DATA SCIENCE IN PYTHON

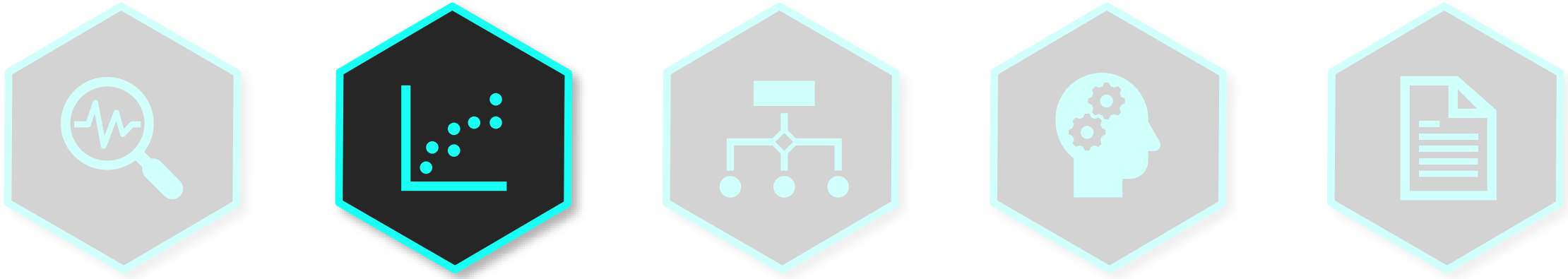
Regression

★★★★★ *With Expert Python Instructor Chris Bruehl*

MAVEN ANALYTICS

ABOUT THIS SERIES

This is **Part 2** of a **5-Part series** designed to take you through several applications of data science using Python, including **data prep & EDA**, **regression**, **classification**, **unsupervised learning** & **NLP**



PART 1
Data Prep & EDA

PART 2
Regression

PART 3
Classification

PART 4
Unsupervised Learning

PART 5
Natural Language Processing

COURSE STRUCTURE

This is a **project-based** course for students looking for a practical, hands-on approach to learning data science and applying regression models with Python

Additional resources include:

- **Downloadable PDF** to serve as a helpful reference when you're offline or on the go
- **Quizzes** & **Assignments** to test and reinforce key concepts, with step-by-step solutions
- **Interactive demos** to keep you engaged and apply your skills throughout the course

COURSE OUTLINE

1	**Intro to Data Science**	*Introduce the fields of data science and machine learning, review essential skills, and introduce each phase of the data science workflow*
2	**Regression 101**	*Review the basics of regression, including key terms, the types and goals of regression analysis, and the regression modeling workflow*
3	**Pre-modeling Data Prep & EDA**	*Recap the data prep & EDA steps required to perform modeling, including key techniques to explore the target, features, and their relationships*
4	**Simple Linear Regression**	*Build simple linear regression models in Python and learn about the metrics and statistical tests that help evaluate their quality and output*
5	**Multiple Linear Regression**	*Build multiple linear regression models in Python and evaluate the model fit, perform variable selection, and compare models using error metrics*

COURSE OUTLINE

6	**Model Assumptions**	*Review the assumptions of linear regression models that need to be met to ensure that the model's predictions and interpretation are valid*
7	**Model Testing & Validation**	*Test model performance by splitting data, tuning the model with the train & validation data, selecting the best model, and scoring it on the test data*
8	**Feature Engineering**	*Apply feature engineering techniques for regression models, including dummy variables, interaction terms, binning, and more*
9	**Regularized Regression**	*Introduce regularized regression techniques, which are alternatives to linear regression, including Ridge, Lasso, and Elastic Net regression*
10	**Time Series Analysis**	*Learn methods for exploring time series data and how to perform time series forecasting using linear regression and Prophet*

WELCOME TO MAVEN CONSULTING GROUP

THE SITUATION

You've just been hired as an Associate Data Scientist for **Maven Consulting Group** to work on a team that specializes in price research for various industries.

THE ASSIGNMENT

You'll have access to data on the price data for several different industries, including diamonds, computer prices, and apartment rent.

Your task is to **build regression models** that can accurately predict the price of goods, while giving their clients insights into the factors that impact pricing.

THE OBJECTIVES

1. **Explore** & visualize the data
2. **Prepare** the data for modelling
3. **Apply regression algorithms** to the data
4. **Evaluate** how well your models fit
5. **Select** the best model and interpret it

MCG

MAVEN CONSULTING GROUP

SETTING EXPECTATIONS

This course covers both the **theory** & **application** of linear regression models

- *We'll start with Ordinary Least Squares (OLS), including evaluation metrics, assumptions, and validation & testing*
- *We'll then pivot to regularized regression models, which are extensions of linear regression with special properties*

We'll also do an overview of **time series analysis** and forecasting methods

- *We'll cover time series behavior, smoothing, decomposition, and basic forecasting models & theory, but won't do a deep dive into advanced time series modeling techniques*

We'll use **Jupyter Notebooks** as our primary coding environment

- *Jupyter Notebooks are free to use, and the industry standard for conducting data analysis with Python (we'll introduce Google Colab as an alternative, cloud-based environment as well)*

You do **NOT** need to be a Python expert to take this course

- *It is strongly recommended that you complete the first course in this series, Data Prep & EDA, and have some basic statistics knowledge, but we will teach the relevant Python code for building regression models from scratch*

INSTALLATION & SETUP

INSTALLATION & SETUP

In this section we'll install Anaconda and introduce **Jupyter Notebook**, a user-friendly coding environment where we'll be coding in Python

TOPICS WE'LL COVER:

- Installing Anaconda
- Launching Jupyter
- Google Colab

GOALS FOR THIS SECTION:

- Install Anaconda and launch Jupyter Notebook
- Get comfortable with the Jupyter Notebook environment and interface

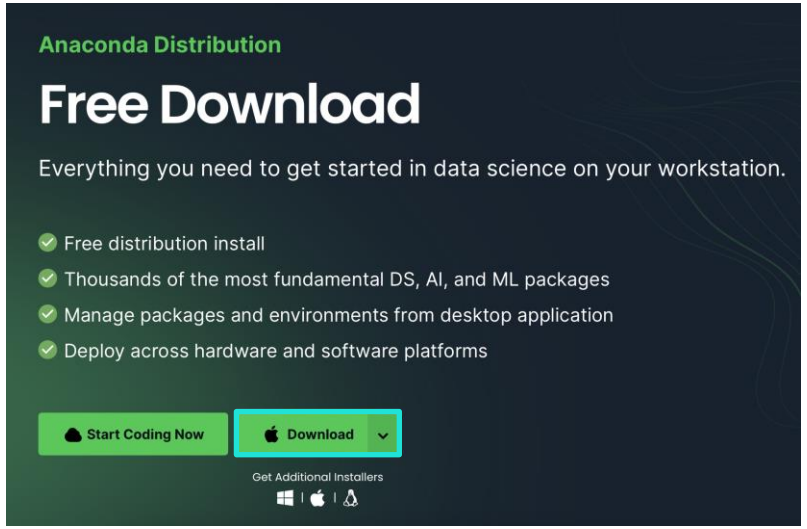
INSTALL ANACONDA (MAC)

