DATA SOCIETY®

Regression - Day 2

“One should look for what is and not what he thinks should be.”
-Albert Einstein.

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	
Summarize how to implement multiple regression	
Clean the new multiple regression dataset and use EDA to understand our data	
Define best practices of model building and what the training / test sets are and when to use them	
Implement and evaluate a multiple linear regression model on the training set	
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Import packages

- Let's import the libraries we will be using today

```
import os
import pandas as pd
import numpy as np
import pickle

import statsmodels.api as sm
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from sklearn.feature_selection import VarianceThreshold

# New today - we will introduce it when we use it.
from sklearn.model_selection import train_test_split
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/Desktop/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/Desktop/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\af-werx"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"
```

Working directory

- Set working directory to the `data_dir` variable we set
- We do this using the `os.chdir` function, change directory
- We can then check the working directory using `.getcwd()`
- For complete documentation of the `os` package, ***click here***

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/af-werx/data
```

What is multiple linear regression?

- A statistical technique that uses multiple explanatory variables to predict the outcome of a response variable.
- The goal of multiple linear regression is to model the linear relationship between the independent variables and dependent variable.

Difference between MLR and LR

- Simple linear regression has only one x and one y variable.
- Multiple linear regression has one y and two or more x variables.

Datasets for multiple regression

- **We will be using two datasets today to showcase how using multiple linear regression can be used**

- **One dataset in class, to learn the concepts**
  - Fast food data

- **One dataset for our in-class exercises**
  - Chemical manufacturing process

Fast food dataset

- Today, we will use `fast_food_data.csv` for multiple regression
- Our dependent variable will be `Calories` and we will have multiple independent variables (predictors)
- Here is the data dictionary of the most important variables:

Field	Definition
`Fast Food Restaurant`	Name of the restaurant offering the product
`Item`	Name of the product
`Type`	Type of the product
`Serving Size (g)`	Serving size in grams
`Calories`	Calories of the product
`Total Fat (g)`	Fat of the product in grams
`Sodium (mg)`	Sodium of the product in milligrams
`Carbs (g)`	Carbs of the product in grams
`Sugars (g)`	Sugars of the product in grams
`Protein (g)`	Protein of the product in grams
`Revenue 2016 (billion dollars)`	Revenue of the product in billion dollars

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	
Clean the new multiple regression dataset and use EDA to understand our data	
Define best practices of model building and what the training / test sets are and when to use them	
Implement and evaluate a multiple linear regression model on the training set	
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Multiple regression

- The concept of multiple linear regression is really not that different from simple linear regression
- Instead of a single predictor variable, we have at least two predictor variables
- We still have one target variable that is continuous

Multiple regression: methodology

Stays the same as single regression

- We still follow the slope intercept model from single linear regression: $y = mx + b$
- We still have one coefficient for intercept (b)
- Summary function of `lm` measures the same thing
- We still follow LINE for assumptions of linear regression

Introduced for multiple regression

- Instead of finding 1 coefficient for slope, we will find multiple correlations
- The number of coefficients for slope depends directly on how many predictors we use
- One additional assumption for multiple regression is that the predictors are not correlated

Using fast_food_data dataset for regression

- Your manager at a public health organization wants you to create a basic dataframe with calories, total fat, and sugars in order to analyze the nutritional value of fast food from various restaurants
- Let us analyze how `Calories`, the number of calories in a product, is affected by `Total Fat (g)` and `Sugars (g)`
- Therefore, we set the model up like this:
  - Dependent variable: `Calories`
  - Predictors: `Total Fat (g)` and `Sugars (g)`

Read in the data and subset

- We read in the `fast_food_data.csv` dataset and assign it to the variable `fast_food`

```
# This dataset is of type dataframe. Let's assign this dataset to a variable, so that we can manipulate it freely.
fast_food = pd.read_csv('fast_food_data.csv')
```

- We subset the data so that it only contains `Calories`, `Total Fat (g)` and `Sugars (g)`

```
fast_food = fast_food.loc[:,["Calories", "Total Fat (g)", "Sugars (g)"]]
fast_food.head()
```

```
   Calories  Total Fat (g)  Sugars (g)
0       240            8.0         6.0
1       290           11.0         7.0
2       530           27.0         9.0
3       520           26.0        10.0
4       720           40.0        14.0
```

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	
Define best practices of model building and what the training / test sets are and when to use them	
Implement and evaluate a multiple linear regression model on the training set	
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Summary statistics

- Summary statistics help us understand our data better
- Summary statistics we will review are:

Summary statistic	Definition
Count	the count of non-null / NA observations
Mean	the central value or average of the dataset / vector
Standard deviation	the standard deviation of the observations
Min	the minimum number in the vector
1st quartile (25%)	the median of the lower half of the dataset / vector
3rd quartile (75%)	the median of the higher half of the dataset / vector
Max	the maximum number in the vector

Summary statistics: multiple regression

- Numerical summary of all three variables together

```
# Let's look at the summary statistics.
print(fast_food.describe())
```

```
          Calories  Total Fat (g)  Sugars (g)
count   126.000000     126.000000  126.000000
mean    532.492063      28.544444   13.280952
std     250.844294      18.240963   21.022216
min     130.000000       3.500000    0.000000
25%     330.000000      14.175000    3.000000
50%     515.000000      22.500000    7.000000
75%     670.000000      39.500000   11.000000
max    1240.000000      87.000000   93.000000
```

Summary statistics: covariance and correlation

Let's calculate the covariance and correlation of the data and save the correlation values

```
print(fast_food.cov())
```

```
                  Calories  Total Fat (g)
Sugars (g)
Calories       62922.859937    4304.584356
1308.093448
Total Fat (g)   4304.584356     332.732729
5.880293
Sugars (g)      1308.093448       5.880293
441.933554
```

```
print(fast_food.corr())
```

```
                Calories  Total Fat (g)  Sugars
(g)
Calories        1.000000       0.940761
0.248060
Total Fat (g)   0.940761       1.000000
0.015335
Sugars (g)      0.248060       0.015335
1.000000
```

```
fast_food_cor = fast_food.corr()
```

Data cleaning: review preparing for regression

Let's follow the same process for simple linear regression in cleaning our data:

- **NAs**
  - They can produce non-numeric results (e.g. `NaNs`)
  - Some functions may fail if `NAs` are present

- **Zero** and **Near-zero variance (NZV)** variables
  - Zero variance **occurs when a variable has a single unique value**
  - Near-zero variance happens when there are **very few instances of different values** and one dominant unique value
  - They are **not predictive**
  - They can cause **trouble** with numerical precision and when standardizing data

Data cleaning: NAs

- First, we check how many NAs there are in each column

```
# Check how many values are null in the Calories column.
print(fast_food.Calories.isnull().sum())
```

```
0
```

```
# Check how many values are null in the Total Fat (g) column.
print(fast_food["Total Fat (g)"].isnull().sum())
```

```
0
```

```
# Check how many values are null in the Sugars (g) column.
print(fast_food["Sugars (g)"].isnull().sum())
```

```
0
```

- **We see that none of the rows have NA values**
- We can move on to check for non-zero variance

Data cleaning: near zero variance

- We are concerned with variables that have zero variance because they will not be helpful in prediction
- This is especially useful when you have high-dimension datasets
- We once again use the function `VarianceThreshold` from `sklearn.feature_selection`
- Your manager would like for you to create a few simple visualizations depicting the calories, fats and sugars of fast food products to get a basic overview of the nutritional value of these foods

