MODEL ASSUMPTIONS

MODEL ASSUMPTIONS

In this section we'll cover the **assumptions of linear regression** models which should be checked and met to ensure that the model's predictions and interpretation are valid

TOPICS WE'LL COVER:

- Assumptions Overview
- Linearity
- Independence of Errors
- Normality of Errors
- No Perfect Multicollinearity
- Equal Variance of Errors
- Outliers & Influence

GOALS FOR THIS SECTION:

- Review the assumptions of linear regression models
- Learn to diagnose and fix violations to each assumption using Python
- Assess the influence of outliers on a regression model, and learn methods for dealing with them

ASSUMPTIONS OF LINEAR REGRESSION

There are a few key **assumptions of linear regression** models that can be violated, leading to unreliable predictions and interpretations

- If the goal is *inference*, these all need to be checked rigorously
- If the goal is *prediction*, some of these can be relaxed

1. **Linearity**: a linear relationship exists between the target and features
2. **Independence of errors**: the residuals are not correlated
3. **Normality of errors**: the residuals are approximately normally distributed
4. **No perfect multicollinearity**: the features aren't perfectly correlated with each other
5. **Equal variance of errors**: the spread of residuals is consistent across predictions

You can use the **L.I.N.N.E** acronym (like *linear* regression) to remember them

It's worth noting that you might see resources saying there are anywhere from 3 to 6 assumptions, but these 5 are the ones to focus on

LINEARITY

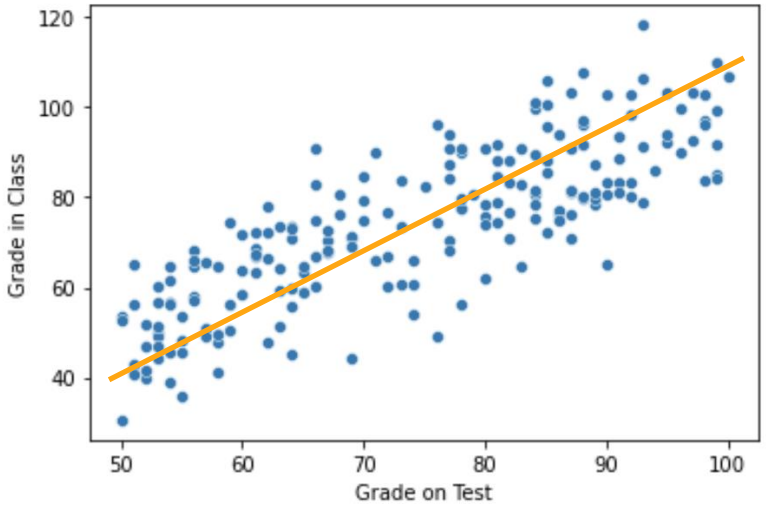
- Assumptions Overview
- Linearity
- Independence of Errors
- Normality of Errors
- No Perfect Multicollinearity
- Equal Variance of Errors
- Outliers & Influence

Linearity assumes there's a linear relationship between the target and each feature

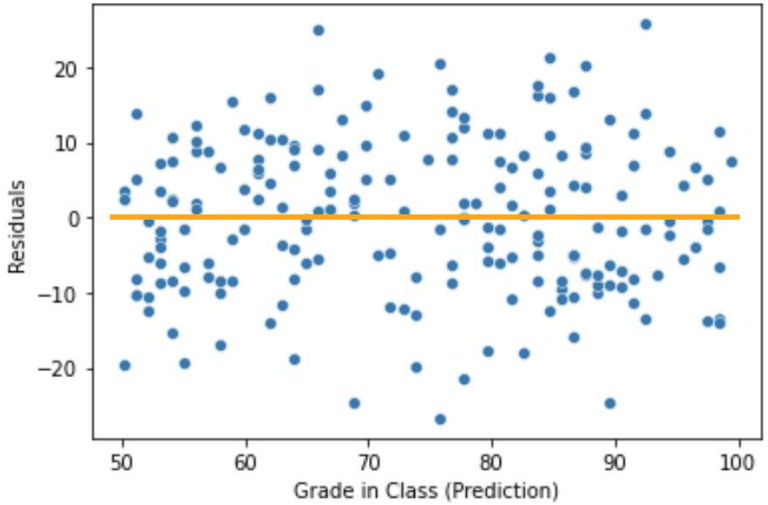
- If this assumption is violated, it means the model isn't capturing the underlying relationship between the variables, which could lead to inaccurate predictions

You can diagnose linearity by using **scatterplots** and **residual plots**:

Ideal Scatterplot



Ideal Residual Plot



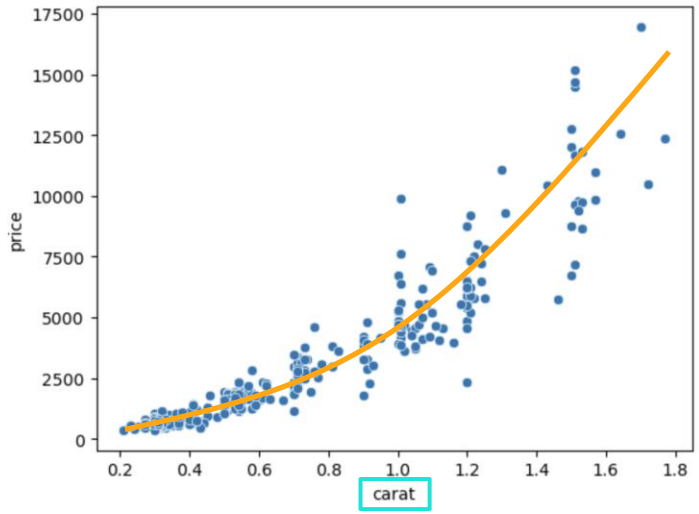
LINEARITY

You can fix linearity issues by **transforming features** with non-linear relationships

- Common transformations include polynomial terms (x^2, x^3, *etc.*) and log transformations (*log(x)*)

Model Sample (carat vs price)

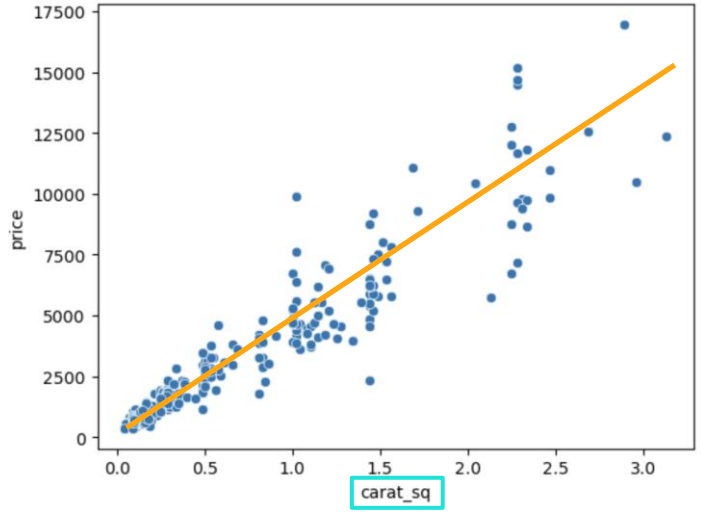
```
d_sample = diamonds.sample(300)

sns.scatterplot(d_sample, x="carat", y="price");
```



The scatterplot has a U-like shape, which is similar to the $y=x^2$ plot, so let's try squaring the "carat" feature

Model Sample (carat^2 vs price)

```
d_sample["carat_sq"] = d_sample["carat"]**2

sns.scatterplot(d_sample, x="carat_sq", y="price");
```



The "carat_sq" has a much more linear relationship with the "price" target variable

LINEARITY

- Assumptions Overview
- **Linearity**
- Independence of Errors
- Normality of Errors
- No Perfect Multicollinearity
- Equal Variance of Errors
- Outliers & Influence

You can fix linearity issues by **transforming features** with non-linear relationships

- Common transformations include polynomial terms (x^2, x^3, *etc.*) and log transformations (*log(x)*)

```
diamonds["carat_sq"] = diamonds["carat"]**2

features = [
    "carat",
    "carat_sq",
    "depth",
    "table",
    "x",
    "y"
]

X = sm.add_constant(diamonds.loc[:, features])
y = diamonds["price"]

model = sm.OLS(y, X).fit()

model.summary()
```

When adding polynomial terms, you need to include the lower order terms to the model, regardless of significance

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.862
Model:	OLS	**Adj. R-squared:**	0.862
Method:	Least Squares	**F-statistic:**	5.621e+04
Date:	Mon, 07 Aug 2023	**Prob (F-statistic):**	0.00
Time:	19:37:20	**Log-Likelihood:**	-4.7036e+05
No. Observations:	53943	**AIC:**	9.407e+05
Df Residuals:	53936	**BIC:**	9.408e+05
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	3.055e+04	507.475	60.194	0.000	2.96e+04	3.15e+04
carat	1.707e+04	199.145	85.697	0.000	1.67e+04	1.75e+04
carat_sq	-1340.7930	39.739	-33.740	0.000	-1418.681	-1262.905
depth	-271.0651	5.237	-51.757	0.000	-281.330	-260.800
table	-114.8015	3.073	-37.354	0.000	-120.825	-108.778
x	-2763.0880	56.306	-49.073	0.000	-2873.448	-2652.728
y	16.2115	25.074	0.647	0.518	-32.933	65.356

R^2 increased from 0.859

We can drop "y" ($p > 0.05$)

INDEPENDENCE OF ERRORS

Independence of errors assumes that the residuals in your model have no patterns or relationships between them *(they aren't autocorrelated)*

- In other words, it checks that you haven't fit a linear model to time series data

You can diagnose independence with the **Durbin-Watson Test**:

- **H_o: DW=2** – the errors are NOT autocorrelated
- **H_a: DW≠2** – the errors are autocorrelated
- As a rule of thumb, values between 1.5 and 2.5 are accepted

```
features = ["carat", "carat_sq", "depth", "table", "x"]

X = sm.add_constant(diamonds.loc[:, features])
y = diamonds["price"]

model = sm.OLS(y, X).fit()

model.summary()
```

Omnibus:	13855.832	**Durbin-Watson:**	1.999
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	293840.413
Skew:	0.723	**Prob(JB):**	0.00
Kurtosis:	14.342	**Cond. No.**	6.97e+03

All good!