Installing Anaconda

Launching Jupyter

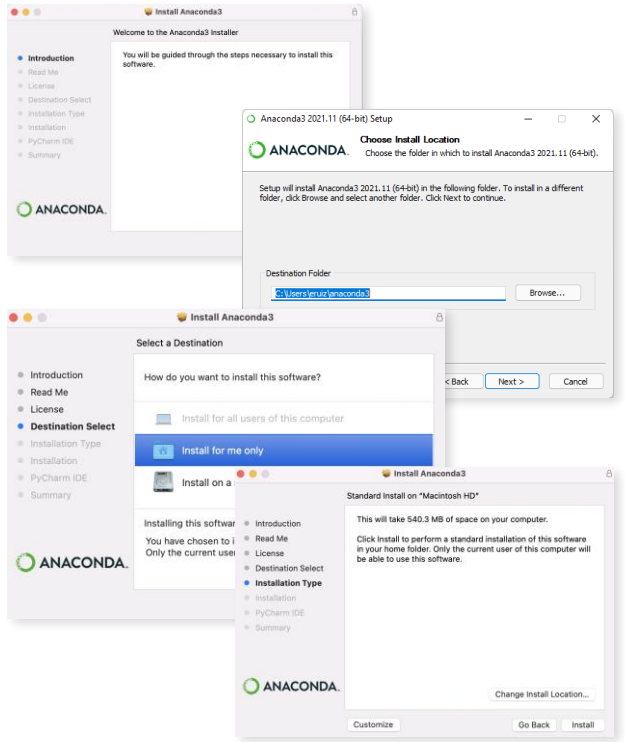
Google Colab

1) Go to **anaconda.com/products/distribution** and click 



2) Launch the downloaded Anaconda **pkg** file

3) Follow the **installation steps**
(default settings are OK)



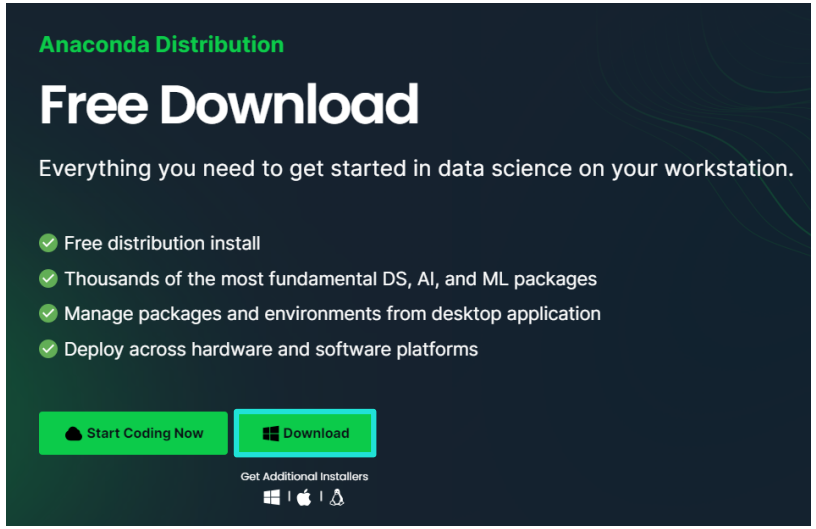
INSTALL ANACONDA (PC)

Installing Anaconda

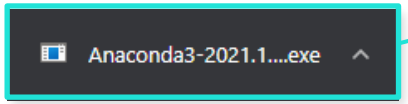
Launching Jupyter

Google Colab

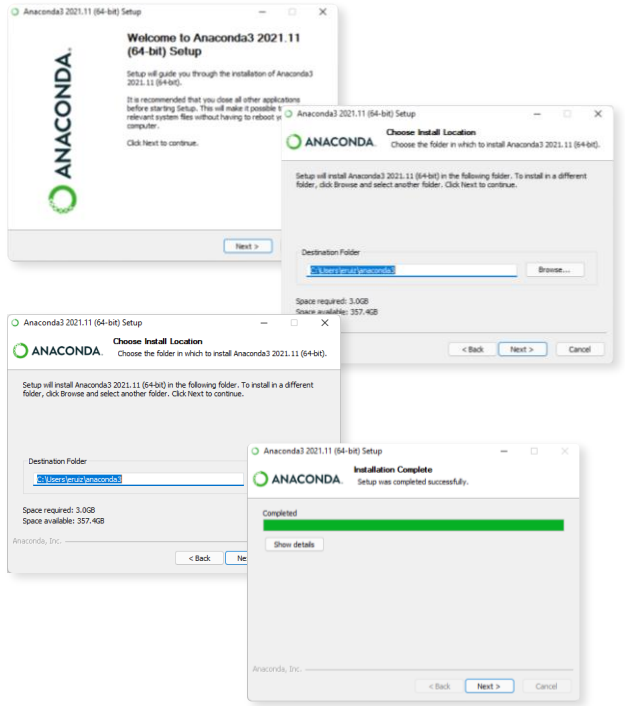
1) Go to **anaconda.com/products/distribution** and click **Download**



2) Launch the downloaded Anaconda **exe** file



3) Follow the **installation steps**
(default settings are OK)