```
# Using sklearn, let's look for low variance within the columns.
# First, instantiate the function.
selector = VarianceThreshold()
# Then, name the cleaned dataset fast_food_clean.
fast_food_clean = selector.fit_transform(fast_food)
# Let's see if the dimensions changed.
print(fast_food_clean.shape)
```

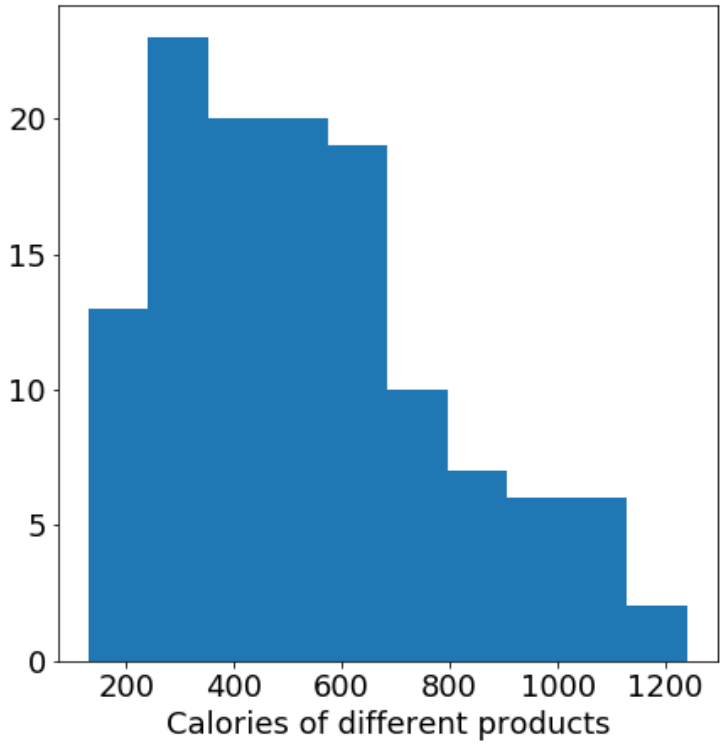
```
(126, 3)
```

- We see that the columns remained in the dataset
- Therefore we can confirm that we do not have an issue with either variable having zero variance

EDA: histogram `Calories`

- Let us start EDA by looking at a histogram of `Calories`
- We will use `plt` to now build a histogram
- **What observations can we make about `Calories`?**

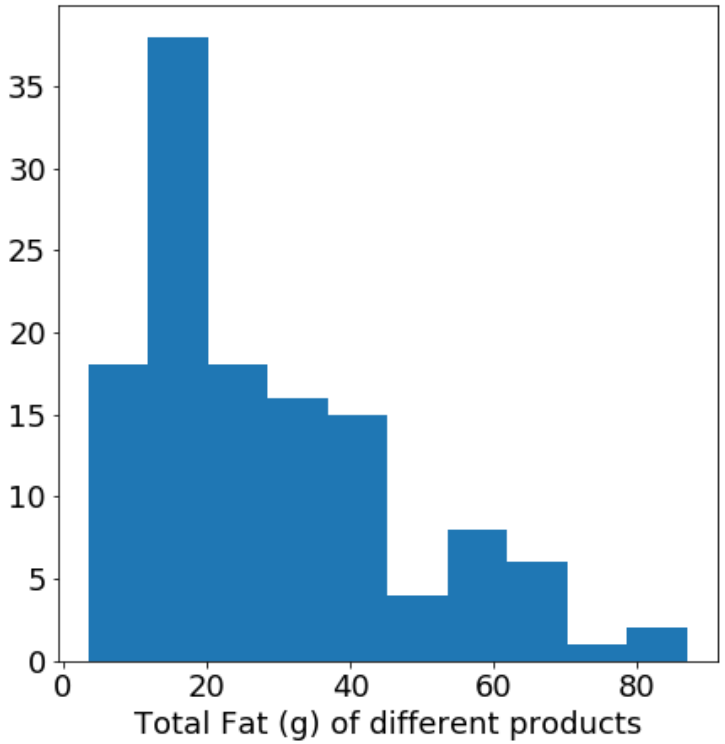
```
plt.hist(fast_food['Calories'],  bins
= 10)
plt.xlabel('Calories of different
products')
```



EDA: histogram `Total Fat (g)`

- Next, let us look at a histogram of `Total Fat (g)`
- **What observations can we make about `Total Fat (g)`?**

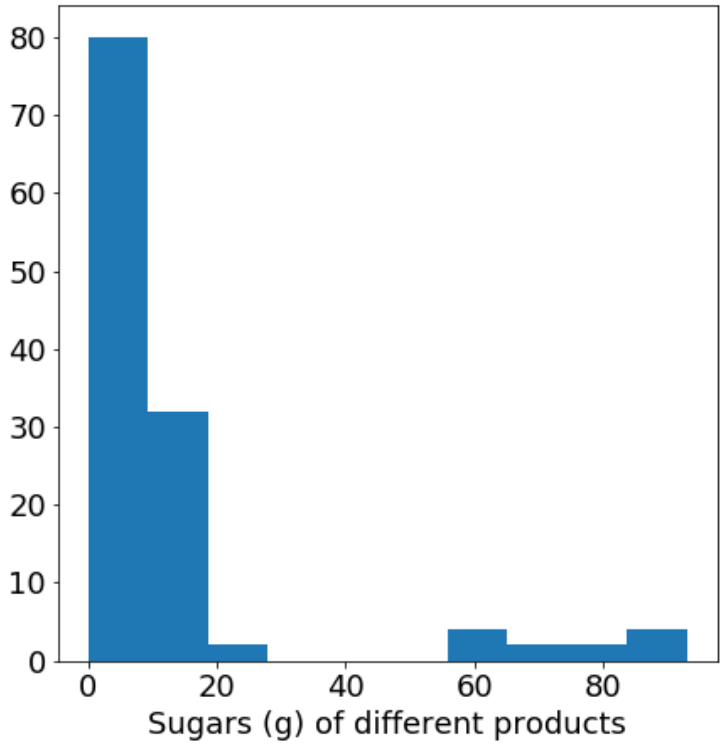