You can fix independence issues by using a time series model *(more later!)*

INDEPENDENCE OF ERRORS

IMPORTANT: If your data is sorted by your target, or potentially an important predictor variable, it can cause this assumption to be violated

- Use **df.sample(frac=1)** to fix this by randomly shuffling your dataset rows

Model (sorted by price)

```
diamonds = diamonds.sort_values("price")

X = sm.add_constant(diamonds.loc[:, features])
y = diamonds["price"]

model = sm.OLS(y, X).fit()

model.summary()
```

Omnibus:	13855.832	**Durbin-Watson:**	1.252
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	293840.413
Skew:	0.723	**Prob(JB):**	0.00
Kurtosis:	14.342	**Cond. No.**	6.97e+03

Sorting the DataFrame by the target (price) leads to a Durbin-Watson statistic outside the desired range

Model (randomized rows)

```
diamonds = diamonds.sample(frac=1)

X = sm.add_constant(diamonds.loc[:, features])
y = diamonds["price"]

model = sm.OLS(y, X).fit()

model.summary()
```

Omnibus:	13855.832	**Durbin-Watson:**	2.002
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	293840.413
Skew:	0.723	**Prob(JB):**	0.00
Kurtosis:	14.342	**Cond. No.**	6.97e+03

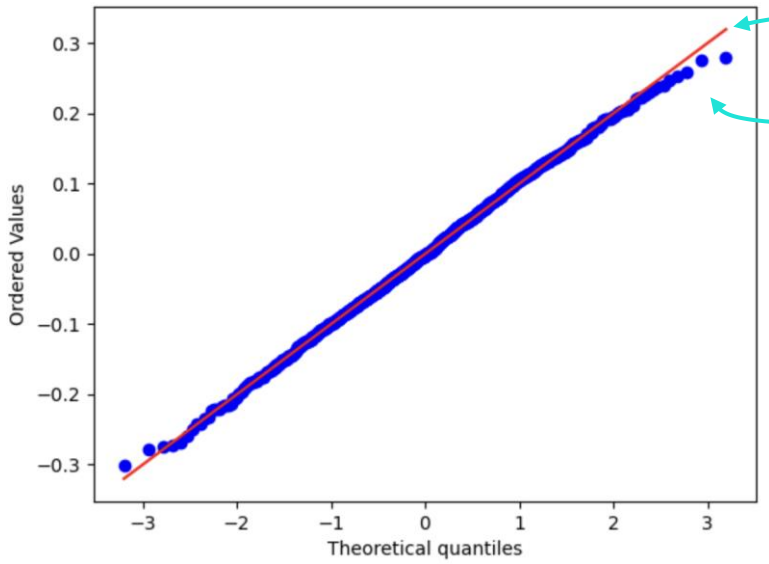
Randomizing the order brings things back to normal

NORMALITY OF ERRORS

Normality of errors assumes the *residuals* are approximately normally distributed

You can diagnose normality by using a **QQ plot** (quantile-quantile plot):

Ideal QQ Plot



The x-axis represents the number of standard deviations away from the mean residuals lie

*The **red line** represents a normal distribution*

It's normal for a few points to fall off the line, particularly outside of 2 standard deviations

The **Jarque-Bera Test** (JB) in the model summary tests the normality of errors

However, this test has numerous issues and is far too restrictive to use in practice

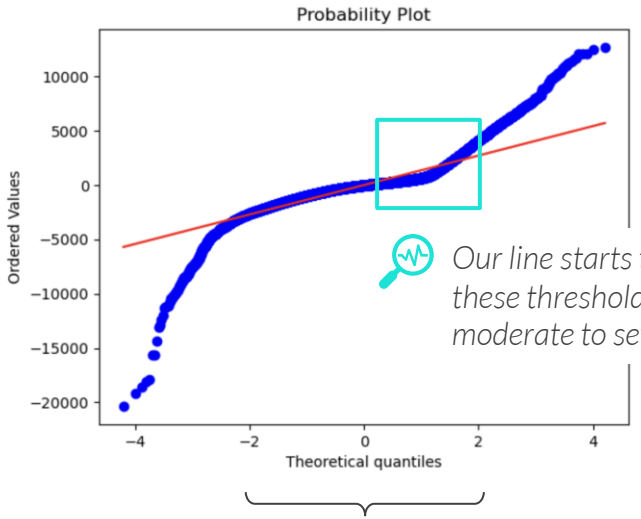
NORMALITY OF ERRORS

You can typically fix normality issues by applying a **log transform on the target**

- Other options are applying log transforms to features or simply leaving the data as is

Price Model

```
import scipy.stats as stats
import matplotlib.pyplot as plt

stats.probplot(model.resid, dist="norm", plot=plt);
```

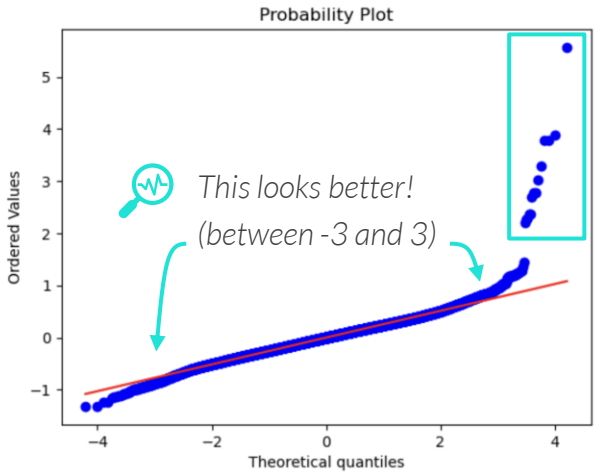


Our line starts to curve in between these thresholds, which indicates moderate to severe non-normality

You generally want to see points fall along the line in between -2 and +2 standard deviations

Log of Price Model

```
import numpy as np

X = sm.add_constant(diamonds.loc[:, features])
y = np.log(diamonds["price"])

model = sm.OLS(y, X).fit()

stats.probplot(model.resid, dist="norm", plot=plt);
```



This looks better! (between -3 and 3)

There are still large residuals left, but they don't violate the normality assumption (check later)

PRO TIP: INTERPRETING TRANSFORMED TARGETS

Transforming your target fundamentally changes the interpretation of your model, so you need to **invert the transformation** to understand coefficients & predictions

- Inverting a feature coefficient returns the associated multiplicative change in the target's value
- Inverting a prediction returns the target in its original units

```
X = sm.add_constant(diamonds.loc[:, features])
y = np.log(diamonds["price"])

model = sm.OLS(y, X).fit()

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	5.4119	0.089	60.924	0.000	5.238	5.586
carat	2.3598	0.035	67.643	0.000	2.291	2.428
carat_sq	-0.6431	0.007	-92.340	0.000	-0.657	-0.629
depth	-0.0081	0.001	-8.874	0.000	-0.010	-0.006
table	-0.0159	0.001	-29.451	0.000	-0.017	-0.015
x	0.4294	0.009	46.887	0.000	0.411	0.447

*A 1-unit increase in carat is associated with a **10.6X increase in price***

($e^{2.3598} = 10.59$)

```
#[constant, carat, carat_sq, depth, table, x]
diamond = [1, 1.5, 1.5**2, 62, 59, 7.2]

np.exp(model.predict(diamond))
```

```
array([9458.79802436])
```

*A diamond with these features is predicted to cost **$9,458***

Transformation	Inverse
y = np.sqrt(x)	x = y**2
y = np.log(x)	x = np.exp(y)
y = np.log10(x)	x = 10 ** y
y = 1/x	x = 1/y

NO PERFECT MULTICOLLINEARITY

No perfect multicollinearity assumes that features aren't perfectly correlated with each other, as that would lead to unreliable and illogical model coefficients

- If two features have a correlation (r) of 1, there are infinite ways to minimize squared error
- Even if it's not perfect, strong multicollinearity (r > 0.7) can still cause issues to a model

You can diagnose multicollinearity with the **Variance Inflation Factor** (VIF):

- Each feature is treated as the target, and R^2 measures how well the other features predict it
- As a rule of thumb, a VIF > 5 indicates that a variable is causing multicollinearity problems

```
from statsmodels.stats.outliers_influence import variance_inflation_factor as vif

variables = sm.OLS(y, X).exog

pd.Series([vif(variables, i) for i in range(variables.shape[1])], index=X.columns)
```

```
const       6289.392199
carat        217.938040
carat_sq      43.152926
depth          1.378775
table          1.156815
x             84.132321
dtype: float64
```

We can ignore the VIF for the intercept, but most of our variables have a VIF > 5

NO PERFECT MULTICOLLINEARITY

There are several ways to fix multicollinearity issues:

- The most common is to **drop features** with a VIF > 5 *(leave at least 1 to see the impact)*
- Another is to **engineer combined features** *(like "GDP" and "Population" to "GDP per capita")*
- You can also turn to **regularized regression** or **tree-based models** *(more later!)*

Original Model

```
const        6289.392199
carat         217.938040
carat_sq       43.152926
depth           1.378775
table           1.156815
x              84.132321
dtype: float64
```

Removing *x*