LAUNCHING JUPYTER

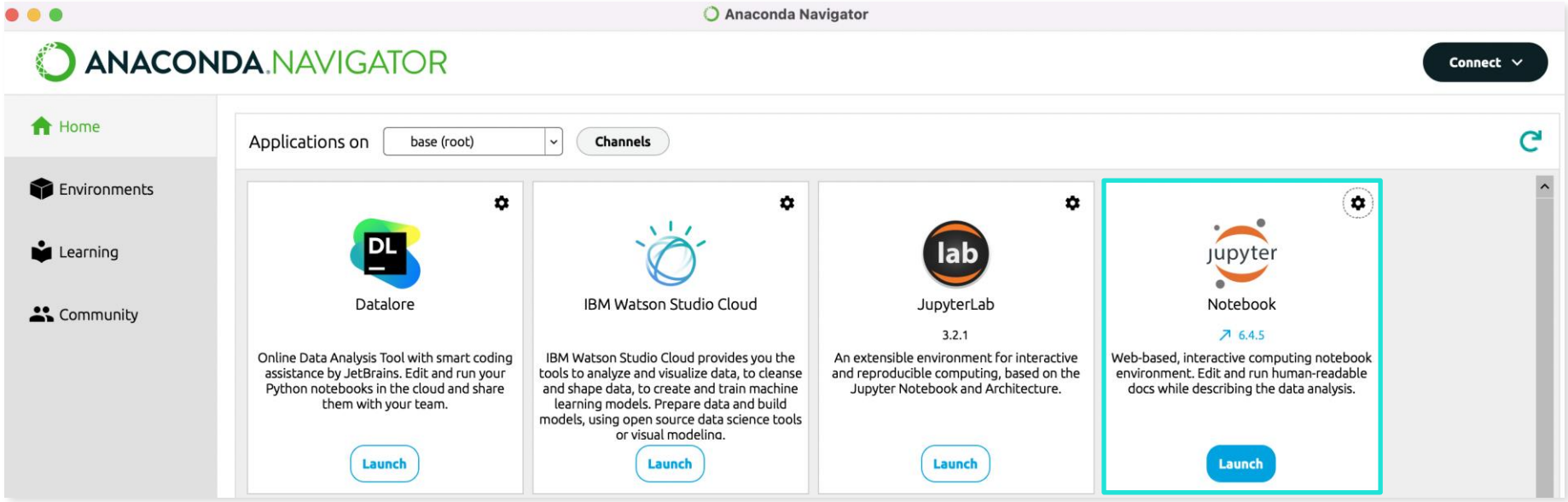
Installing Anaconda

Launching Jupyter

Google Colab

1) Launch **Anaconda Navigator**

2) Find **Jupyter Notebook** and click **Launch**



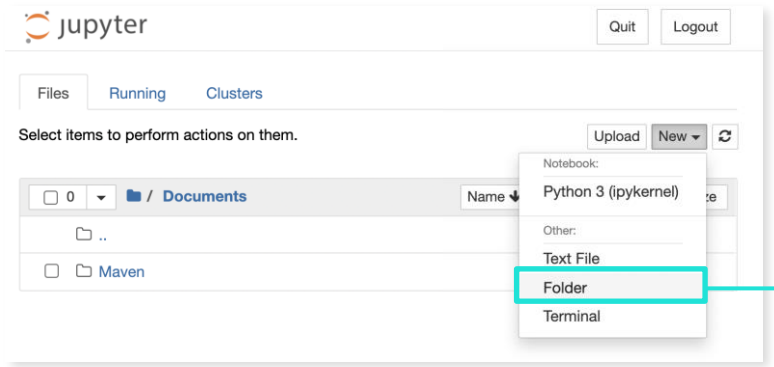
YOUR FIRST JUPYTER NOTEBOOK

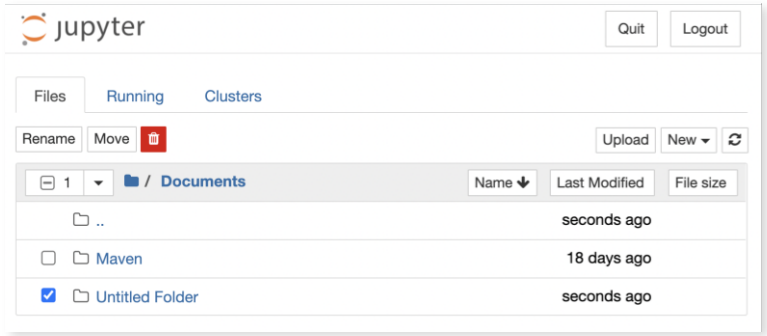
Installing Anaconda

Launching Jupyter

Google Colab

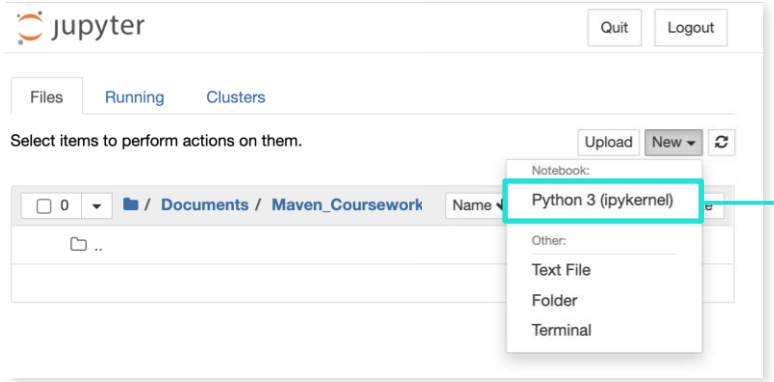
1) Once inside the Jupyter interface, **create a folder** to store your notebooks for the course

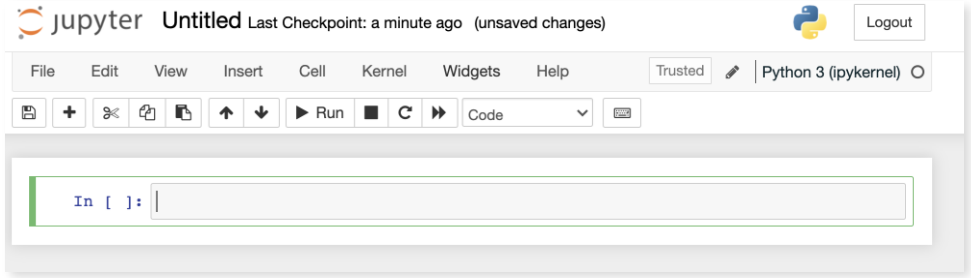




NOTE: *You can rename your folder by clicking "Rename" in the top left corner*

2) Open your new coursework folder and **launch your first Jupyter notebook**!