```
plt.hist(fast_food['Total Fat (g)'],
bins = 10)
plt.xlabel('Total Fat (g) of different
products')
```



EDA: histogram `Sugars (g)`

- Lastly, let us look at a histogram of `Sugars (g)`
- **What observations can we make about `Sugars (g)`?**

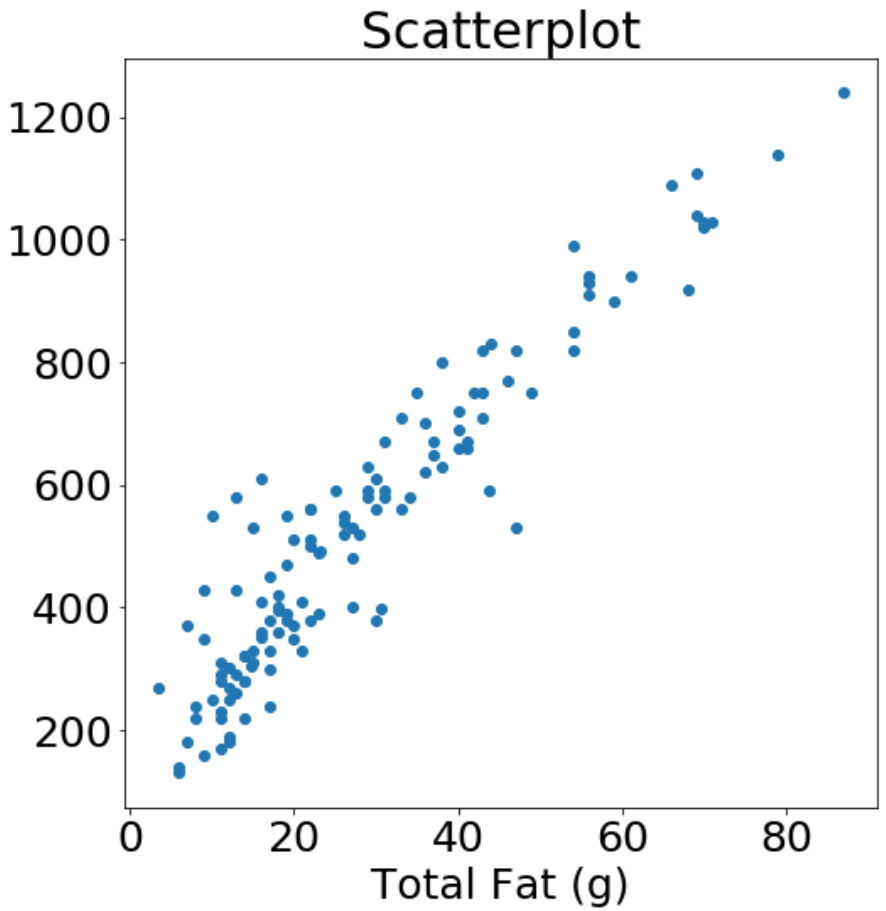
```
plt.hist(fast_food['Sugars (g)'], bins = 10)
plt.xlabel('Sugars (g) of different products')
```



EDA: scatterplot `Calories` and `Total Fat (g)`

- Let's use EDA to understand the relationship between `Calories` and `Total Fat (g)`
- We do that by building a scatterplot of `Calories` and `Total Fat (g)`

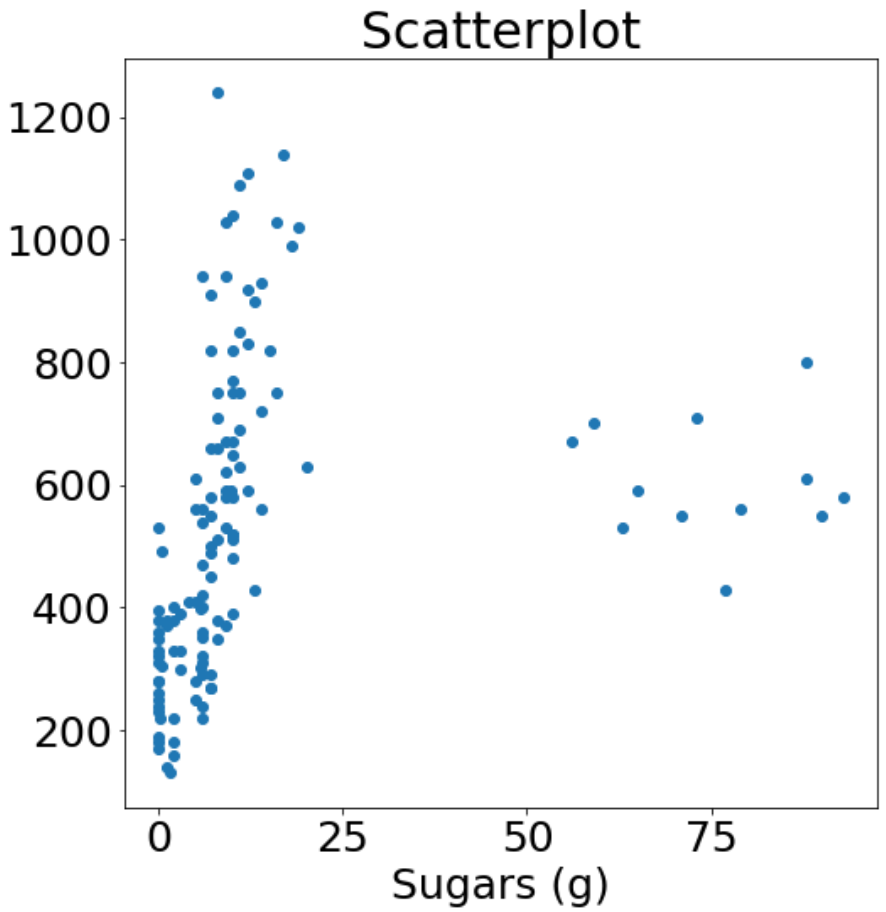
```
# Make scatterplot.
plt.scatter(fast_food['Total Fat (g)'],
            fast_food['Calories'])
plt.title("Scatterplot")
plt.xlabel("Total Fat (g)")
plt.ylabel("Calories")
plt.show()
```



EDA: scatterplot `Calories` and `Sugars (g)`

- Let's use EDA to understand the relationship between `Calories` and `Sugars (g)`
- We do that by building a scatterplot of `Calories` and `Sugars (g)`