```
const        3539.505406
carat          11.182845
carat_sq       11.055436
depth           1.105012
table           1.146514
dtype: float64
```

We still have VIF > 5 for our carat terms, but we can generally ignore those since they are polynomial terms

The VIF can also be ignored when using **dummy variables** (more later!)

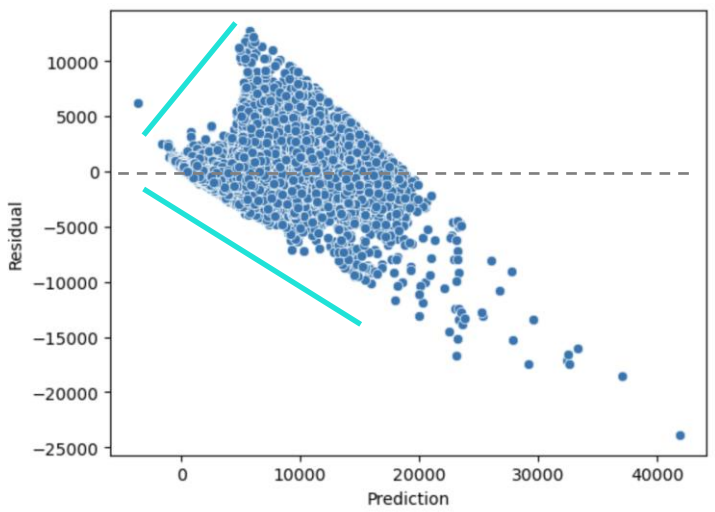
EQUAL VARIANCE OF ERRORS

Equal variance of errors assumes the residuals are consistent across predictions

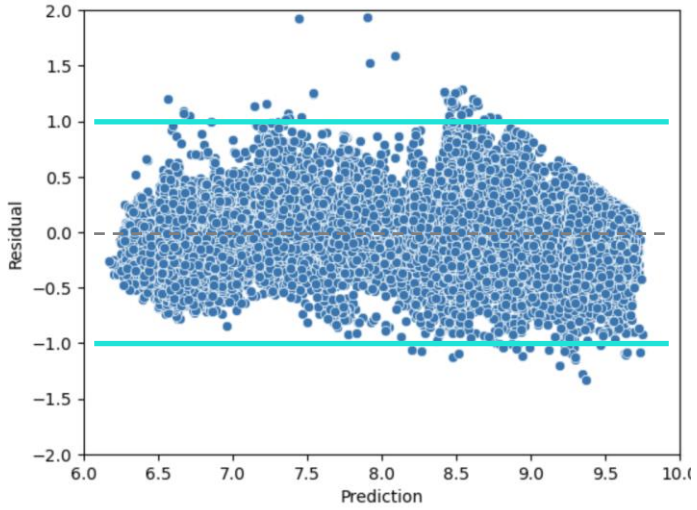
- In other words, average error should stay roughly the same across the range of the target
- Equal variance is known as **homoskedasticity**, and non-equal variance is **heteroskedasticity**

You can diagnose heteroskedasticity with **residual plots**:

Heteroskedasticity
(original regression model)



Homoskedasticity
(after fixing violated assumptions)



Our model predicts much better now!

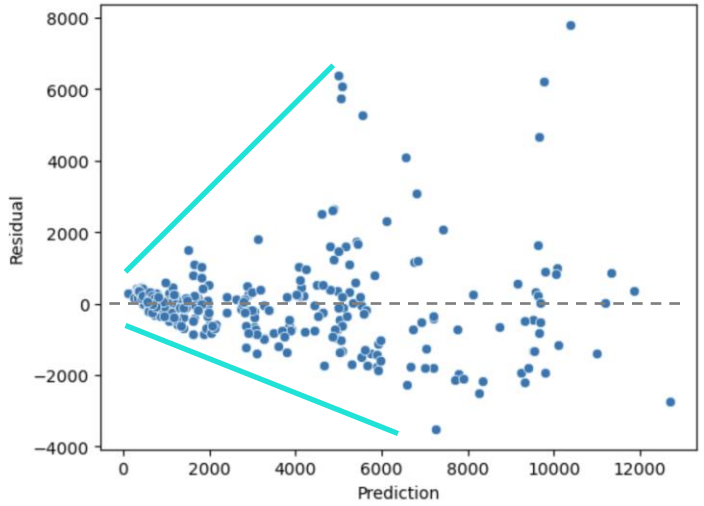
EQUAL VARIANCE OF ERRORS

You can typically fix heteroskedasticity by **applying a log transform** on the target

- In other words, the average error should stay roughly the same across the target variable

Price Model

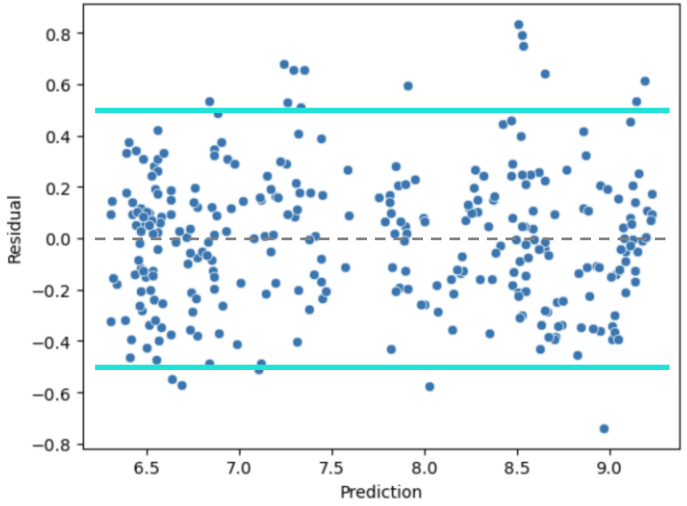
```
X = sm.add_constant(d_sample.loc[:, features])
y = d_sample["price"]

model = sm.OLS(y, X).fit()

sns.scatterplot(x=model.predict(), y=model.resid);
```



*Errors have a **cone shape** along the x-axis*

Log of Price Model

```
X = sm.add_constant(d_sample.loc[:, features])
y = np.log(d_sample["price"])

model = sm.OLS(y, X).fit()

sns.scatterplot(x=model.predict(), y=model.resid);
```



*Errors are **spread evenly** along the x-axis*

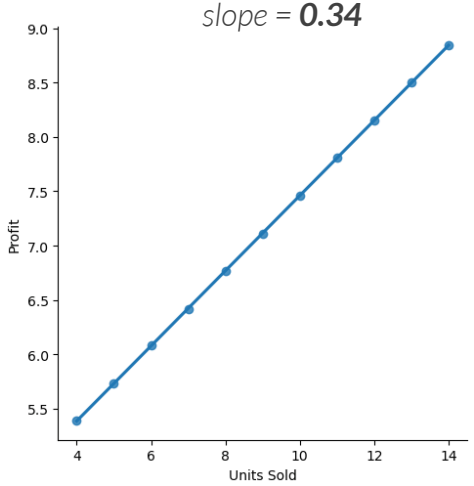
OUTLIERS & INFLUENCE

Outliers are extreme data points that fall well outside the usual pattern of data

- Some outliers have dramatic **influence** on model fit, while others won't
- Outliers that impact a regression equation significantly are called **influential points**

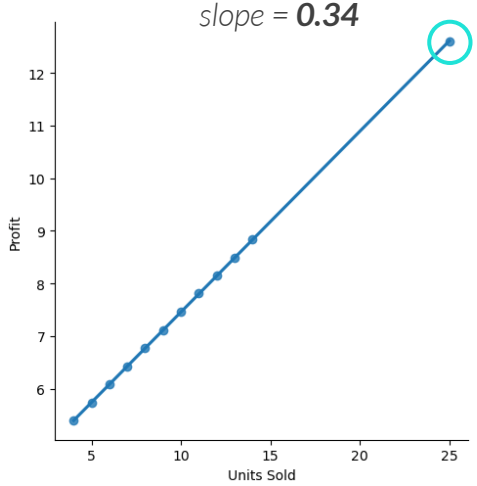
No Outliers

*slope = **0.34***



Non-influential Outlier

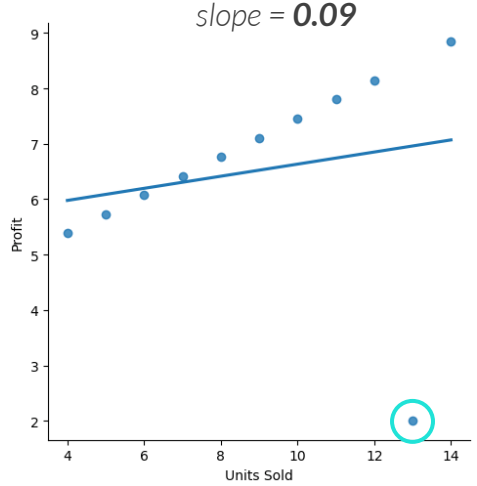
*slope = **0.34***



This is an outlier in both profit and units sold, but it's in line with the rest of the data, so it's not influential

Influential Outlier

*slope = **0.09***



This is an outlier in terms of profit that doesn't follow the same pattern as the rest of the data, so it changes the regression line

An outlier's influence **depends on the size of the dataset**
(large datasets are impacted less)

OUTLIERS & INFLUENCE

- Assumptions Overview
- Linearity
- Independence of Errors
- Normality of Errors
- No Perfect Multicollinearity
- Equal Variance of Errors
- Outliers & Influence

Cook's Distance measures the influence a data point has on the regression line

- It works by fitting a regression without the point and calculating the impact
- Cook's D > 1 is considered a significant problem, while Cook's D > 0.5 is worth investigating
- Use **.get_influence().summary_frame()** on a regression model to return influence statistics

```
influence = model.get_influence()
inf_summary_df = influence.summary_frame()
inf_summary_df.head(2)
```

	dfb_const	dfb_carat	dfb_carat_sq	dfb_depth	dfb_table	cooks_d	standard_resid	hat_diag	dffits_internal	student_resid	dffits
34922	0.001189	-0.003014	0.002259	-0.000472	-0.001511	0.000006	-0.721335	0.000054	-0.005286	-0.721332	-0.005286
12330	0.001943	0.000297	-0.000960	0.000947	-0.005508	0.000017	1.373486	0.000044	0.009106	1.373497	0.009106

The other metrics can be ignored

```
inf_summary_df["cooks_d"].sort_values(ascending=False).head()
```

```
14403    8.652876
35846    2.282304
39151    0.740038
4131     0.448035
29088    0.429981
Name: cooks_d, dtype: float64
```

Not surprisingly, the most influential points were the largest diamonds!