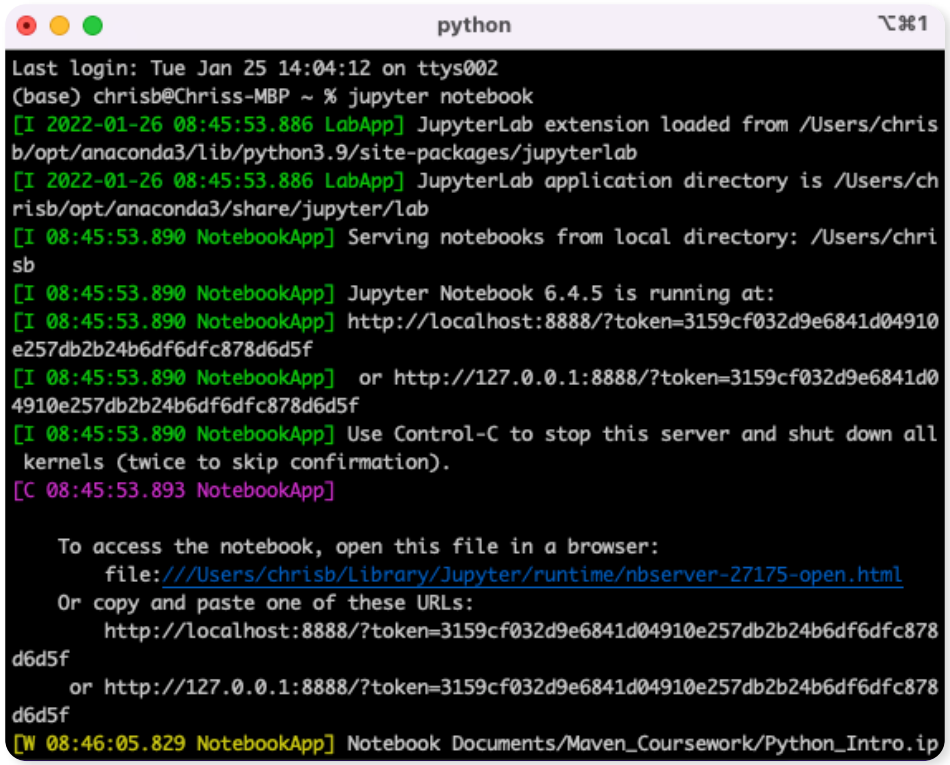


NOTE: *You can rename your notebook by clicking on the title at the top of the screen*

THE NOTEBOOK SERVER

NOTE: When you launch a Jupyter notebook, a terminal window may pop up as well; this is called a **notebook server**, and it powers the notebook interface

- Installing Anaconda
- Launching Jupyter
- Google Colab



If you close the server window, **your notebooks will not run!**

Depending on your OS, and method of launching Jupyter, one may not open – as long as you can run your notebooks, don't worry!

ALTERNATIVE: GOOGLE COLAB

Installing Anaconda

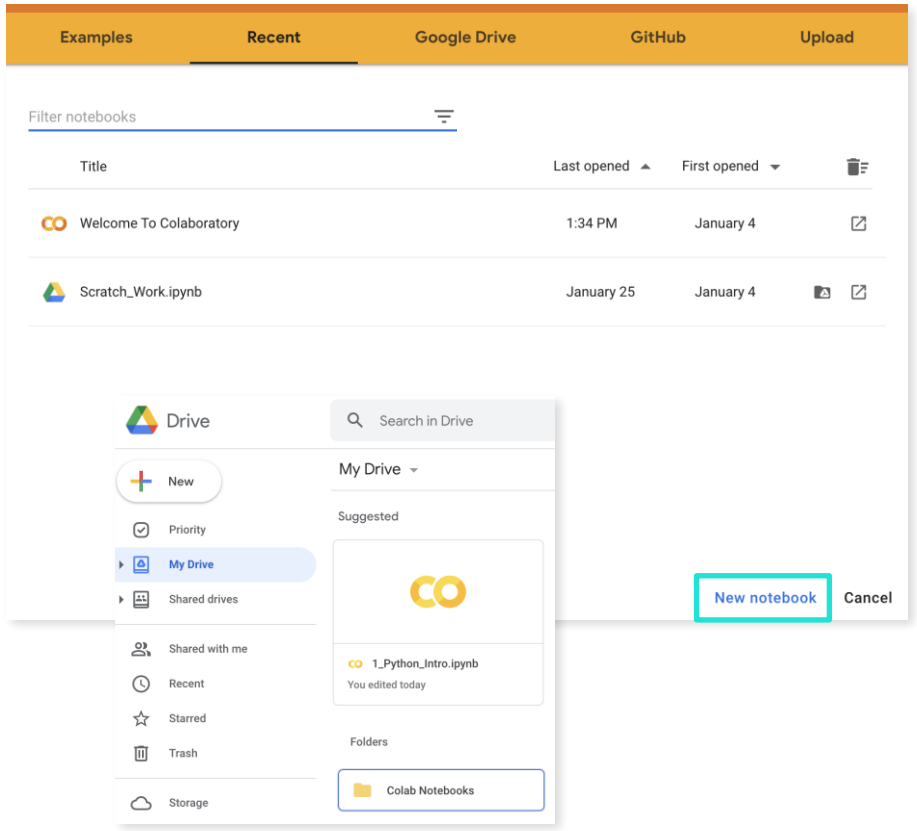
Launching Jupyter

Google Colab

Google Colab is Google's cloud-based version of Jupyter Notebooks

To create a Colab notebook:

1. Log in to a Gmail account
2. Go to **colab.research.google.com**
3. Click "**new notebook**"



Colab is very similar to Jupyter Notebooks *(they even share the same file extension)*; the main difference is that you are connecting to **Google Drive** rather than your machine, so files will be stored in Google's cloud

INTRO TO DATA SCIENCE

INTRO TO DATA SCIENCE

In this section we'll **introduce the field of data science**, discuss how it compares to other data fields, and walk through each phase of the data science workflow

TOPICS WE'LL COVER:

- What is Data Science?
- Essential Skills
- Machine Learning
- Data Science Workflow

GOALS FOR THIS SECTION:

- Compare data science and machine learning with other common data analytics fields
- Introduce supervised and unsupervised learning, and examples of each technique
- Review the machine learning landscape and commonly used algorithms
- Discuss essential skills, and review each phase of the data science workflow

WHAT IS DATA SCIENCE?

Data science is about *using data to make smart decisions*

Wait, isn't that **business intelligence** ?

Yes! The differences lie in the **types of problems** you solve, and **tools and techniques** you use to solve them:

What *happened*?

- Descriptive Analytics
- Data Analysis
- Business Intelligence

What's *going to* happen?

