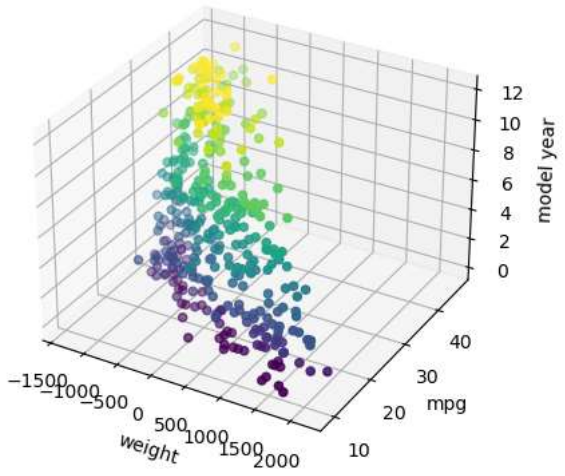
392 rows × 3 columns

In [22]:
```
# Build model and get results
numeric_interaction_model = sm.OLS(y, sm.add_constant(X_numeric_interaction))
numeric_interaction_results = numeric_interaction_model.fit()
```

To visualize how this model works, let's look at a 3D scatter plot, with the same x-axis (`weight`) and y-axis (`mpg`) as before, this time with a z-axis showing the `model year`. This is an interactive plot, so you can click and drag to see the data from different angles!

In [23]:
```
%matplotlib nbagg
fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# Set up 3D scatter plot
ax.scatter(
    data["weight"],      # x axis
    data["mpg"],         # y axis
    data["model year"],  # z axis
    c=data["model year"],
    cmap="viridis"
)
ax.set_xlabel("weight")
ax.set_ylabel("mpg")
ax.set_zlabel("model year");
```

```
<IPython.core.display.Javascript object>
```

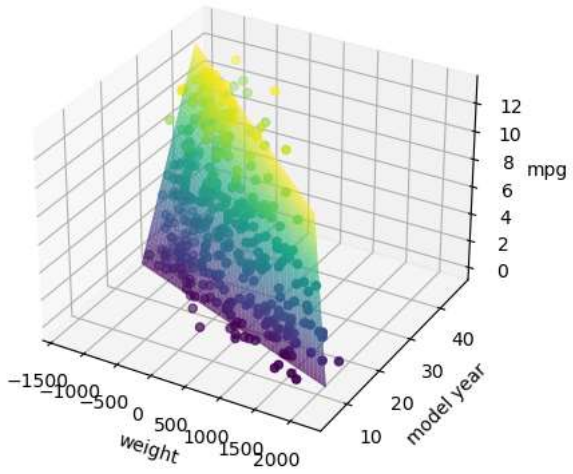


Now we can generate a plane of weights, years, and predictions, then plot that plane on the same axes as the scatter plot.

In [24]:
```
import numpy as np

# Find 50 evenly-spaced points in the ranges of weight and model year and create meshes
weights = np.linspace(data["weight"].min(), data["weight"].max() + 1, 50)
years = np.linspace(data["model year"].min(), data["model year"].max() + 1, 50)
weights, years = np.meshgrid(weights, years)

# Extract relevant values from modeling results
constant = numeric_interaction_results.params["const"]
weight_coef = numeric_interaction_results.params["weight"]
year_coef = numeric_interaction_results.params["model year"]
interaction_coef = numeric_interaction_results.params["weight x model_year"]

# Make a hyperplane of predictions using the values extracted from the model
predictions = (
    # beta 0
    constant + \
    # beta 1 times weight
    weight_coef * weights + \
    # beta 2 times model year
    year_coef * years + \
    # beta 3 times weight times model year
    interaction_coef * weights * years
)
```