```
X.loc[[14403, 35846, 39151]]
```

	const	carat	carat_sq	depth	table
14403	1.0	5.01	25.1001	65.5	59.0
35846	1.0	4.50	20.2500	65.8	58.0
39151	1.0	4.13	17.0569	64.8	61.0

OUTLIERS & INFLUENCE

There are several ways to deal with influential points:

- You can **remove them** or **leave them in**
- You can **engineer features** to help capture their variance
- You can use **robust regression** *(outside the scope of this course)*

Model (with outliers)

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.932
Model:	OLS	**Adj. R-squared:**	0.932
Method:	Least Squares	**F-statistic:**	1.836e+05
Date:	Wed, 09 Aug 2023	**Prob (F-statistic):**	0.00
Time:	12:03:15	**Log-Likelihood:**	-4982.5
No. Observations:	53943	**AIC:**	9975.
Df Residuals:	53938	**BIC:**	1.002e+04

	coef	std err	t	P>|t|	[0.025	0.975]
const	8.1659	0.068	120.117	0.000	8.033	8.299
carat	3.9530	0.008	490.335	0.000	3.937	3.969
carat_sq	-0.9248	0.004	-257.139	0.000	-0.932	-0.918
depth	-0.0273	0.001	-32.592	0.000	-0.029	-0.026
table	-0.0183	0.001	-33.355	0.000	-0.019	-0.017

Model (without 2 outliers)

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.933
Model:	OLS	**Adj. R-squared:**	0.933
Method:	Least Squares	**F-statistic:**	1.889e+05
Date:	Wed, 09 Aug 2023	**Prob (F-statistic):**	0.00
Time:	12:03:40	**Log-Likelihood:**	-4269.4
No. Observations:	53941	**AIC:**	8549.
Df Residuals:	53936	**BIC:**	8593.

	coef	std err	t	P>|t|	[0.025	0.975]
const	8.1992	0.067	122.200	0.000	8.068	8.331
carat	4.0257	0.008	491.983	0.000	4.010	4.042
carat_sq	-0.9610	0.004	-261.491	0.000	-0.968	-0.954
depth	-0.0280	0.001	-33.812	0.000	-0.030	-0.026
table	-0.0186	0.001	-34.428	0.000	-0.020	-0.018

R^2 improved slightly and the coefficients changed a bit, but not much changed (this is a large dataset)

RECAP: ASSUMPTIONS OF LINEAR REGRESSION

Assumption	Problem to solve	Effect on Model	Diagnosis	Fix	Drawbacks to Fix	Should I Fix it?
Linearity	Feature-target relationship is not linear	Suboptimal model accuracy, non-normal residuals	Curved patterns in feature-target scatterplots	Add polynomial terms	Slightly less intuitive coefficients	The more curved the relationship, the worse it is. It's usually worth fixing as it can improve accuracy.
Independence of Errors	Residuals aren't independent of each other	Suboptimal model structure	Durbin-Watson stat not between 1.5–2.5	Use time-series modeling techniques	None	Yes
Normality of Errors	Residuals aren't normally distributed around 0	Inaccurate predictions & less reliable coefficient estimates	Data significantly deviates from the line of normality in QQ plot inside of +/- 2 standard deviations	Transform the target with log or other transforms. Add features to model if available	Less interpretability	If it's due to a non-linear pattern or missing variable, or if the fix significantly improves accuracy (unless you need a simple model)
No Perfect Multicollinearity	Features with perfect, or near perfect, correlation with each other	Unstable & illogical model coefficients	Features have a Variance Inflation Factor greater than 5	Drop features or use regularized regression	May lose some training accuracy	For perfect or near perfect collinearity, yes. For features closer to r= 0.7, gains seen in training usually offset accuracy.
Equal Variance of Errors	Residuals are not consistent across the full range of predicted values	Unreliable confidence intervals for coefficients & predictions	Cone-shaped residual plots	Transform the target.	Less interpretability	This is generally ok to leave as is if your goal is prediction
Outliers & Influence	Influential data points	Lower accuracy & possible violated assumptions	Cook's D greater than 1 (highly influential) or 0.5 (potentially problematic)	Remove data points or engineer features to explain outliers	Removing data points is not ideal if we need to predict them	If we expect the data to have similar points, removing them won't change model performance (consider a model with and without outliers)

ASSIGNMENT: MODEL ASSUMPTIONS

NEW MESSAGE

July 10, 2023

From: **Cam Correlation** *(Sr. Data Scientist)*

Subject: **Model Assumptions**

Hi there!

Can you check the model assumptions for our computer price regression model?

Make any changes necessary to improve model fit!

Don't force it though, use your best judgement and remember that not all assumptions will be violated and there are some trade-offs for fixes!

Thanks!

04_assumptions_assignments.ipynb

Reply | Forward

Key Objectives

1. Check for violated assumptions on a fitted regression model
2. Implement fixes for assumptions
3. Decide whether the fixes are worth the trade-offs and settle on a final model

KEY TAKEAWAYS

Linear regression models have **5 key assumptions** that must be checked

- *Linearity, independence of errors, normality or errors, no perfect multicollinearity, and equal variance of errors*
- *If the goal is inference, they need to be checked rigorously, but for prediction some of them can be relaxed*

Diagnosing & fixing these assumptions can help improve model accuracy

- *Use residual plots, QQ plots, the Durbin-Watson test, and the Variance Inflation Factor to diagnose*
- *Transforming the features and/or target (polynomial terms, log transforms, etc.) can typically help fix issues*

Outliers that significantly impact a model are known as **influential points**

- *Use Cook's distance to identify influential points (Cook's D > 1)*
- *Before removing influential points, consider if you expect to encounter similar data points in new data*

MODEL TESTING & VALIDATION

MODEL TESTING & VALIDATION

In this section we'll cover **model testing & validation**, which is a crucial step in the modeling process designed to ensure that a model performs well on new, unseen data

TOPICS WE'LL COVER:

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

GOALS FOR THIS SECTION:

- Understand the concept of data splitting and its impact on model fit, bias, and variance
- Split data in Python into training and test data sets, and then into training and validation data sets
- Learn the concept of cross validation, and its advantages & disadvantages over simple validation
- Evaluate models by tuning on training & validation data, refitting on both, then scoring on test data

RECAP: REGRESSION MODELING WORKFLOW

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

1 Scoping a project → 2 Gathering data ⇄ 3 Cleaning data ⇄ 4 Exploring data ⇄ **5 Modeling data** → 6 Sharing insights

Preparing for Modeling

Get your data ready to be input into an ML algorithm

- Single table, non-null
- Feature engineering
- Data splitting

Applying Algorithms

Build regression models from training data

- Linear regression
- Regularized regression
- Time series

Model Evaluation

Evaluate model fit on training & validation data

- R-squared & MAE
- Checking assumptions
- Validation performance

Model Selection

Pick the best model to deploy and identify insights

- Test performance
- Interpretability

This is what we've covered so far

RECAP: REGRESSION MODELING WORKFLOW

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

1 Scoping a project → 2 Gathering data ⇄ 3 Cleaning data ⇄ 4 Exploring data ⇄ **5 Modeling data** → 6 Sharing insights

Preparing for Modeling

Get your data ready to be input into an ML algorithm

- Single table, non-null
- Feature engineering
- **Data splitting**

*This needs to be the **first step** in the process*

Applying Algorithms

Build regression models from training data

- Linear regression
- Regularized regression
- Time series

Model Evaluation

Evaluate model fit on training & validation data

- R-squared & MAE
- Checking assumptions
- **Validation performance**

So we can do these properly

Model Selection

Pick the best model to deploy and identify insights

- **Test performance**
- Interpretability

MODEL SCORING STEPS

These **model scoring steps** need to be considered *before* you start modeling:

1. **Split the data** into a "training" and "test" set
   - *Around 80% of the data is used for training and 20% is reserved for testing*
2. Select a **validation method**
   - *Either split the training set into "training" and "validation" or use cross validation*
3. **Tune the model** using the training data
   - *Gradually improve the model by fitting on the training set and scoring on the validation set*
4. **Score the model** using the test data
   - *Once you've settled on a model, combine the training & validation data sets and refit the model, then score it on the test data to evaluate model performance*

DATA SPLITTING

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

Data splitting involves separating a data set into "training" and "test" sets

- **Training data**, or in-sample data, is used to fit and tune a model
- **Test data**, or out-of-sample data, provides realistic estimates of accuracy on new data

	carat	cut	color	clarity	depth	table	x	y	z	price
0	0.60	Fair	G	SI1	64.5	56.0	5.27	5.24	3.39	1305
1	2.00	Very Good	I	SI2	63.7	61.0	7.85	7.97	5.04	12221
2	0.34	Premium	G	VS2	61.1	60.0	4.51	4.53	2.76	596
3	0.36	Premium	E	SI2	62.4	58.0	4.56	4.54	2.84	605
4	0.45	Very Good	F	VS1	62.1	59.0	4.85	4.81	3.00	1179
5	0.30	Good	F	VS2	63.1	55.0	4.24	4.29	2.69	484
6	0.43	Ideal	D	SI1	61.6	56.0	4.86	4.82	2.98	1036
7	0.51	Ideal	G	VS1	62.0	56.0	5.16	5.06	3.17	1781
8	0.72	Ideal	F	VVS1	61.0	56.0	5.78	5.80	3.53	4362
9	0.31	Very Good	I	VVS1	62.8	55.0	4.33	4.36	2.73	571

***Training** data (80%)*

***Test** data (20%)*

DATA SPLITTING

Data splitting involves separating a data set into "training" and "test" sets

- **Training data**, or in-sample data, is used to fit and tune a model
- **Test data**, or out-of-sample data, provides realistic estimates of accuracy on new data

	carat	cut	color	clarity	depth	table	x	y	z	price
0	0.60	Fair	G	SI1	64.5	56.0	5.27	5.24	3.39	1305
1	2.00	Very Good	I	SI2	63.7	61.0	7.85	7.97	5.04	12221
2	0.34	Premium	G	VS2	61.1	60.0	4.51	4.53	2.76	596
3	0.36	Premium	E	SI2	62.4	58.0	4.56	4.54	2.84	605
4	0.45	Very Good	F	VS1	62.1	59.0	4.85	4.81	3.00	1179
5	0.30	Good	F	VS2	63.1	55.0	4.24	4.29	2.69	484
6	0.43	Ideal	D	SI1	61.6	56.0	4.86	4.82	2.98	1036
7	0.51	Ideal	G	VS1	62.0	56.0	5.16	5.06	3.17	1781
8	0.72	Ideal	F	VVS1	61.0	56.0	5.78	5.80	3.53	4362
9	0.31	Very Good	I	VVS1	62.8	55.0	4.33	4.36	2.73	571

← *In practice, the rows are sampled **randomly***

80/20 is the most common ratio for train/test data, but anywhere from **70-90% can be used for training**

For smaller data sets, a higher ratio of test data is needed to ensure a representative sample

DATA SPLITTING

You can split data in Python using scikit-learn's **train_test_split** function

- train_test_split(feature_df, target_column, test_pct, random_state)

```
from sklearn.model_selection import train_test_split

diamonds = diamonds.sample(1000)

X = sm.add_constant(diamonds[["carat"]])
y = diamonds["price"]

# Test Split
X, X_test, y, y_test = train_test_split(X, y, test_size=.2, random_state=12345)

print(
    f"Training Set Rows: {X.shape[0]}",
    f"Test Set Rows: {X_test.shape[0]}",
)
```