- Predictive Analytics
- Data Mining
- **Data Science**

DATA SCIENCE SKILL SET

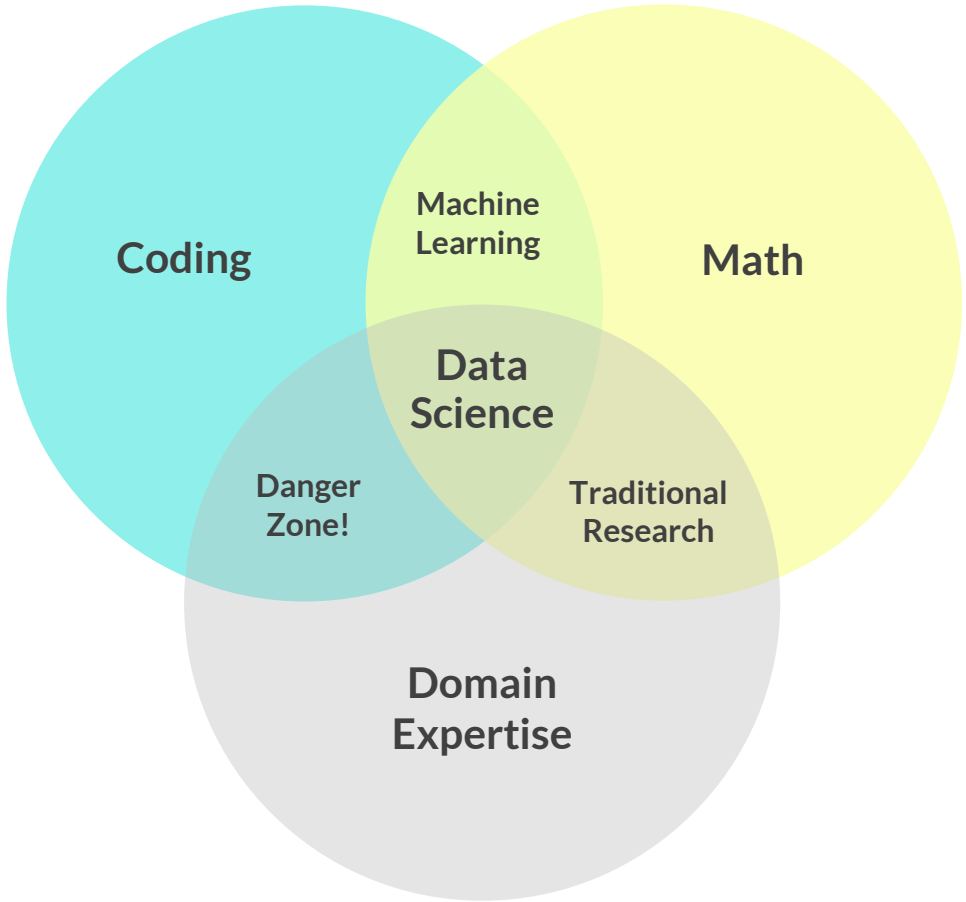
What is Data Science?

Essential Skills

Machine Learning

Data Science Workflow

Data science requires a blend of **coding**, **math**, and **domain expertise**



The key is in applying these along with soft skills like:

- Communication
- Problem solving
- Curiosity & creativity
- Grit
- Googling prowess

Data scientists & analysts approach problem solving in similar ways, but data scientists will often work with larger, more complex data sets and utilize advanced algorithms

WHAT IS MACHINE LEARNING?

Machine learning uses algorithms applied by data scientists to enable computers to learn and make decisions from data

Machine learning algorithms fall into two broad categories:

Supervised Learning

Using historical data to predict the future

What will house prices look like for the next 12 months?

How can I flag suspicious emails as spam?

Unsupervised Learning

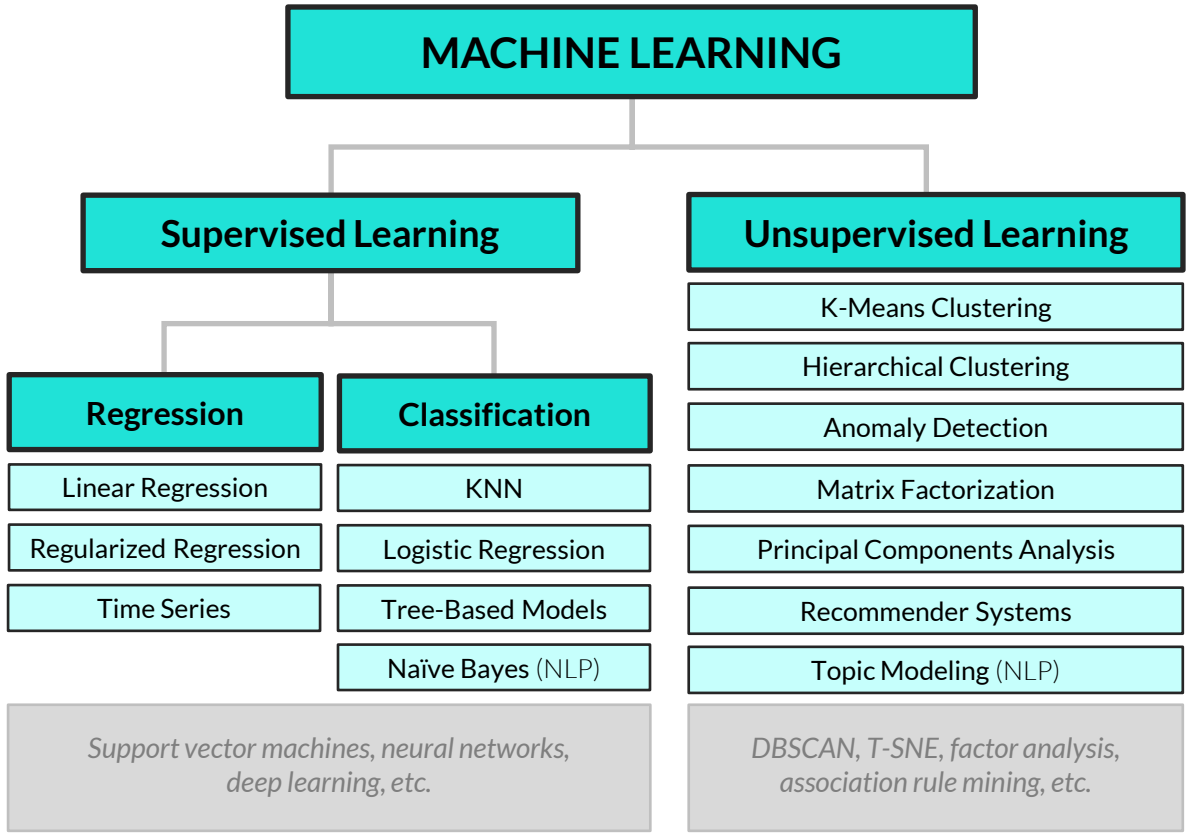
Finding patterns and relationships in data

How can I segment my customers?

Which TV shows should I recommend to each user?

COMMON ALGORITHMS

These are some of the most **common machine learning algorithms** that data scientists use in practice



Another category of machine learning algorithms is called **reinforcement learning**, which is commonly used in robotics and gaming

Fields like **deep learning** and **natural language processing** utilize both supervised and unsupervised learning techniques

DATA SCIENCE WORKFLOW

- What is Data Science?
- Essential Skills
- Machine Learning
- Data Science Workflow

The **data science workflow** consists of scoping the project, gathering, cleaning and exploring the data, applying models, and sharing insights with end users

This is not a linear process! *You'll likely go back to further gather, clean and explore your data*

STEP 1: SCOPING A PROJECT

What is Data Science?