In [25]:
```
fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# Set up 3D scatter plot
ax.scatter(
    data["weight"],      # x axis
    data["mpg"],         # y axis
    data["model year"],  # z axis
    c=data["model year"],
    cmap="viridis"
)
ax.set_xlabel("weight")
ax.set_ylabel("model year")
ax.set_zlabel("mpg")

# Plot surface of predictions on the same axes
ax.plot_surface(
    weights,        # x
    predictions,    # y
    years,          # z
    cmap="viridis", # color is drawn from z axis value
    alpha=0.7
);
```

```
<IPython.core.display.Javascript object>
```



The formula being plotted here is:

$$ \large{ \text{mpg} = \beta_0 + \beta_1 \times \text{weight} + \beta_2 \times \text{model year} + \beta_3 \times \text{weight} \times \text{model year} } $$

Note that the hyperplane is **curved** rather than a flat plane because of the interaction term.

If we just had the formula $\text{mpg} = \beta_0 + \beta_1 \times \text{weight} + \beta_2 \times \text{model year}$ (without the interaction term) then it would have been a flat plane made up of just a straight line with a slope of β_1 along the `weight` axis and a slope of β_2 along the `model year` axis.

Interpreting a Numeric x Numeric Interaction Term

When both of the features used in the interaction are numeric, the interpretation is a little more tricky than with one numeric and one categorical feature. We can't just say "if `category` , there is a `coefficient` change in the target".

Instead, this kind of model is saying that the change in the target associated with the change in a given predictor is dependent on the value of another predictor!

Let's dive into some interpretations for the model shown above.

In [26]:
```
numeric_interaction_results.pvalues
```

Out[26]:
```
const                  9.257577e-197
weight                  1.673750e-33
model year              1.136337e-37
weight x model_year     8.015486e-14
dtype: float64
```

In [27]:
```
numeric_interaction_results.params
```

Out[27]:
```
const                 18.956272
weight                -0.004509
model year             0.676944
weight x model_year   -0.000458
dtype: float64
```

Intercept Coefficient

Compared to a model with categorical features, the interpretation of the intercept is fairly straightforward. The value of the `const` coefficient is about 19, which means that ***for a vehicle with average weight and a model year of 1970, we expect to see a fuel economy of about 19***.

(Remember that we subtracted the mean from `weight` and subtracted 70 from `model_year` in order to make intercepts like this more interpretable.)

One-At-A-Time Interpretations

One way that we can interpret just the `weight` coefficient or just the `model_year` coefficient is by setting the value of the *other* predictor to 0. That way you can ignore the `weight x model_year` beta altogether.

Thus we can say:

1. ***For a vehicle with a model year of 1970, there is a decrease of about 0.005 in mpg for each additional pound of vehicle weight.***
2. ***For a vehicle with average weight, there is an increase of about 0.7 mpg for each year newer of a model year.***

Combined Interpretations

And what about a vehicle that does not have a model year of 1970, or is not of average weight? For that, we need to incorporate the `weight x model year` value, which is about 0.0005.

1. ***For each additional pound of vehicle weight, there is a decrease of about 0.005 + (0.0005 x model year) in mpg.***
2. ***For each year newer of a vehicle model, there is an increase of about 0.7 - (0.0005 x weight) in mpg.***

It will depend on the context whether the one-at-a-time or combined interpretation is more useful. There is a trade-off between how easy the explanation is to grasp vs. how much of the model it explains.

Level Up: Interactions with Scikit-Learn

In the examples above we used `pandas` to create the interaction terms in our dataframes. This is simple enough if you want to have relatively few interaction terms, but what if you wanted to add every single possible pairwise combination of variables? `sklearn` allows you to make lots of interaction terms at once with the `PolynomialFeatures` class and `interaction_only` parameter (documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html)).