```
Training Set Rows: 800 Test Set Rows: 200
```

Perfect 80/20 split!

***4 outputs**:*

- *Training set for features (X)*
- *Test set for features (X_test)*
- *Training set for target (y)*
- *Test set for target (y_test)*

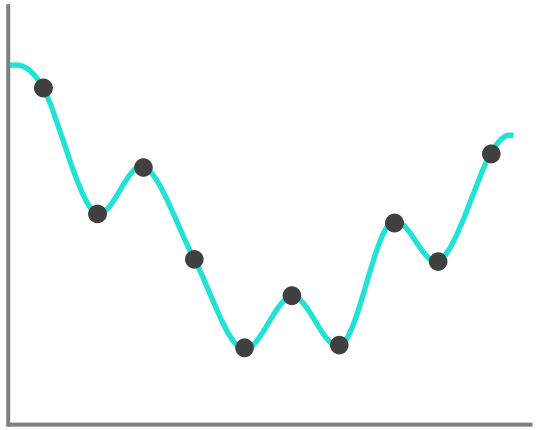
***4 inputs**:*

- *All feature columns*
- *The target column*
- *The percentage of data for the test set*
- *A random state (or a different split is created each time)*

OVERFITTING & UNDERFITTING

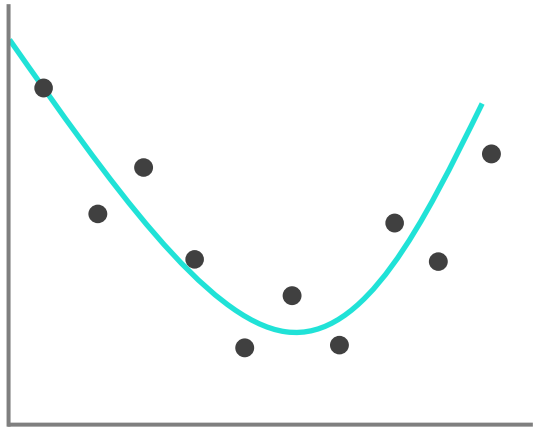
Data splitting is primarily used to avoid **overfitting**, which is when a model predicts known *(Training)* data very well but unknown *(Test)* data poorly

- Overfitting is like memorizing the answers to a test instead of *learning* the material; you'll ace the test, but lack the ability to ***generalize*** and apply your knowledge to unfamiliar questions



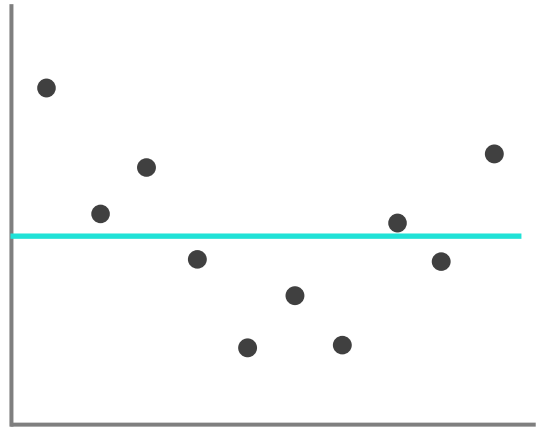
OVERFIT model

- *Models the training data too well*
- *Doesn't generalize well to test data*



WELL-FIT model

- *Models the training data just right*
- *Generalizes well to test data*



UNDERFIT model

- *Doesn't model training data well enough*
- *Doesn't score well on test data either*

OVERFITTING & UNDERFITTING

You can diagnose overfit & underfit models by **comparing evaluation metrics** like R^2, MAE, and RMSE between the train and test data sets

- Large gaps between train and test scores indicate that a model is **overfitting** the data
- Poor results across both train and test scores indicate that a model is **underfitting** the data

You can fix overfit models by:

- Simplifying the model
- Removing features
- Regularization *(more later!)*

You can fix underfit models by:

- Making the model more complex
- Add new features
- Feature engineering *(more later!)*

Models will usually have **lower performance on test data** compared to the performance on training data, so small gaps are expected – remember that no model is perfect!

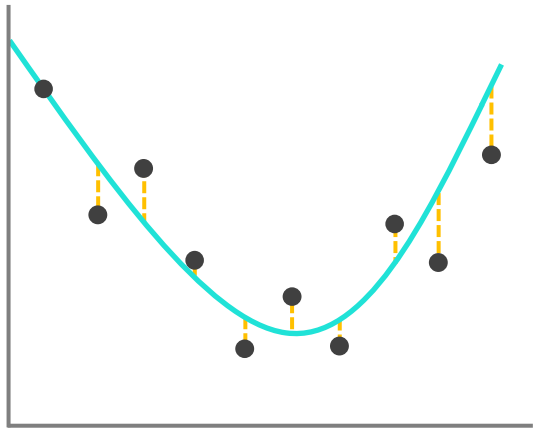
THE BIAS-VARIANCE TRADEOFF

When splitting data for regression, there are two types of errors that can occur:

- **Bias**: How much the model fails to capture the relationships in the training data
- **Variance**: How much the model fails to generalize to the test data

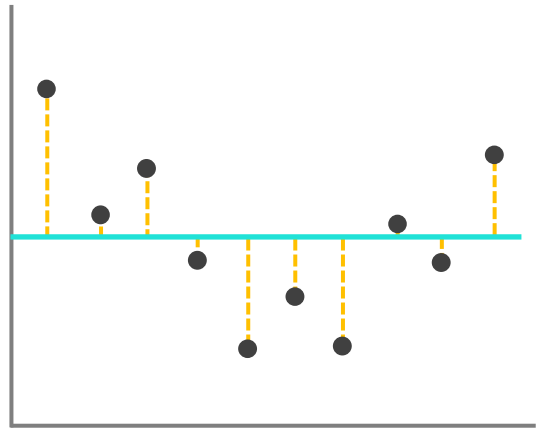
OVERFIT model

- ***Low bias*** *(no error)*



WELL-FIT model

- ***Medium bias*** *(some error)*



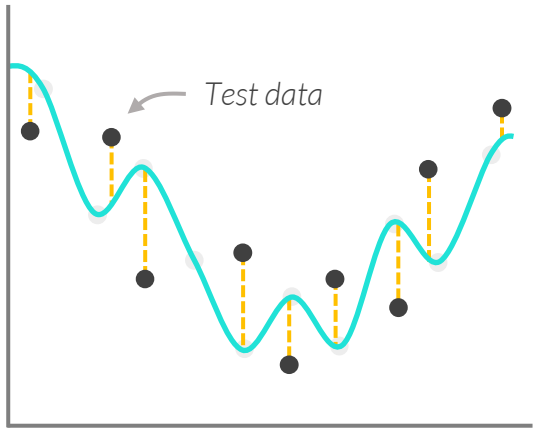
UNDERFIT model

- ***High bias*** *(lots of error)*

THE BIAS-VARIANCE TRADEOFF

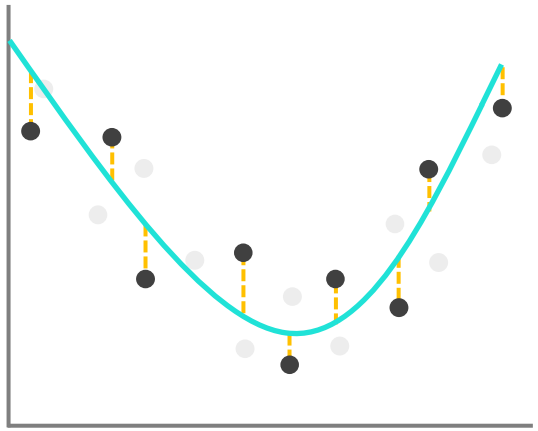
When splitting data for regression, there are two types of errors that can occur:

- **Bias**: How much the model fails to capture the relationships in the training data
- **Variance**: How much the model fails to generalize to the test data



OVERFIT model

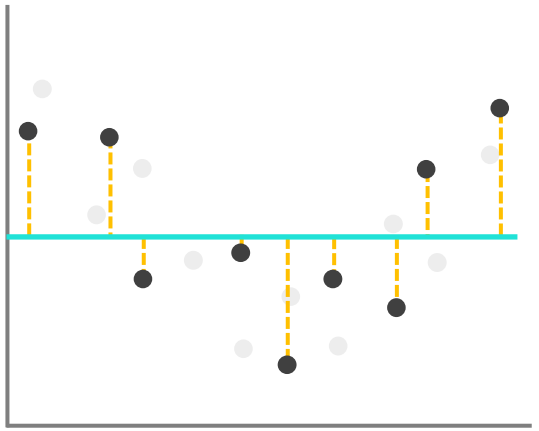
- ***Low bias*** *(no error)*
- ***High variance*** *(much more error)*



WELL-FIT model

- ***Medium bias*** *(some error)*
- ***Medium variance*** *(a bit more error)*

This is the bias-variance tradeoff!



UNDERFIT model

- ***High bias*** *(lots of error)*
- ***Low variance*** *(same error)*

THE BIAS-VARIANCE TRADEOFF

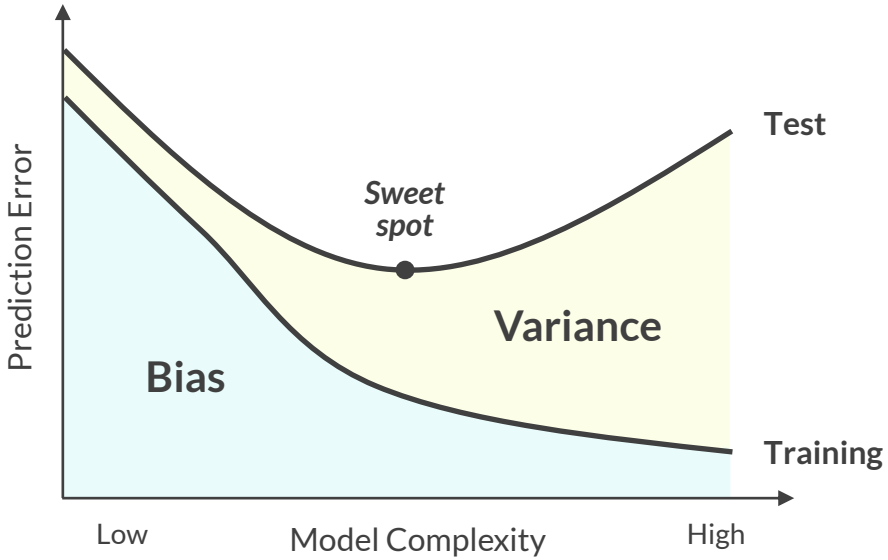
- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

The **bias-variance tradeoff** aims to find a balance between the two types of errors

- It's rare for a model to have low bias *and* low variance, so finding a "sweet spot" is key
- This is something that you can monitor during the model tuning process

High bias models

- Fail to capture trends in the data
- Are underfit
- Are too simple
- Have a high error rate on both train & test data



High variance models

- Capture noise from the training data
- Are overfit
- Are too complex
- Have a large gap between training & test error

VALIDATION DATA

Validation data is a subset of the training data set that is used to assess model fit and provide feedback to the modeling process

- This is an extension of a simple train-test split of the data

Train-Test Split

*With a simple train / test split, you **cannot use test data to optimize** your models*

Train-Validation-Test Split

With separate data sets for validation and testing, the validation data can and should be used to:

- *Check for overfitting*
- *Fine tune model parameters*
- *Verify the impact of specific features on out-of-sample data*
- *Verify the impact of outliers on out-of-sample data*

VALIDATION DATA

You can also use the **train_test_split** function to create the validation data set

- First split off the test data, then separate the training and validation sets

```
from sklearn.model_selection import train_test_split

# Test Split
X, X_test, y, y_test = train_test_split(X, y, test_size=.2, random_state=12345)

# Validation Set (.25 of .8 = .2)
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
                                                      test_size=.25,
                                                      random_state=12345
                                                     )

print(
    f"Training Set Rows: {X_train.shape[0]}",
    f"Validation Set Rows: {X_valid.shape[0]}",
    f"Test Set Rows: {X_test.shape[0]}",
)
```

First split off the test data

Then do the same thing to split off the validation data

The test size is 25% of the training data (80%), which is 20% of the full data set

```
Training Set Rows: 600 Validation Set Rows: 200 Test Set Rows: 200
```

Perfect 60/20/20 split!

MODEL TUNING

Model tuning is the process of making gradual improvements to a model by fitting it on the training data and scoring it on the validation data

- This lets you compare modifications to your models *(like adding features)* and avoid overfitting

Model #1

3 variables

- Training R^2 = **0.67**
- Validation R^2 = **0.66**

Both R^2 values seem relatively low with a small gap between them
(underfit – high bias, low variance)

Action: Add complexity

Model #2

20 variables

- Training R^2 = **0.89**
- Validation R^2 = **0.71**

Training R^2 improved but the gap between them grew considerably
(overfit – low bias, high variance)

Action: Simplify

Model #3

3 variables + 1 squared term

- Training R^2 = **0.81**
- Validation R^2 = **0.79**

Training R^2 dropped a bit but the gap between them closed down
(best fit – balanced bias & variance)

Action: Decide on model

MODEL TUNING

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

To tune your model in Python:

1. Use **sm.OLS(y_train, X_train).fit()** to fit the model on the training data
2. Use **r2(y_train, model.predict(X_train))** to return R^2 for the training data *(or use mae / mse)*
3. Use **r2(y_valid, model.predict(X_valid))** to return R^2 for the validation data *(or use mae / mse)*