Essential Skills

Machine Learning

Data Science Workflow

1 → 2 ⇄ 3 ⇄ 4 ⇄ 5 → 6

Scoping a Project | Gathering Data | Cleaning Data | Exploring Data | Modeling Data | Sharing Insights

Projects don't start with *data*, they start with a **clearly defined scope**:

- Who are your end users or stakeholders?
- What business problems are you trying to help them solve?
- Is this a supervised or unsupervised learning problem? *(do you even need data science?)*
- What data do you need for your analysis?

STEP 2: GATHERING DATA

A project is only as strong as the underlying data, so **gathering the right data** is essential to set a proper foundation for your analysis

Data can come from a variety of sources, including:

- Files *(flat files, spreadsheets, etc.)*
- Databases
- Websites
- APIs

STEP 3: CLEANING DATA

- What is Data Science?
- Essential Skills
- Machine Learning
- Data Science Workflow

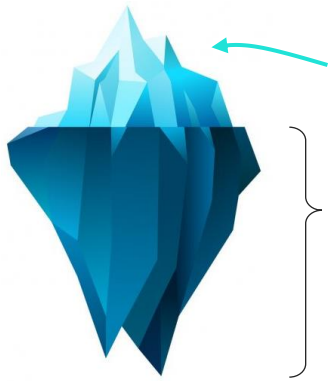
1 → 2 ⇄ 3 ⇄ 4 ⇄ 5 → 6

Scoping a Project | Gathering Data | **Cleaning Data** | Exploring Data | Modeling Data | Sharing Insights

A popular saying within data science is *"garbage in, garbage out"*, which means that **cleaning data** properly is key to producing accurate and reliable results

Data cleaning tasks may include:

- Resolving formatting issues
- Correcting data types
- Imputing missing data
- Restructuring the data



Building models
The flashy part of data science

Cleaning data
Less fun, but very important

(Data scientists estimate that around 50-80% of their time is spent here!)

STEP 4: EXPLORING DATA

- What is Data Science?
- Essential Skills
- Machine Learning
- Data Science Workflow

1 → 2 ⇄ 3 ⇄ 4 ⇄ 5 → 6

Scoping a Project → Gathering Data ⇄ Cleaning Data ⇄ **Exploring Data** ⇄ Modeling Data → Sharing Insights

Exploratory data analysis (EDA) is all about exploring and understanding the data you're working with before applying models or algorithms

EDA tasks may include:

- Slicing & dicing the data
- Profiling the data
- Visualizing the data

A good number of the **final insights** that you share will come from the exploratory analysis phase!

STEP 5: MODELING DATA

Modeling data involves structuring and preparing data for specific modeling techniques, and applying those models to make predictions or discover patterns

Modeling tasks include:

- Data splitting
- Feature selection & engineering
- Training & validating models

With fancy new algorithms introduced every year, you may feel the need to learn and apply the latest and greatest techniques

In practice, **simple is best**; businesses & leadership teams appreciate solutions that are easy to understand, interpret and implement

STEP 6: SHARING INSIGHTS

1 → 2 ⇄ 3 ⇄ 4 ⇄ 5 → 6

Scoping a Project | Gathering Data | Cleaning Data | Exploring Data | Modeling Data | **Sharing Insights**

The final step of the workflow involves summarizing your key findings and **sharing insights** with end users or stakeholders:

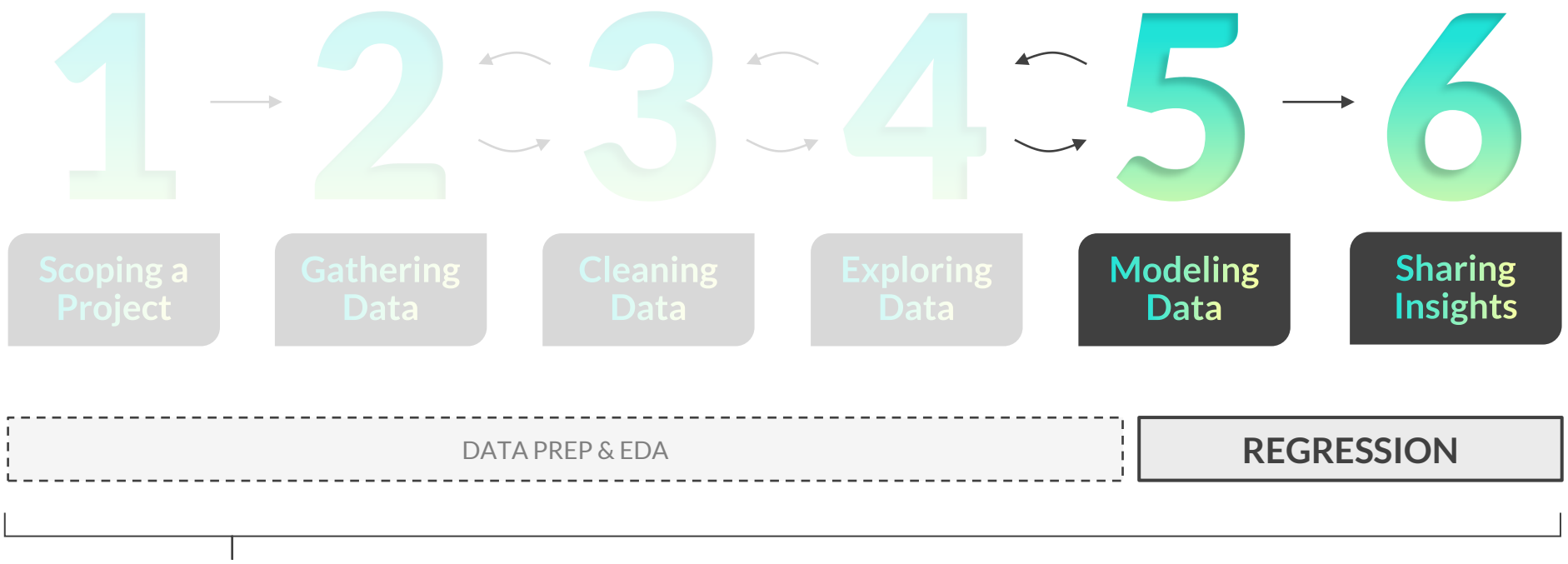
- Reiterate the problem
- Interpret the results of your analysis
- Share recommendations and next steps
- Focus on potential impact, not technical details

Even with all the technical work that's been done, it's important to remember that the focus here is on **non-technical solutions**

NOTE: Another way to share results is to deploy your model, or put it into production

REGRESSION MODELING

- What is Data Science?
- Essential Skills
- Machine Learning
- Data Science Workflow