```
# Make scatterplot.
plt.scatter(fast_food['Sugars (g)'],
            fast_food['Calories'])
plt.title("Scatterplot")
plt.xlabel("Sugars (g)")
plt.ylabel("Calories")
plt.show()
```



Modeling with our dataset

- Let's get our data ready for the next step - modeling
- We will split our `fast_food` dataset into two variables
  - `X`: the predictors, `Total Fat (g)` and `Sugars (g)`
  - `y`: the target variable `Calories`

```
# Three variables for multiple regression.

# Set X to `Total Fat (g)` and `Sugars (g)`
X = pd.DataFrame(fast_food.loc[:, ['Total Fat (g)', 'Sugars (g)']])
# Add a constant.
X = sm.add_constant(X)
# Rename columns for clarity when interpreting results.
X = X.rename(columns = {'const':'constant', 0:'Fat', 1:'Sugar'})

# y - `Calories`
y = pd.DataFrame(fast_food.loc[:,'Calories'])
# Rename columns for clarity when interpreting results.
y = y.rename(columns = {0:'Calories'})
```

Knowledge Check 1

Exercise 1

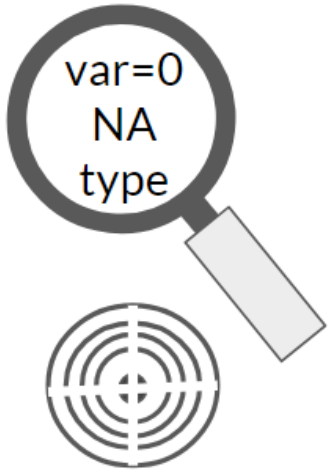
Module completion checklist

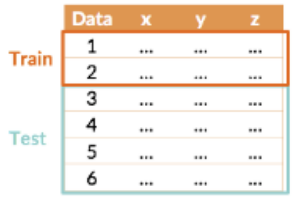
Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	✔
Define best practices of model building and what the training / test sets are and when to use them	
Implement and evaluate a multiple linear regression model on the training set	
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Modeling best practices at all stages

Before modeling

EDA





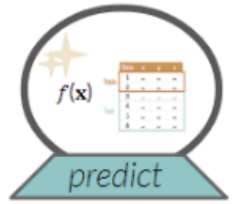
While modeling

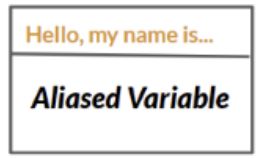
$f(\mathbf{x})$

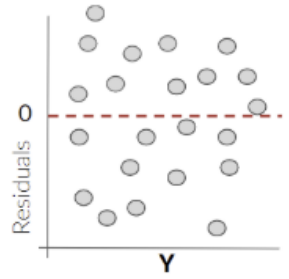
p-val t-test f-statistic

R^2 & adjusted R^2

After modeling



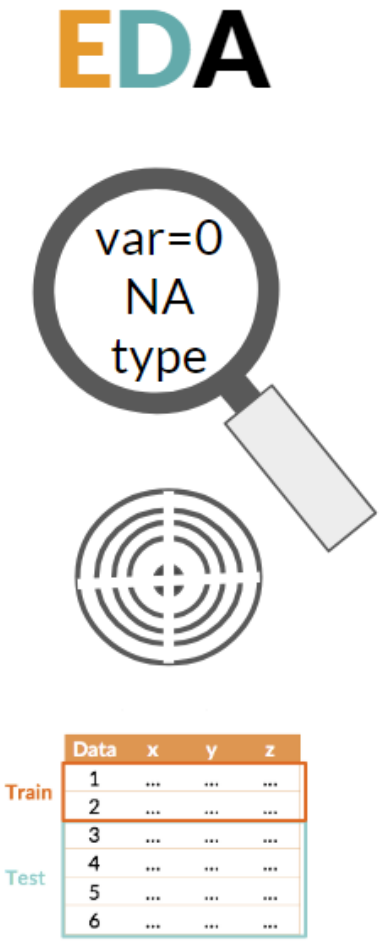




Before modeling

Best practices:

- Perform thorough EDA
- Get rid of variables that have **no or little variance**
- Clean up the dataset and **remove NAs**
- Check that variables are the desired **type**
- Define & refine your target variable
- Split data into a **training** set and a **test** set



While modeling

Best practices:

- Use the **t-statistic** and **p-values** with different predictor variables to see if variables should be kept in the model
- Use **R-squared** and **adjusted R-squared** to compare different versions of the model
- Use the **F-statistic** and the **p-value** to see if model performance improves as different variable combinations are tried

$$f(\mathbf{x})$$

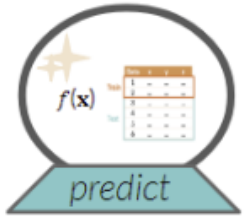
p-val t-test f-statistic

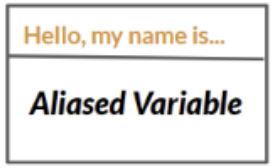
R^2 & adjusted R^2

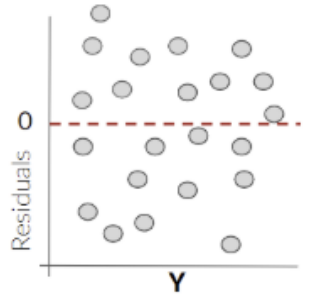
After modeling

Best practices:

- Apply model to test data to assess performance
- Check residuals to see if transformed versions of variables need to be added
- Make sure there aren't any points causing too much trouble in your model (Cook's distance)







Model fit: generalizable

- **Generalizability**: the performance and adaptability of a model when applied to new conditions while maintaining the same basic set of explanatory variables.
- But what if we don't only risk overfitting from adding too many variables?
- What if the overfitting comes from our **DATA** ?

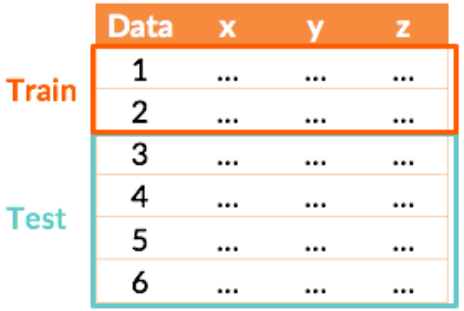
All data is a sample

- It's important to remember that whatever dataset we have, it is only a sample of all the possible outcomes
  - More tests might have yielded more results
  - The participants might be unique
  - There might have been special circumstances
- Does the model perform well outside of the data we used to create it?
- How would we test that?

Splitting data into train and test

- We ran our simple linear regression model on 100% of the data
- That is not a best practice
- When you are using a **supervised learning method** to build a *generalizable* model:

You should <u>always remember</u> to split your data into a **training** and a **test** set



Understanding training and test data

Training

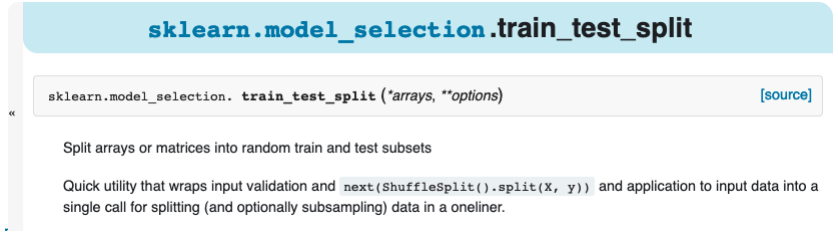
- This is the data that you **train your model on**
- Usually about **70% of your dataset**
- Use a larger portion of the data to train so that the model gets a large enough sample of the population
- If there is not a large population on the whole, cross-validation techniques can be implemented

Test

- This is the data that you **test your model on**
- Usually about **30% of your dataset**
- Use a smaller portion to test your trained model on
- If cross-validation is implemented, small test sets will be held out multiple times

Train and test: sklearn train_test_split

- We will use `train_test_split` from `sklearn.model_selection`



sklearn.model_selection.train_test_split

sklearn.model_selection. **train_test_split** (**arrays*, ***options*) [source]

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

- `sklearn` is the most used library in Python for machine learning, we will use it in most of the machine learning related classes coming up
- However, `statsmodel` gives us much more thorough of an output for regression

Parameters:

arrays : *sequence of indexables with same length / shape[0]

Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size : *float, int or None, optional (default=0.25)*

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. By default, the value is set to 0.25. The default will change in version 0.21. It will remain 0.25 only if `train_size` is unspecified, otherwise it will complement the specified `train_size`.

train_size : *float, int, or None, (default=None)*

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state : *int, RandomState instance or None, optional (default=None)*

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

shuffle : *boolean, optional (default=True)*

Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify : *array-like or None (default=None)*

If not None, data is split in a stratified fashion, using this as the class labels.

Apply to fast_food regression

- Now we are going to use `sklearn.model_selection.train_test_split` to split the `fast_food` dataset
- Always remember to use `np.random.seed(1)` when you create the training partition

```
# Set the seed.
np.random.seed(1)

# Create the training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)

# Check to see if the datasets split correctly.
print(X_train.shape, y_train.shape)
```

```
(88, 3) (88, 1)
```

```
print(X_test.shape, y_test.shape)
```

```
(38, 3) (38, 1)
```

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	✔
Define best practices of model building and what the training / test sets are and when to use them	✔
Implement and evaluate a multiple linear regression model on the training set	
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Multiple linear regression on fast_food

- We will run a multiple linear regression on our training data using `sm.OLS`
- In order to create a generalizable and great performing model, your manager would like you split the data into a training and test set

```
# Build a linear model on training data.
model_m = sm.OLS(y_train, X_train).fit()
```