```
from sklearn.metrics import r2_score as r2

model = sm.OLS(y_train, X_train).fit()

print(f"Training R2: {r2(y_train, model.predict(X_train))}")
print(f"Validation R2: {r2(y_valid, model.predict(X_valid))}")
```

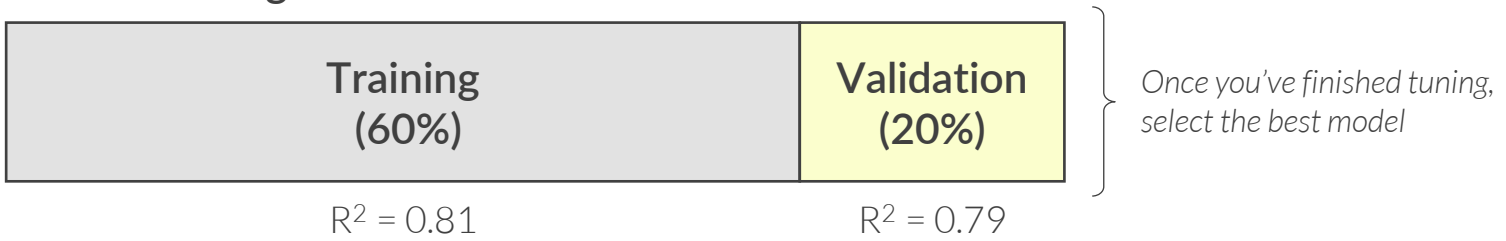
```
Training R2: 0.856811264298398
Validation R2: 0.8262889176864605
```

MODEL SCORING

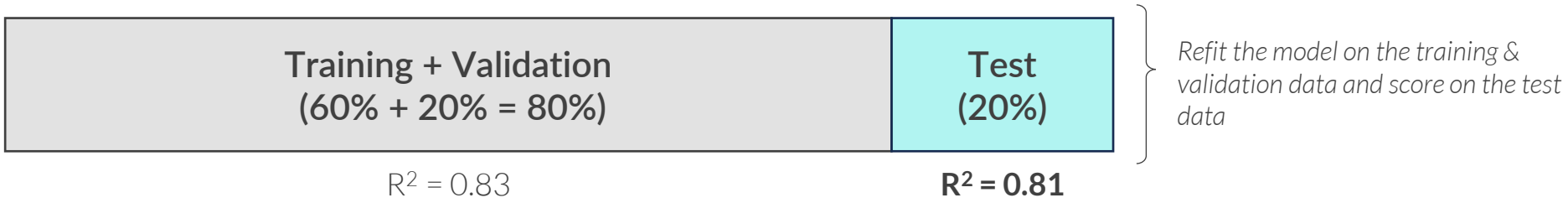
Once you've tuned and selected the best model, you need to refit the model on both the training and validation data, then **score the model** on the test data

- Combining the validation data back with the training data helps improve coefficient estimates
- The final "model score" that you'd share is the score from the test data

Model Tuning



Model Scoring



MODEL SCORING

- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

To score your model in Python:

1. Use **sm.OLS(y, X).fit()** to fit the final model on the training and validation data
2. Use **r2(y, model.predict(X))** to return R^2 for the training and validation data *(or use mae / mse)*
3. Use **r2(y_test, model.predict(X_test))** to return R^2 for the test data *(or use mae / mse)*

```
# Final Test

model = sm.OLS(y, X).fit()

print(f"Training R2: {r2(y, model.predict(X))}")
print(f"Test R2: {r2(y_test, model.predict(X_test))}")
```

```
Training R2: 0.8496755718733405
Test R2: 0.8270499414189432
```

CROSS VALIDATION

Cross validation is another validation method that splits, or "folds", the training data into "k" parts, and treats each part as the validation data across iterations

- You fit the model k times on the training folds, while validating on a different fold each time

Train-Test Split

5-Fold Cross Validation

Fold 1	*Fold 2*	*Fold 3*	*Fold 4*	*Fold 5*	*Validation Score*
Training	Training	Training	Training	Validation	0.42
Training	Training	Training	Validation	Training	0.40
Training	Training	Validation	Training	Training	0.45
Training	Validation	Training	Training	Training	0.41
Validation	Training	Training	Training	Training	0.37

Cross validation score:
0 .41

CROSS VALIDATION

You use cross validation in Python with scikit-learn's **Kfold** function

- sklearn.model_selection.Kfold(n_splits, shuffle=True, random_state)

```
from sklearn.model_selection import KFold
from sklearn.metrics import r2_score as r2


kf = KFold(n_splits=5, shuffle=True, random_state=2023)

# Create a list to store validation scores for each fold
cv_lm_r2s = []

# Loop through each fold in X and y
for train_ind, val_ind in kf.split(X, y):
    # Subset data based on CV folds
    X_train, y_train = X.iloc[train_ind], y.iloc[train_ind]
    X_val, y_val = X.iloc[val_ind], y.iloc[val_ind]
    # Fit the Model on fold's training data
    model = sm.OLS(y_train, X_train).fit()
    # Append Validation score to list
    cv_lm_r2s.append(r2(y_val, model.predict(X_val),))

print("All Validation Scores: ", [round(x, 3) for x in cv_lm_r2s])
print(f"Cross Val Score: {round(np.mean(cv_lm_r2s), 3)} +- {round(np.std(cv_lm_r2s), 3)}")
```

This can be put into a function to reuse!

```
All Validation Scores:  [0.739, 0.756, 0.756, 0.775, 0.716]
Cross Val Score: 0.748 +- 0.02
```

Average validation score, plus/minus the standard deviation

SIMPLE VS CROSS VALIDATION

You should choose one validation approach to use throughout your modeling process, so it's important to highlight the pros and cons of each:

Factor	Simple Validation	Cross Validation
Reliability	**Less reliable**, particularly on smaller data sets	**More reliable**, as the validation process looks at multiple holdout sets
Data Size	More appropriate for **large data sets**, as there is less risk for the holdout set to be biased	More appropriate for **small-medium sized datasets** (<10,000 rows) or (<1M rows) if you have access to sizeable computing power
Training Time	**Faster**, since we're only training and scoring the model once	**Slower**, since we train and score the model once for each fold

ASSIGNMENT: MODEL TESTING & VALIDATION

1 **NEW MESSAGE**

July 12, 2023

From: **Cam Correlation** *(Sr. Data Scientist)*

Subject: **Data Splitting**

Hi there!

The model seems to be shaping up nicely, but I forgot to mention we need to split off a test dataset to assess performance.

Can you split off a test set, and fit your model using cross-validation?

Thanks!

05_data_splitting_assignments.ipynb

Reply Forward

Key Objectives

1. Split your data into training and test
2. Use cross validation to fit your model and report each validation fold score
3. Fit your model on all of your training data and score it on the test dataset

KEY TAKEAWAYS

Data Splitting is a key step in the modeling process

- *Training, Validation, and Test datasets all have unique purposes*

Test Data Sets give us an unbiased estimate of model accuracy

- *Scoring on your test dataset should only be performed once you are ready to decide on a final model*

Validation Data provides critical feedback into model tuning

- *Simple validation is suitable for larger datasets, while cross validation tends to be more reliable on small-medium sized data*
- *Fold your validation data back into training data to fit your final model*

FEATURE ENGINEERING

FEATURE ENGINEERING

In this section we'll cover **feature engineering** techniques for regression models, including dummy variables, interaction terms, binning, and more

TOPICS WE'LL COVER:

- Feature Engineering
- Math Calculations
- Category Mappings

GOALS FOR THIS SECTION:

- Understand the goal and importance of feature engineering in modeling
- Learn common types of feature engineering and how to implement them using Python
- Walk through examples of some more advanced feature engineering options

FEATURE ENGINEERING

Feature engineering is the process of creating or modifying features that you think will be helpful inputs for improving a model's performance

- This is where data scientists add the most value in the modeling process
- Strong domain knowledge is required to get the most out of this step

Feature Engineering

Math Calculations

Category Mappings

	carat	clarity	cut	color	price
0	0.60	SI1	Fair	G	1305
1	2.00	SI2	Very Good	I	12221
2	0.34	VS2	Premium	G	596
3	0.36	SI2	Premium	E	605
4	0.45	VS1	Very Good	F	1179

```
Cross Val R2: 0.849 +- 0.002
```

Only "carat" can be used as a feature here, since the rest or not numeric

	carat	carat_sq	I1_clarity	good_cut	very_good_cut	ideal_cut	premium_cut	colorless	absolutely_colorless	price
0	0.60	0.3600	0	0	0	0	0	0	0	1305
1	2.00	4.0000	0	0	1	0	0	0	0	12221
2	0.34	0.1156	0	0	0	0	1	0	0	596
3	0.36	0.1296	0	0	0	0	1	0	1	605
4	0.45	0.2025	0	0	1	0	0	0	1	1179