Regression models are used to predict the value of numeric variables

Even though regression models fall into the final two steps of the workflow, data prep & EDA should *always* come first to help you get the most out of your models

KEY TAKEAWAYS

Data science is about using data to make smart decisions

- *Supervised learning techniques use historical data to predict the future, and unsupervised learning techniques use algorithms to find patterns and relationships*

Data scientists have both **coding** and **math** skills along with **domain expertise**

- *In addition to technical expertise, soft skills like communication, problem-solving, curiosity, creativity, grit, and Googling prowess round out a data scientist's skillset*

The **data science workflow** starts with defining a clear scope

- *Once the project scope is defined, you can move on to gathering and cleaning data, performing exploratory data analysis, preparing data for modeling, applying algorithms, and sharing insights with end users*

Regression modeling is a key skill used for predicting real-world outcomes

- *Data scientists are often asked to create regression models used to predict numeric values like revenue, website traffic, order volumes, and much more*

REGRESSION 101

REGRESSION 101

In this section we'll cover the basics of **regression**, including key modeling terminology, the types & goals of regression analysis, and the regression modeling workflow

TOPICS WE'LL COVER:

- Regression 101
- Goals of Regression
- Types of Regression
- The Modeling Workflow

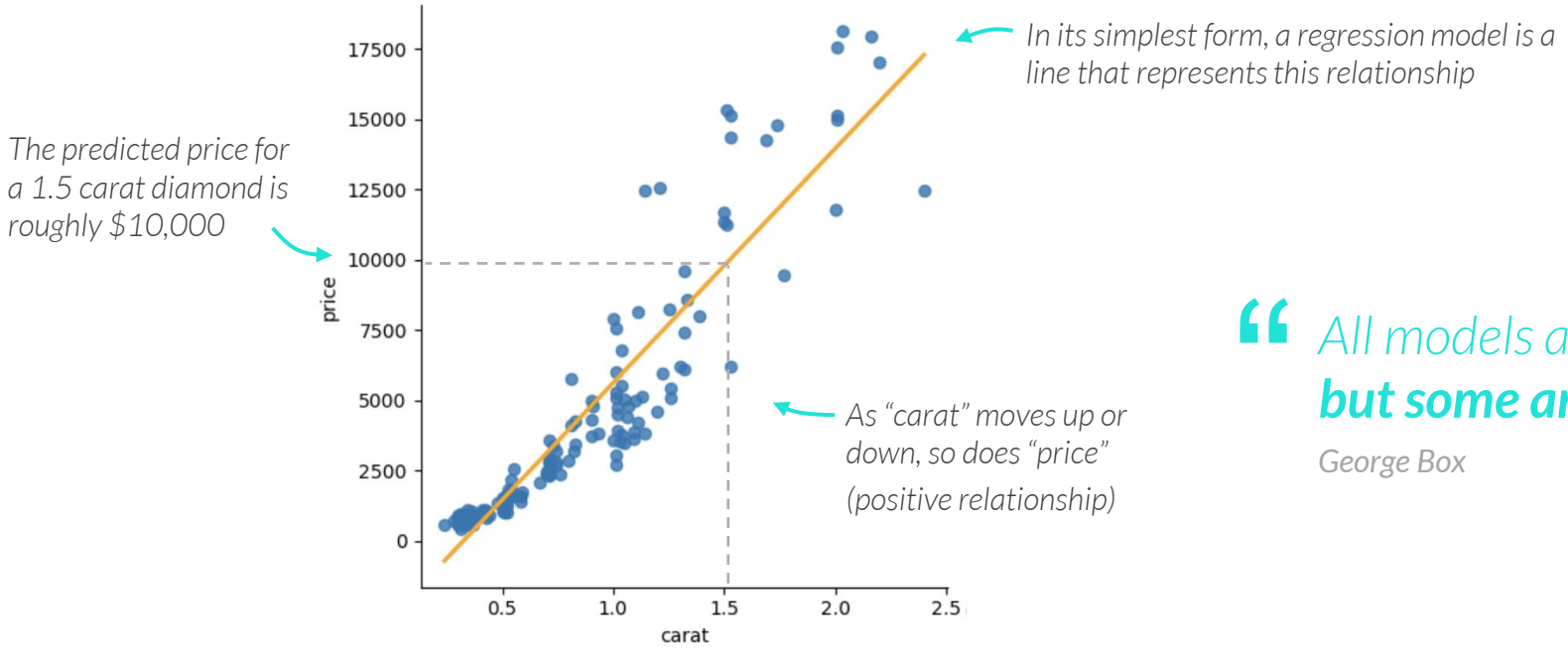
GOALS FOR THIS SECTION:

- Introduce the basics of regression modeling
- Understand key modeling terminology
- Discuss the different goals of regression modeling
- Review the regression modeling workflow

REGRESSION 101

Regression analysis is supervised learning technique used to predict a numeric variable *(target)* by modeling its relationship with a set of other variables *(features)*

EXAMPLE *Predicting the price of diamonds based on carat weight*



> *All models are wrong,*
> ***but some are useful***
>
> *George Box*

REGRESSION 101

Regression analysis is supervised learning technique used to predict a numeric variable *(target)* by modeling its relationship with a set of other variables *(features)*

y Target

- This is the variable **you're trying to predict**
- The target is also known as *"Y"*, *"model output"*, *"response"*, or *"dependent"* variable
- Regression helps understand how the target variable is impacted by the features

x Features

- These are the variables that **help you predict the target variable**
- Features are also known as *"X"*, *"model inputs"*, *"predictors"*, or *"independent"* variables
- Regression helps understand how the features impact, or *predict*, the target

REGRESSION 101

EXAMPLE

Predicting the price of diamonds based on diamond characteristics

	carat	cut	color	clarity	depth	table	x	y	z	price
0	0.26	Ideal	H	IF	61.1	57.0	4.12	4.16	2.53	468
1	1.20	Ideal	H	VS1	61.6	57.0	6.82	6.85	4.21	8275
2	0.62	Good	E	SI2	59.4	65.0	5.54	5.48	3.27	1333
3	1.50	Good	I	VVS2	63.3	58.0	7.24	7.27	4.59	9909
4	1.27	Very Good	I	SI2	59.7	57.0	6.99	7.04	4.19	5371

***Price** is our **target**, since it's what we want to predict*

*Since price is **numerical**, we'll use regression to predict it*

***Carat**, **cut**, **color**, **clarity**, and **the rest of the columns** are all **features**, since they can help us explain, or predict, the price of diamonds*

REGRESSION 101

EXAMPLE

Predicting the price of diamonds based on diamond characteristics

	carat	cut	color	clarity	depth	table	x	y	z	price
0	0.26	Ideal	H	IF	61.1	57.0	4.12	4.16	2.53	468
1	1.20	Ideal	H	VS1	61.6	57.0	6.82	6.85	4.21	8275
2	0.62	Good	E	SI2	59.4	65.0	5.54	5.48	3.27	1333
3	1.50	Good	I	VVS2	63.3	58.0	7.24	7.27	4.59	9909
4	1.27	Very Good	I	SI2	59.7	57.0	6.99	7.04	4.19	5371
5	0.90	Ideal	E	SI2	62.1	56.0	6.25	6.19	3.86	???