```
print(model_m.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               Calories   R-squared:                       0.949
Model:                            OLS   Adj. R-squared:                  0.948
Method:                 Least Squares   F-statistic:                     790.2
Date:                Mon, 22 Jul 2019   Prob (F-statistic):           1.22e-55
Time:                        16:43:45   Log-Likelihood:                -485.05
No. Observations:                  88   AIC:                             976.1
Df Residuals:                      85   BIC:                             983.5
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
constant       119.6427     12.124      9.868      0.000      95.536     143.749
Total Fat (g)   13.2056      0.348     37.972      0.000      12.514      13.897
Sugars (g)       2.6217      0.355      7.385      0.000       1.916       3.327
==============================================================================
Omnibus:                        1.534   Durbin-Watson:                   1.978
Prob(Omnibus):                  0.464   Jarque-Bera (JB):                1.326
Skew:                          -0.138   Prob(JB):                        0.515
Kurtosis:                       2.465   Cond. No.                         67.7
==============================================================================
```

Evaluate: coefficients

- Let's look at our coefficients and evaluate them based on:
  - `std error`
  - `t value/p value`

	coef	std err	t	P>|t|	[0.025	0.975]
constant	119.6427	12.124	9.868	0.000	95.536	143.749
Total Fat (g)	13.2056	0.348	37.972	0.000	12.514	13.897
Sugars (g)	2.6217	0.355	7.385	0.000	1.916	3.327

- `std error`
  - Ideally, we want a lower number relative to its coefficients
  - **All three std error values are lower than their respective coefficients**
- `t-value / p-value`
  - A small p-value indicates it is unlikely the relationship between x and y is based on chance
  - **The p-values indicate that the predictors do have a relationship to the target**

Evaluate: model

OLS Regression Results

Dep. Variable:	Calories	R-squared:	0.949
Model:	OLS	Adj. R-squared:	0.948
Method:	Least Squares	F-statistic:	790.2
Date:	Mon, 22 Jul 2019	Prob (F-statistic):	1.22e-55
Time:	16:43:45	Log-Likelihood:	-485.05
No. Observations:	88	AIC:	976.1
Df Residuals:	85	BIC:	983.5
Df Model:	2		
Covariance Type:	nonrobust		

- **Multiple R-squared**: the R-squared of .949 means that around 94.9% of the variance found in `Calories` can be explained by `Fat` and `Sugar`
- **Adjusted R-squared**: the adjusted R-squared is the exact same as the R-squared, which is not common but could mean that either our predictors are collinear or that the additional predictor is not adding any value to the model
- **F-statistic**: p-value of $1.22e^{-55}$ is **significant**, so we can reject the null hypothesis that this model does not have predictive power

Check LINE

- Don't forget about **LINE**!
- Just in case you did forget:
  - The relationship between the predictor and response variable must be **L**inear
  - The residuals must be **I**ndependent
  - The residuals must be **N**ormally distributed
  - The residuals must have **E**qual variance
- We need to check assumptions to make sure that we should be using a linear model

Check multiple regression model

- Let's go through the three assumptions we validated yesterday and look to see how our multiple regression model does
  - **The relationship between the predictor and response variable must be [L]inear**
  - **The residuals must be [N]ormally distributed**
  - **The residuals must have [E]qual variance**
- And then we will look for outliers in the model and rerun it before we **predict on our test set**

Assumptions: plot

- First, let's find the residuals as outputted from the model
- The residuals are equal to the `actual - predicted` value, which we will also calculate later on when we **predict on our test set**
- Right now, we are just looking at our trained model

```
fitted_m = model_m.fittedvalues
print(fitted_m.head())
```

```
55     1289.501847
65      568.535281
100     243.736271
10      264.904131
85      266.749021
dtype: float64
```

```
residuals_m = model_m.resid
print(residuals_m.head())
```

```
55    -49.501847
65     -8.535281
100   -83.736271
10    -34.904131
85    -46.749021
dtype: float64
```

Assumptions: plot

- We also need a couple more metrics to build the assumption plots:

```
# Get the normalized residuals.
model_m_norm_residuals = model_m.get_influence().resid_studentized_internal
# Get the absolute squared normalized residuals.
model_m_norm_residuals_abs_sqrt = np.sqrt(np.abs(model_m_norm_residuals))
# Get the absolute residuals.
model_m_abs_resid = np.abs(residuals_m)
```

- Finally, we combine `X_train` and `y_train` so we can plot our assumptions

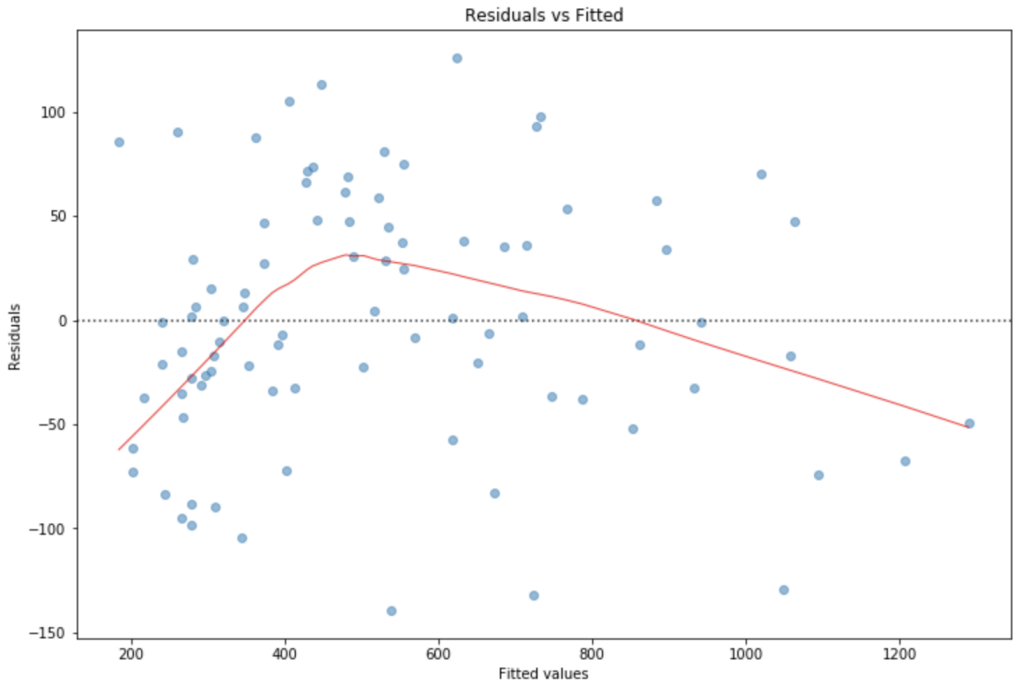
```
# Combine X_train and y_train into one dataframe for plotting.
frames = [X_train,y_train]
training = pd.concat(frames, axis = 1) # axis = 1 allows us to combine by columns
```

Assumption: residuals vs. fitted

The first assumption is:

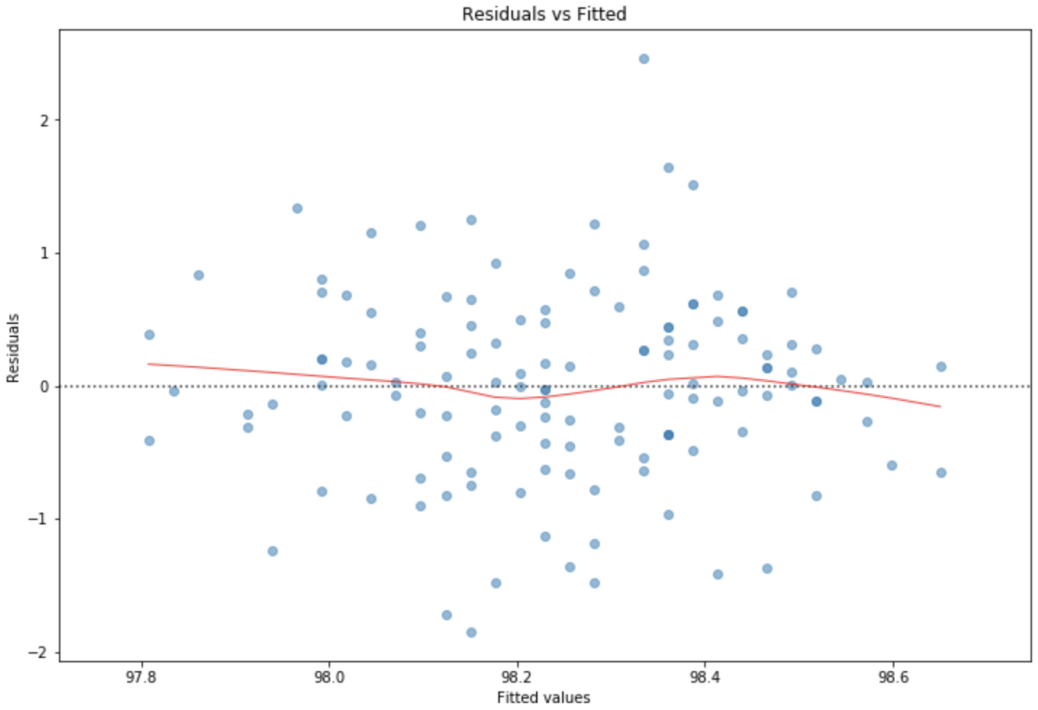
The relationship between the predictor and response variable must be [L]inear