```
Cross Val R2: 0.943 +- 0.003
```

Feature engineering is all about **trial and error**, and you can use cross validation to test new ideas and determine if they improve the model

FEATURE ENGINEERING TECHNIQUES

Feature Engineering

Math Calculations

Category Mappings

These are some commonly used **feature engineering techniques**:

Math Calculations

- Polynomial terms
- Combining features
- Interaction terms

Category Mappings

- Binary columns
- Dummy variables
- Binning

We'll do an in-depth review of these

DateTime Calculations

- Days from "today"
- Time between dates

Group Calculations

- Aggregations
- Ranks within groups

Scaling

- Standardization
- Normalization

We'll briefly demo these techniques (they reference previously learned concepts)

Your own creativity, domain knowledge, and critical thinking will lead to feature engineering ideas not covered in this course, but these are worth keeping in mind!

POLYNOMIAL TERMS

Feature Engineering

Math Calculations

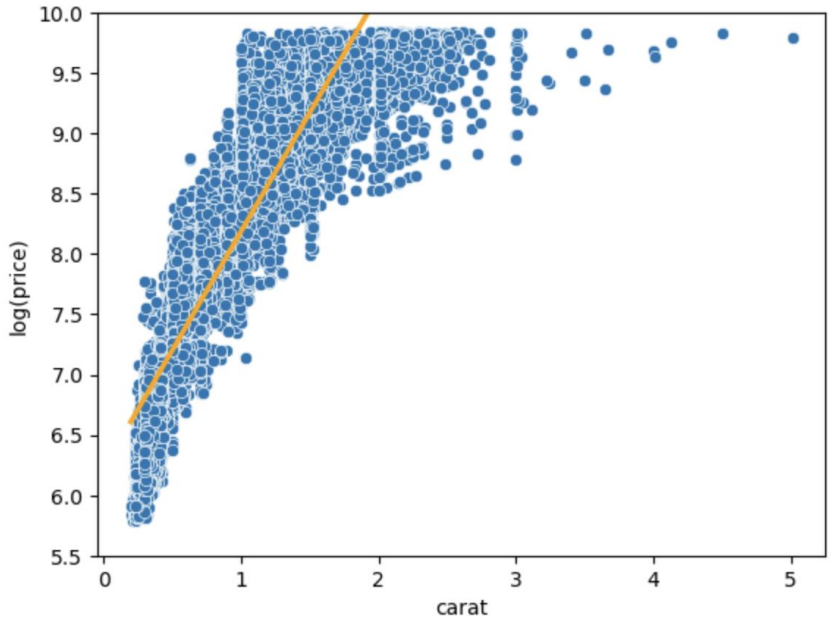
Category Mappings

Adding **polynomial terms** (x^2, x^3, etc.) to regression models can improve fit when spotting "curved" feature-target relationships during EDA

- Generally, as the degree of your polynomial increases, so does the risk of overfitting
- If your goal is prediction, let cross validation guide the complexity of your polynomial term

Model Structure	Cross-Val R^2
Carat	**.847**
Carat + Carat^2	.929
Carat + Carat^2 + Carat^3	.936
Carat + … + Carat^4	.936
Carat + … + Carat^5	.936

Always keep the lower order terms in the model



POLYNOMIAL TERMS

Feature Engineering

Math Calculations

Category Mappings

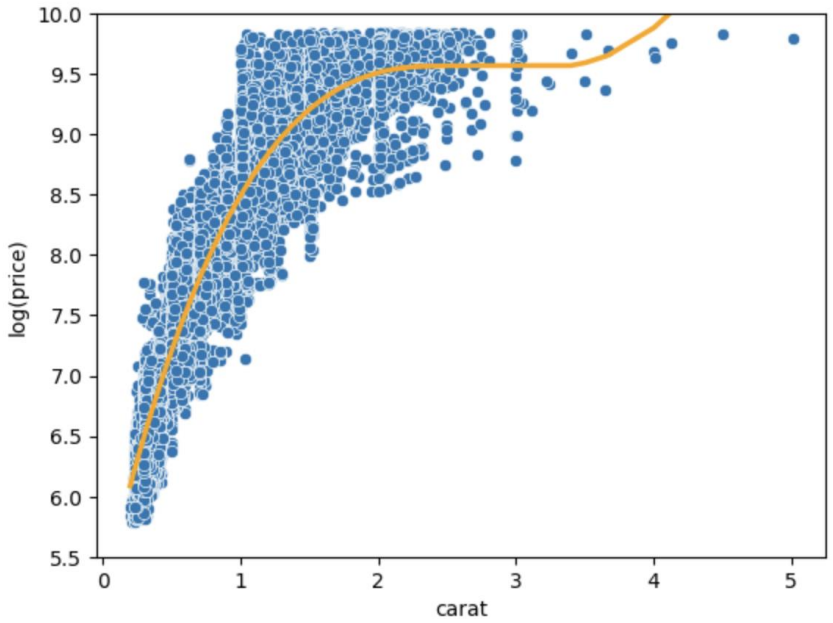
Adding **polynomial terms** (x^2, x^3, etc.) to regression models can improve fit when spotting "curved" feature-target relationships during EDA

- Generally, as the degree of your polynomial increases, so does the risk of overfitting
- If your goal is prediction, let cross validation guide the complexity of your polynomial term

Model Structure	Cross-Val R^2
Carat	.847
Carat + Carat^2	.929
Carat + Carat^2 + Carat^3	**.936**
Carat + … + Carat^4	.936
Carat + … + Carat^5	.936

Cross validation R^2 stops increasing after adding the cubic term, so we should stop there

(if you need to explain this model, you could justify stopping at the squared term for interpretability)



COMBINING FEATURES

Feature Engineering

Math Calculations

Category Mappings

Combining features with arithmetic operators can help avoid multicollinearity issues without throwing away information altogether

- Sums (+), differences (-), products (*) and ratios (÷) of columns are all valid combinations

Current best model:

Model Structure	Cross-Val R^2
Carat + $Carat^2$ + $Carat^3$	.936

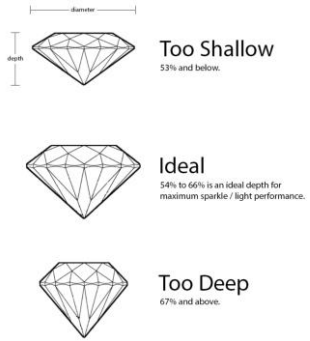
Carat weight is our strongest variable alone, but can we engineer some stronger features?

Feature correlations:

	x	y	z	carat	price
x	1.000000	0.974701	0.970771	0.975093	0.884433
y	0.974701	1.000000	0.952005	0.951721	0.865419
z	0.970771	0.952005	1.000000	0.953387	0.861249
carat	0.975093	0.951721	0.953387	1.000000	0.921591
price	0.884433	0.865419	0.861249	0.921591	1.000000

Carat (weight), x (width), y (length), and z (depth) capture a diamond's size, which is why they are highly correlated with each other and will cause multicollinearity issues in the model

While carat is the most important factor in price, "deep" diamonds are less valuable than "wide" diamonds, since depth can't be seen in jewelry



COMBINING FEATURES

Feature Engineering

Math Calculations

Category Mappings

Combining features with arithmetic operators can help avoid multicollinearity issues without throwing away information altogether

- Sums (+), differences (-), products (*) and ratios (÷) of columns are all valid combinations

Current best model:

Model Structure	Cross-Val R^2
Carat + $Carat^2$ + $Carat^3$	.936

The carat model is still the best!

Models with combined features:

Model Structure	Cross-Val R^2
Original columns: x, y, z	.913
Sum: (x + y + z)	.909
Product: (x * y * z)	.909
Area/depth ratio: (x + y) / z	.877

PRO TIP: There is no guarantee that even the most clever feature engineering will improve your model; give it a try, but be prepared to move on if you don't see results!

INTERACTION TERMS

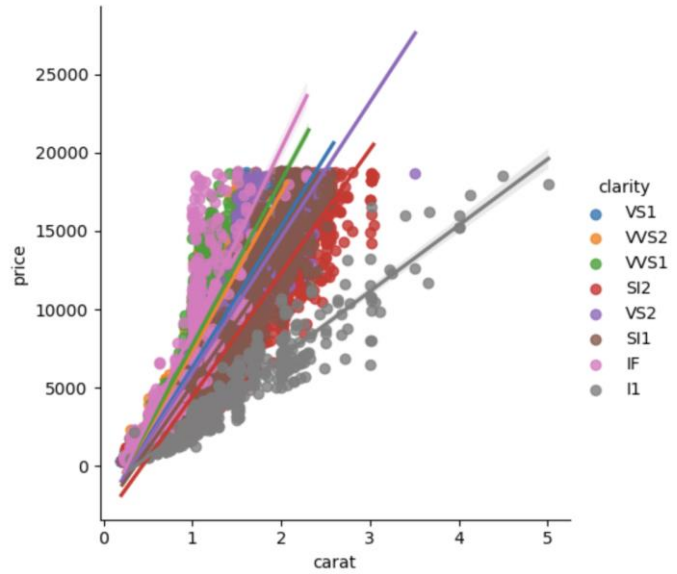
Feature Engineering

Math Calculations

Category Mappings

Interaction terms capture feature-target relationships that change based on the value of another feature

- They can be detected with careful EDA or brute force engineering
- They exist for both categorical-numeric and numeric-numeric feature combinations



"I1" diamonds have a much lower slope coefficient than others

Adding an interaction term:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$$

Interaction term

$$y = \beta_0 + \beta_1 (carat) + \beta_2 (clarity\_I1) + \beta_3 (carat * clarity\_I1)$$

When adding interaction terms, always include the original features in your model

INTERACTION TERMS

Feature Engineering

Math Calculations

Category Mappings

Interaction terms capture feature-target relationships that change based on the value of another feature

- They can be detected with careful EDA or brute force engineering
- They exist for both categorical-numeric and numeric-numeric feature combinations

Current best model:

Model Structure	Cross-Val R^2
Carat + $Carat^2$ + $Carat^3$	.936

The carat model is still the best!

Model with interaction term:

Model Structure	Cross-Val R^2
Carat + I1 + Carat * I1	.870

PRO TIP: Interaction terms are cool, but often don't add enough value to justify the time it takes to find them or the increased complexity!

CATEGORICAL FEATURES

Categorical features must be converted to numeric before modeling

- In their simplest form, they can be represented with binary columns

Feature Engineering

Math Calculations

Category Mappings

```
diamonds = diamonds.assign(
    clarity_IF = np.where(diamonds["clarity"]=="IF", 1, 0)
)

diamonds.iloc[330:335]
```

This assigns a value of 1 to diamonds with "IF" clarity and a value of 0 to any others

	carat	cut	color	clarity	depth	table	price	x	y	z	clarity_IF
330	0.31	Ideal	G	IF	61.5	54.0	871	4.40	4.41	2.71	1
331	0.53	Ideal	F	IF	61.9	54.0	2802	5.22	5.25	3.24	1
332	0.71	Ideal	D	VVS2	61.6	56.0	4222	5.69	5.77	3.53	0
333	1.16	Ideal	G	SI2	61.5	54.0	5301	6.78	6.75	4.16	0
334	0.61	Very Good	I	IF	60.2	57.0	2057	5.49	5.58	3.33	1

This field that represents clarity is now numeric and can be input into a model

CATEGORICAL FEATURES

Feature Engineering

Math Calculations

Category Mappings

Categorical features must be converted to numeric before modeling

- In their simplest form, they can be represented with binary columns

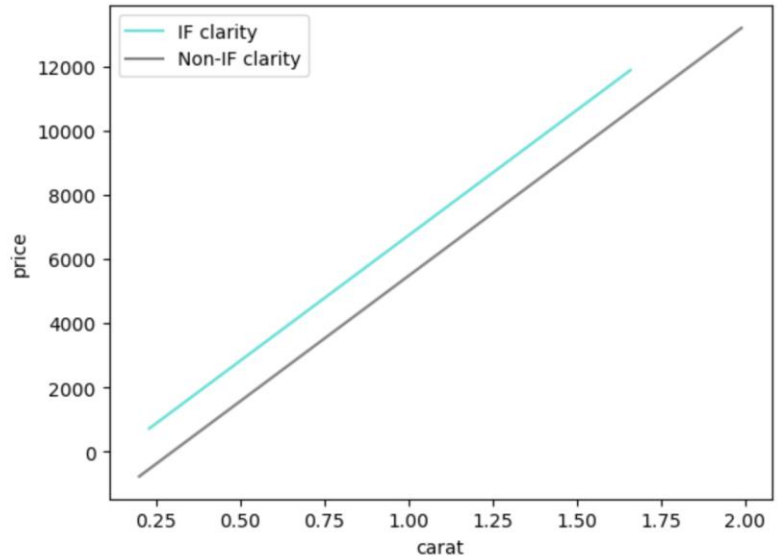
Interpreting coefficients for binary columns:

```
X = sm.add_constant(diamonds[["carat", "clarity_IF"]])
y = diamonds["price"]

model = sm.OLS(y, X).fit()

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2341.7610	13.159	-177.963	0.000	-2367.552	-2315.970
carat	7810.9754	14.010	557.532	0.000	7783.516	7838.435
clarity_IF	1261.0975	37.075	34.015	0.000	1188.431	1333.764

Diamonds with a clarity of "IF" have a price $1,278 dollars higher, on average, than non-IF diamonds

Effect on a model:



Binary columns have the effect of shifting the intercept of the line without affecting its slope!