In [28]:
```
# Prep subset of data for creating interactions
sklearn_data = data.drop("origin", axis=1).select_dtypes("number")
sklearn_data.rename(columns={"model year": "model_year"}, inplace=True)
sklearn_data
```

Out[28]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year
0	18.0	8	307.0	130	526.415816	12.0	0
1	15.0	8	350.0	165	715.415816	11.5	0
2	18.0	8	318.0	150	458.415816	11.0	0
3	16.0	8	304.0	150	455.415816	12.0	0
4	17.0	8	302.0	140	471.415816	10.5	0
...	...	...	...	...	...	...	...
387	27.0	4	140.0	86	-187.584184	15.6	12
388	44.0	4	97.0	52	-847.584184	24.6	12
389	32.0	4	135.0	84	-682.584184	11.6	12
390	28.0	4	120.0	79	-352.584184	18.6	12
391	31.0	4	119.0	82	-257.584184	19.4	12

392 rows × 7 columns

In [29]:
```
# Separate X and y
y = sklearn_data["mpg"]
X = sklearn_data.drop("mpg", axis=1)
```

In [30]:
```
from sklearn.preprocessing import PolynomialFeatures

# Fit a PolynomialFeatures transformer on X
poly = PolynomialFeatures(interaction_only=True, include_bias=False)
poly.fit(X)

# Create transformed version of X that has all of the original features plus
# interactions between all features
new_col_names = pd.Series(poly.get_feature_names(X.columns)).str.replace(" ", " x ")
X_interactions = pd.DataFrame(poly.transform(X), columns=new_col_names)
X_interactions
```

Out[30]:

	cylinders	displacement	horsepower	weight	acceleration	model_year	cylinders x displacement	cylinders x horsepower	cylinders x weight	cylinders x acceleration	...	displacement x horsepower	disp
0	8.0	307.0	130.0	526.415816	12.0	0.0	2456.0	1040.0	4211.326531	96.0	...	39910.0	16160
1	8.0	350.0	165.0	715.415816	11.5	0.0	2800.0	1320.0	5723.326531	92.0	...	57750.0	25039
2	8.0	318.0	150.0	458.415816	11.0	0.0	2544.0	1200.0	3667.326531	88.0	...	47700.0	14577
3	8.0	304.0	150.0	455.415816	12.0	0.0	2432.0	1200.0	3643.326531	96.0	...	45600.0	13844
4	8.0	302.0	140.0	471.415816	10.5	0.0	2416.0	1120.0	3771.326531	84.0	...	42280.0	14236
...	...	...	...	...	...	...	...	...	...	...	...	...	
387	4.0	140.0	86.0	-187.584184	15.6	12.0	560.0	344.0	-750.336735	62.4	...	12040.0	-2626
388	4.0	97.0	52.0	-847.584184	24.6	12.0	388.0	208.0	-3390.336735	98.4	...	5044.0	-8221
389	4.0	135.0	84.0	-682.584184	11.6	12.0	540.0	336.0	-2730.336735	46.4	...	11340.0	-9214
390	4.0	120.0	79.0	-352.584184	18.6	12.0	480.0	316.0	-1410.336735	74.4	...	9480.0	-4231
391	4.0	119.0	82.0	-257.584184	19.4	12.0	476.0	328.0	-1030.336735	77.6	...	9758.0	-3065

392 rows × 21 columns

As you might imagine, creating this many interaction terms will make the model much more difficult to interpret! Using scikit-learn like this typically only makes sense in a predictive analysis rather than an inferential one.

Additional Resources

- You can use the Python library `seaborn` to visualize interactions as well. Have a look here (https://blog.insightdatascience.com/data-visualization-in-python-advanced-functionality-in-seaborn-20d217f1a9a6) for more information.
- This resource (http://www.medicine.mcgill.ca/epidemiology/joseph/courses/EPIB-621/interaction.pdf) walks over multiple examples of regressions with interaction terms. Even though the code is in R, it might give you some additional insights into how interactions work.
- This resource (http://joelcarlson.github.io/2016/05/10/Exploring-Interactions/) also uses R but has several visualizations of interactions with the auto MPG dataset specifically!

Summary

Great! You now know how to describe interactions, how to include them in your model and how to interpret them. You'll practice what you learned here in the next lab.