```
import seaborn as sns

# Let's look at assumption 1.
plot_lm_1 = plt.figure(1)
plot_lm_1.set_figheight(8)
plot_lm_1.set_figwidth(12)

plot_lm_1.axes[0] = sns.residplot(fitted_m,
'Calories', data = training,
                          lowess = True,
                          scatter_kws =
{'alpha': 0.5},
                          line_kws = {'color':
'red', 'lw': 1, 'alpha': 0.8})

plot_lm_1.axes[0].set_title('Residuals vs
Fitted')
plot_lm_1.axes[0].set_xlabel('Fitted values')
plot_lm_1.axes[0].set_ylabel('Residuals')
```

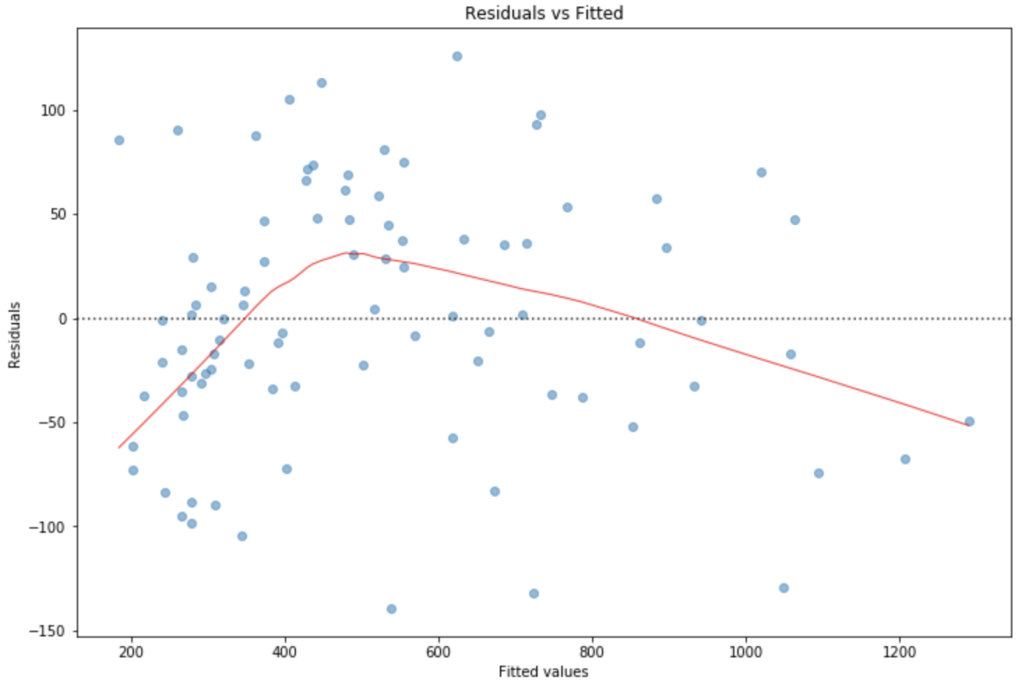


Assumption: linear not satisfied

- **This is what the assumption would look like if it were satisfied**



- **Our multiple linear regression model does not meet the assumption**

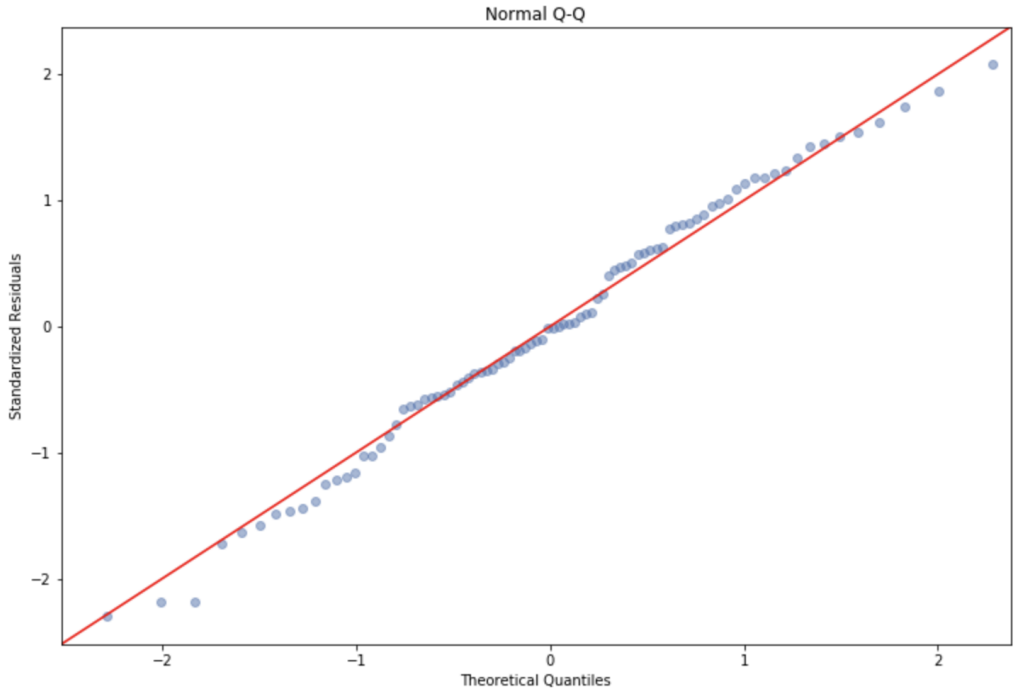


Assumption: normally distributed residuals

The second assumption is:

The residuals must be [N]ormally distributed