DUMMY VARIABLES

Feature Engineering

Math Calculations

Category Mappings

A **dummy variable** is a field that only contains zeros and ones to represent the presence (1) or absence (0) of a value, also known as one-hot encoding

- They are used to transform a categorical field into multiple numeric fields
- Use **pd.get_dummies()** to create dummy variables in Python

```
diamonds[["carat", "clarity"]].head()
```

	carat	clarity
0	0.52	VVS1
1	1.50	VS2
2	0.91	SI2
3	1.04	SI1
4	0.30	SI1

```
pd.get_dummies(diamonds[["carat", 'clarity']])
```

	carat	clarity_I1	clarity_IF	clarity_SI1	clarity_SI2	clarity_VS1	clarity_VS2	clarity_VVS1	clarity_VVS2
0	0.52	0	0	0	0	0	0	1	0
1	1.50	0	0	0	0	0	1	0	0
2	0.91	0	0	0	1	0	0	0	0
3	1.04	0	0	1	0	0	0	0	0
4	0.30	0	0	1	0	0	0	0	0

*These dummy variables are **numeric representations** of the "clarity" field*

DUMMY VARIABLES

Feature Engineering

Math Calculations

Category Mappings

In linear regression models, you need to **drop a dummy variable** category using the "drop_first=True" argument to avoid perfect multicollinearity

- The category that gets dropped is known as the "reference level"

```
diamonds[["carat", "clarity"]].head()
```

	carat	clarity
0	0.52	VVS1
1	1.50	VS2
2	0.91	SI2
3	1.04	SI1
4	0.30	SI1

```
pd.get_dummies(diamonds[["carat", "clarity"]], drop_first=True)
```

	carat	clarity_IF	clarity_SI1	clarity_SI2	clarity_VS1	clarity_VS2	clarity_VVS1	clarity_VVS2
0	0.52	0	0	0	0	0	1	0
1	1.50	0	0	0	0	1	0	0
2	0.91	0	0	1	0	0	0	0
3	1.04	0	1	0	0	0	0	0
4	0.30	0	1	0	0	0	0	0

PRO TIP: Your model accuracy will be the same regardless of which dummy column dropped, but some reference levels are more intuitive to interpret than others

If you want to choose your reference level, skip the drop_first argument and drop the desired reference level manually

DUMMY VARIABLES

Feature Engineering

Math Calculations

Category Mappings

In linear regression models, you need to **drop a dummy variable** category using the "drop_first=True" argument to avoid perfect multicollinearity

- The category that gets dropped is known as the "reference level"

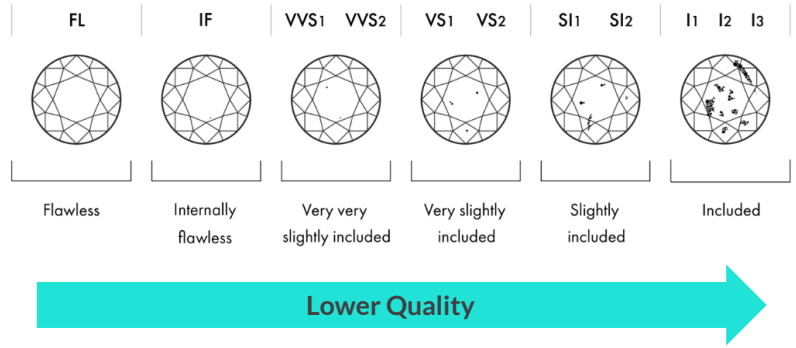
Interpreting coefficients for dummy variables:

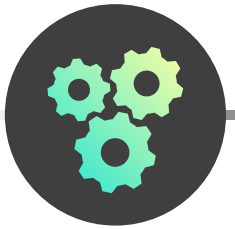
	coef	std err	t	P>|t|	[0.025	0.975]
const	-6959.7458	56.310	-123.596	0.000	-7070.115	-6849.376
carat	8436.8446	14.191	594.519	0.000	8409.030	8464.659
clarity_IF	5571.4246	64.005	87.047	0.000	5445.973	5696.876
clarity_SI1	3783.4198	55.091	68.676	0.000	3675.440	3891.399
clarity_SI2	2925.0475	55.439	52.761	0.000	2816.385	3033.710
clarity_VS1	4668.3822	56.206	83.058	0.000	4558.217	4778.547
clarity_VS2	4441.5674	55.344	80.254	0.000	4333.093	4550.042
clarity_VVS1	5226.2087	59.423	87.949	0.000	5109.738	5342.679
clarity_VVS2	5212.6809	57.817	90.158	0.000	5099.358	5326.004

The reference level is represented in the intercept term, and the coefficients of other categories are compared to it

Diamonds with a clarity of "IF" have a predicted price of $5,571 higher than our reference level (diamonds with a clarity of I1)

The coefficients align with the diamond quality chart!





BINNING CATEGORICAL DATA

Feature Engineering

Math Calculations

Category Mappings

Adding dummy variables for each categorical column in your data can lead to very wide data sets, which tends to increase model variance

Grouping, or **binning categorical data**, solves this and can improve interpretability

- After binning, create dummy variables for the groups *(which should be fewer than before)*

```
diamonds.sample(1000)["clarity"].value_counts()
```

```
SI1     242
VS2     218
SI2     176
VS1     153
VVS2     85
VVS1     73
IF       40
I1       13
Name: clarity, dtype: int64
```

If we had less data, the "I1" category would be too rare to produce reliable estimates, as random data splitting means we might not see "I1" diamonds in our test or validation sets!

(categories with low counts are especially at risk of overfitting)

```
mapping_dict = {
    "IF": "Great",
    "VVS1": "Great",
    "VVS2": "Great",
    "VS1": "Average",
    "VS2": "Average",
    "SI1": "Below Average",
    "SI2": "Below Average",
    "I1": "Below Average"
}

diamonds = diamonds.assign(clarity_reduced = diamonds["clarity"].map(mapping_dict))
```

We can map the 8 clarity values into just 3 buckets

```
diamonds["clarity_reduced"].value_counts()
```

```
Below Average    418
Average          371
Great            198
Name: clarity_reduced, dtype: int64
```

BINNING CATEGORICAL DATA

Feature Engineering

Math Calculations

Category Mappings

Adding dummy variables for each categorical column in your data can lead to very wide data sets, which can increase model variance with small data sets

Grouping, or **binning categorical data**, solves this and improves interpretability

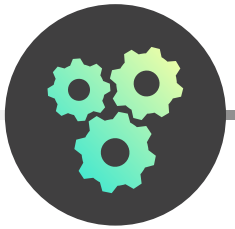
- After binning, create dummy variables for the groups *(which should be fewer than before)*

Dummy variables:

	IF	SI1	SI2	VS1	VS2	VVS1	VVS2
0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0
2	0	0	0	0	1	0	0
3	0	0	1	0	0	0	0
4	0	0	0	1	0	0	0

Binning:

	Below Average	Great
0	1	0
1	1	0
2	0	0
3	1	0
4	0	0



BINNING NUMERIC DATA

Binning numeric data lets you turn numeric features into categories

- Generally, this is less accurate than using raw values, but it is a highly interpretable way of capturing non-linear trends and numeric fields with a high percentage of missing values

Feature Engineering

Math Calculations

Category Mappings

```
diamonds = diamonds.assign(
   carat_bins = pd.cut(
        diamonds["carat"],
        bins=[0, .5, 1, 1.5, 2, 2.5, 3, 4, 6],
        labels=["0-.5", ".5 -1", "1-1.5", "1.5-2", "2-2.5", "2.5-3", "3-4", "4+"]
    )
)

diamonds[["carat", "carat_bins"]].head()
```

Carat is a continuous field, but we're binning it into values at various intervals, making it a categorical field

	carat	carat_bins
0	0.71	.5 -1
1	0.30	0-.5
2	1.01	1-1.5
3	0.24	0-.5
4	0.52	.5 -1

Now you can get dummy variables from this!

BINNING NUMERIC DATA

Binning numeric data lets you turn numeric features into categories

- Generally, this is less accurate than using raw values, but it is a highly interpretable way of capturing non-linear trends and numeric fields with a high percentage of missing values

Feature Engineering

Math Calculations

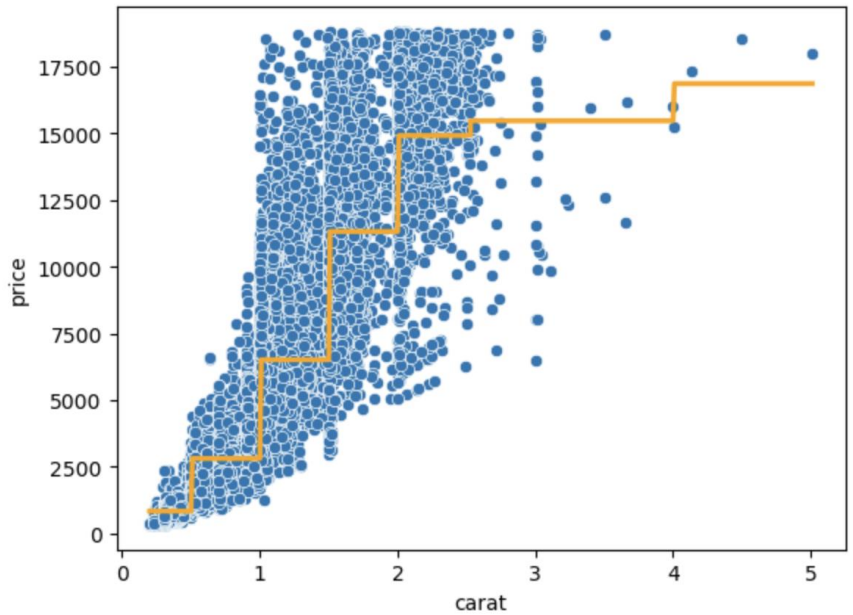
Category Mappings

Current best model:

Model Structure	Cross-Val R^2
Carat + $Carat^2$ + $Carat^3$	.936

Models with binned data:

Model Structure	Cross-Val R^2
Carat	.847
Carat Bins	**.870**



Note that the binned data has created steps in the model versus the earlier smooth lines / curves

ASSIGNMENT: FEATURE ENGINEERING

NEW MESSAGE

July 14, 2023

From: **Cam Correlation** *(Sr. Data Scientist)*

Subject: **Feature Engineering**

Hello again!

We're starting to get somewhere with this model, nice job!

I notice that your last model didn't have any of the categorical features in it, can you make sure to include them?

I've also included some additional feature engineering ideas I had in the notebook.

Thanks!

06_feature_engineering_assignment.ipynb

Reply Forward

Key Objectives

1. Perform feature engineering on numeric and categorical features
2. Evaluate model performance after including the new features
3. Select only features that improve model fit

KEY TAKEAWAYS

- **Feature engineering** lets you turn raw data into useful model features
  - *This is critical to getting the best accuracy out of your data sets*
- Several calculations can be applied to **enhance numeric features**
  - *Combining features, polynomial terms, interaction terms, and binning can be applied on numeric variables*
- It also allows you to **prepare categorical features** for modeling
  - *Techniques like binary columns, dummy variables, and binning let you turn categorical variables into numeric*
- Most ideas will come from **domain expertise** and **critical thinking**
  - *Thinking carefully about what might influence your target variable can lead to the creation of powerful features*