```
from statsmodels.graphics.gofplots import
ProbPlot

QQ = ProbPlot(model_m_norm_residuals)
plot_lm_2 = QQ.qqplot(line = '45', alpha = 0.5,
color = '#4C72B0', lw = 1)

plot_lm_2.set_figheight(8)
plot_lm_2.set_figwidth(12)

plot_lm_2.axes[0].set_title('Normal Q-Q')
plot_lm_2.axes[0].set_xlabel('Theoretical
Quantiles')
plot_lm_2.axes[0].set_ylabel('Standardized
Residuals');
```



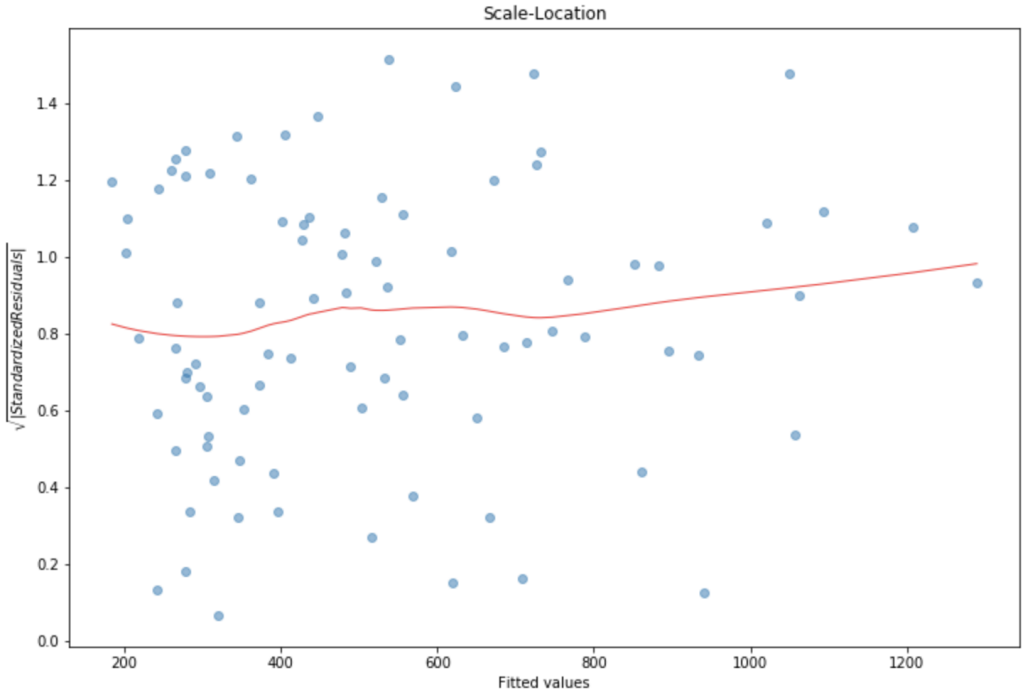
- **Our model does satisfy the assumption**

Assumption: equal residual variance

The third assumption is:

The residuals must have [E]qual variance

```
plot_lm_3 = plt.figure(3)
plot_lm_3.set_figheight(8)
plot_lm_3.set_figwidth(12)

plt.scatter(fitted_m,
model_m_norm_residuals_abs_sqrt, alpha = 0.5)
sns.regplot(fitted_m,
model_m_norm_residuals_abs_sqrt,
            scatter = False,
            ci = False,
            lowess = True,
            line_kws = {'color': 'red', 'lw': 1,
'alpha': 0.8})

plot_lm_3.axes[0].set_title('Scale-Location')
plot_lm_3.axes[0].set_xlabel('Fitted values')
plot_lm_3.axes[0].set_ylabel('$\sqrt{|Standardized
 Residuals|}$')
```



- **Our model does satisfy the assumption**

Knowledge Check 2

Exercise 2

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	✔
Define best practices of model building and what the training / test sets are and when to use them	✔
Implement and evaluate a multiple linear regression model on the training set	✔
Manage influential points and highly correlated variables as necessary and rerun optimized model	
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Influential cases: residuals vs. leverage

- We just evaluated our model against three of the assumptions
- Next, let's look for outliers
- We'll use `statsmodel.outlier_test()` to test for outliers which uses the ***Bonferroni outlier test***
- Your manager would like for you to test for outliers, to see how much they are affecting the potential errors in your model

```
# Identify the influential points ('\n' syntax creates a new
# line in the output).
test_m = model_m.outlier_test()
print("Bad data points (bonf(p) < 0.05):\n", test_m[test_m['bonf(p)'] < 0.05])
```

```
Bad data points (bonf(p) < 0.05):
 Empty DataFrame
Columns: [student_resid, unadj_p, bonf(p)]
Index: []
```

```
# Save the final outliers.
test_final_m = test_m[test_m['bonf(p)'] < 0.05]
```

Removing outliers from regression dataset

- Now that we have a saved dataframe of outliers, let's remove them from our original dataset
- In our case, there were no outliers
- However, we will create a new training dataset anyway, for the sake of getting familiar with the code

```
test_final_m = test_m[test_m['bonf(p)'] < 0.05]
# Make sure that you drop outliers from both X and y train sets.
X_train_no_outliers = X_train.drop(test_final_m.index)
y_train_no_outliers = y_train.drop(test_final_m.index)
# Look at the shape of the new dataframe to check that the rows have actually been dropped.
print(X_train_no_outliers.shape)
```

```
(88, 3)
```

```
print(y_train_no_outliers.shape)
```

```
(88, 1)
```

Rerun multiple regression model

- If we had dropped some outliers, we could rerun the model with the code below using `sm.OLS` and the new training dataset
- If we had dropped some outliers, we could also re-evaluate the model with the assumptions. We are not going to do that here, since we dropped no outliers

```
# Build a linear model on training data.
model_m_no_outliers = sm.OLS(y_train_no_outliers, X_train_no_outliers).fit()
```

```
model_m_no_outliers.summary()
```

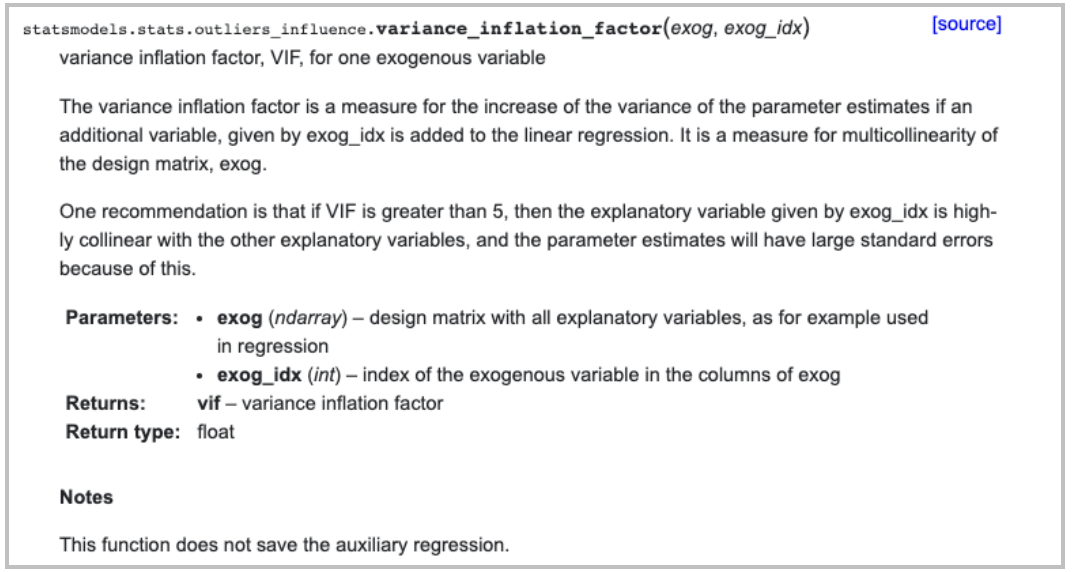
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Calories   R-squared:                       0.949
Model:                            OLS   Adj. R-squared:                  0.948
Method:                 Least Squares   F-statistic:                     790.2
Date:                Tue, 23 Jul 2019   Prob (F-statistic):           1.22e-55
Time:                        10:19:25   Log-Likelihood:                -485.05
No. Observations:                  88   AIC:                             976.1
Df Residuals:                      85   BIC:                             983.5
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
constant       119.6427     12.124      9.868      0.000      95.536     143.749
Total Fat (g)   13.2056      0.348     37.972      0.000      12.514      13.897
Sugars (g)       2.6217      0.355      7.385      0.000       1.916       3.327
==============================================================================
Omnibus:                        1.534   Durbin-Watson:                   1.978
Prob(Omnibus):                  0.464   Jarque-Bera (JB):                1.326
Skew:                          -0.138   Prob(JB):                        0.515
Kurtosis:                       2.465   Cond. No.                         67.7
==============================================================================
```

Correlated variables

- Another issue we should always check for in multiple linear regression is **correlation between predictor variables**
- Correlation between predictor variables is also called **multicollinearity**
- For this, we look at the **variance inflation factor (VIF)**

Variance inflation factor (VIF)

- VIF is obtained using the R-squared value of the regression of that variable against all other explanatory variables
- A VIF is calculated for each predictor, those with high values are removed
- The definition of 'high' is somewhat arbitrary but usually between 5-10
- We'll define it as 10 for this model
- We will use the `variance_inflation_factor` function from `statsmodels.stats.outliers_influence`



`statsmodels.stats.outliers_influence.`**`variance_inflation_factor`**(*exog*, *exog_idx*) [source]

variance inflation factor, VIF, for one exogenous variable

The variance inflation factor is a measure for the increase of the variance of the parameter estimates if an additional variable, given by exog_idx is added to the linear regression. It is a measure for multicollinearity of the design matrix, exog.

One recommendation is that if VIF is greater than 5, then the explanatory variable given by exog_idx is highly collinear with the other explanatory variables, and the parameter estimates will have large standard errors because of this.

Parameters:
- **exog** (*ndarray*) – design matrix with all explanatory variables, as for example used in regression
- **exog_idx** (*int*) – index of the exogenous variable in the columns of exog

Returns: **vif** – variance inflation factor

Return type: float

Notes

This function does not save the auxiliary regression.

Testing the model

- **Let's see if our model is at risk for collinearity**
- We build a dataframe `vif` and we populate it using the `variance_inflation_factor` function on each of the predictors in `X_train_no_outliers`

```
vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(X_train_no_outliers.values, i) for i in
range(X_train_no_outliers.shape[1])]
vif["features"] = X_train_no_outliers.columns
print(vif)
```

```
   VIF Factor        features
0    3.480119        constant
1    1.013046  Total Fat (g)
2    1.013046     Sugars (g)
```

- Looks good! No, VIF factor is higher than 10
- There is no multicollinearity in our model

Knowledge Check 3

Exercise 3

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	✔
Define best practices of model building and what the training / test sets are and when to use them	✔
Implement and evaluate a multiple linear regression model on the training set	✔
Manage influential points and highly correlated variables as necessary and rerun optimized model	✔
Predict on the test dataset using statsmodel and evaluate the model using RMSE	

Predicting on fast_food test data

- So far today, we have:
  - cleaned our `fast_food` dataset
  - split our cleaned data into test and train
  - evaluated our model's metrics
  - validated that it meets the assumptions of linear regression
  - looked at the influential points
  - looked for any highly correlated variables
- Remember this process when working with multiple linear regression
- Now we can test how the final model performs on data it **has not seen** by using our training model, `model_m_no_outliers`, to **PREDICT** values for our reserved test set

Introducing predict

- `statsmodels.api` that we imported earlier today has a `predict` function that we will now use
- We'll use our hold out sample `X_test`
- **Hold out data** is the validation data we will use to validate the model
- This way, we can also test the error of the trained model by evaluating the residuals

$$actual - predicted$$

statsmodels.regression.linear_model.OLS.predict

`OLS.predict(params, exog=None)`

Return linear predicted values from a design matrix.

Parameters:
- **params** (*array-like*) – Parameters of a linear model
- **exog** (*array-like, optional.*) – Design / exogenous data. Model exog is used if None.

Returns:

Return type: An array of fitted values

Notes

If the model has not yet been fit, params is not optional.

Predict: `Calories` in test data

We can now use `predict` to calculate `Calories` in the test dataset

```
# Predict values of `Calories` using the test data.
prediction = model_m_no_outliers.predict(X_test)
print(prediction.head())
```

```
120     440.360393
46      682.044927
73      344.137642
90     1085.980138
102     352.002607
dtype: float64
```

Predict: residuals of model

- Let's build a dataframe that contains:
  - `actual`: using `y_test.Calories`
  - `predicted`: using the predictions we just calculated
  - `residuals`: the `actual - predicted`

```
actual = y_test.Calories
prediction = model_m_no_outliers.predict(X_test)
residuals = y_test.Calories - prediction


fast_food_results =
pd.concat([actual.rename('actual'),

prediction.rename('predicted'),

residuals.rename('residuals')], axis = 1)
```

```
print(fast_food_results.head())
```

```
     actual    predicted  residuals
120     430   440.360393 -10.360393
46      660   682.044927 -22.044927
73      380   344.137642  35.862358
90     1030  1085.980138 -55.980138
102     300   352.002607 -52.002607
```

Predict: mean squared error

- Root mean squared error objectively **assesses model performance**
- It is a metric that compares various methods that give us a continuous output
- This allows us to **compare models** and come up with a final model champion
- We are going to define a function so we know exactly what is going on
- This is the formula we want in our function:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

```
def rmse(predictions,actual):
    return np.sqrt(((prediction-actual) **
2).mean())

print(rmse(prediction,actual))
```

```
65.83430802746773
```

Regression: key points

- A couple things to remember:
  - A linear model can be **garbage**
  - There are common issues across bad models
  - Residuals are helpful in creating **useful** models

Pickle RMSE

- In order for us to compare other continuous models to this model, let us pickle the final model's RMSE
- RMSE can be used to compare various methods that give us a continuous output
- RMSE is used to assess the performance of the models
- After using RMSE what can we say about the performance of our model?
- We cans see that the error rate decreases

```
multiple_regression_RMSE = rmse(prediction,actual)
pickle.dump(multiple_regression_RMSE, open("multiple_regression_RMSE.sav", "wb" ))
```

Knowledge Check 4

Exercise 4

Module completion checklist

Objective	Complete
Understand multiple linear regression and introduce dataset	✔
Summarize how to implement multiple regression	✔
Clean the new multiple regression dataset and use EDA to understand our data	✔
Define best practices of model building and what the training / test sets are and when to use them	✔
Implement and evaluate a multiple linear regression model on the training set	✔
Manage influential points and highly correlated variables as necessary and rerun optimized model	✔
Predict on the test dataset using statsmodel and evaluate the model using RMSE	✔

Workshop: Next steps!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code
- This is an exploratory exercise to get you comfortable with the content we discussed today

First stage

- Go back and predict on the single variable regression model from yesterday
- Calculate the RMSE for this model, how does it compare to the multiple regression model?
- Add 1-2 more variables to today's model, make sure they are continuous
- Run, evaluate and optimize the new, additional variable model

Second stage

- Add a variable to the model you built on your own data
- Clean your data and then split into train and test sets
- See how it does, evaluate it thoroughly on train (assumptions, outliers, collinearity)
- Predict using your optimized model on the test dataset

This completes module!

Congratulations!