*We'll use records with **observed values** for both the features and target to "train" our regression model...*

*...then apply that model to new, **unobserved values** containing features but no target*

This is what our model will predict!

GOALS OF REGRESSION

Regression models are used for two primary goals: **prediction** and **inference**

- The goal shapes the modeling approach, including the regression algorithm used, the complexity of the model, and more

PREDICTION

- Used to **predict** the target as accurately as possible
- *"What is the predicted price of a diamond given its characteristics?"*

INFERENCE

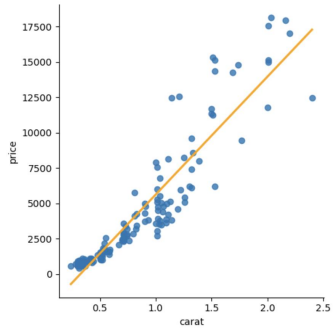
- Used to **understand the relationships** between the features and target
- *"How much do a diamond's size and weight impact its price?"*

You often need to **strike a balance** between these goals – a model that is very inaccurate won't be too trustworthy for inference, and understanding the impact that variables have on predictions can help make them more accurate

TYPES OF REGRESSION

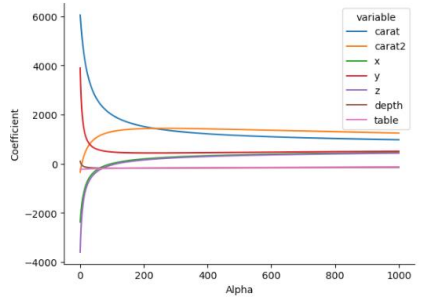
These are some of the major **types of regression** modeling techniques:

Linear Regression



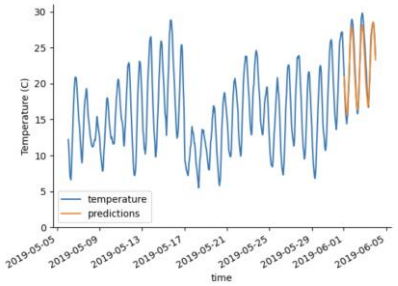
Models the relationship between the features & target using a linear equation

Regularized Regression



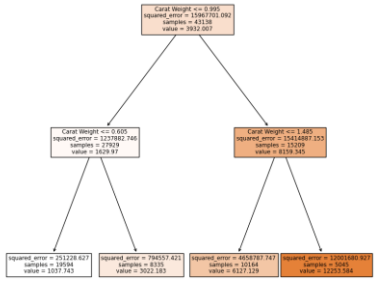
An extension of linear regression that penalizes model complexity

Time-Series Forecasting



Predicts future data using historical trends & seasonality

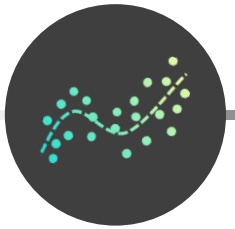
Tree-Based Regression



Splits data by maximizing the difference between groups

Even though **logistic regression** (which you may have heard of) has "regression" in its name, it's actually a classification modeling technique!

REGRESSION MODELING WORKFLOW



- Regression 101
- Goals of Regression
- Types of Regression
- The Modeling Workflow

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

KEY TAKEAWAYS

Regression modeling is used to **predict numeric values**

- *There are several types of regression models, but we will mostly focus on linear regression in this course*

The **target** is the value we want to predict, and the **features** help us predict it

- *The target is also known as "Y", "model output", "response", or "dependent" variable*
- *Features are also known as "X", "model inputs", "predictors", or "independent" variables*

Regression Modeling has two primary goals: **prediction** and **inference**

- *Prediction focuses on predicting the target as accurately as possible*
- *Inference is used to understand the relationship between the features and target*

The **modeling workflow** is designed to ensure strong performance

- *Splitting data, feature engineering, and model validation all work to ensure your model is as accurate as possible*

PRE-MODELING DATA PREP & EDA

PRE-MODELING DATA PREP & EDA

In this section we'll review the **data prep & EDA** steps required before applying regression algorithms, including key techniques to explore the target, features, and their relationships

TOPICS WE'LL COVER:

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

GOALS FOR THIS SECTION:

- Visualize & explore the target and features
- Quantify the strength of linear relationships
- Identify relationships between the target and features, as well as between features themselves
- Review the data prep steps required for modeling

EDA FOR REGRESSION

Exploratory data analysis (EDA) is the process of exploring and visualizing data to find useful patterns & insights that help inform the modeling process

- In regression, EDA lets you identify & understand the most promising features for your model
- It also helps uncover potential issues with the features or target that need to be addressed

When performing **EDA for regression**, it's key to explore:

- The target variable
- The features
- Feature-target relationships
- Feature-feature relationships

Even though data cleaning comes before exploratory data analysis, it's common for EDA to **reveal additional data cleaning steps** needed before modeling

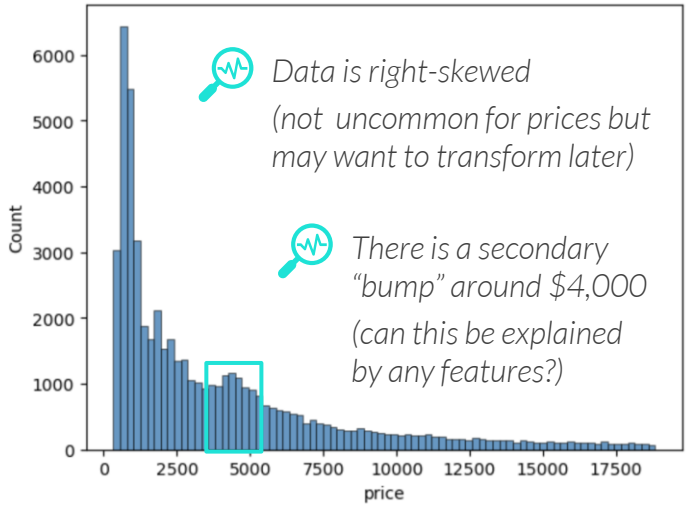
EXPLORING THE TARGET

Exploring the target variable lets you understand where most values lie, how spread out they are, and what shape, or distribution, they have

- Charts like **histograms** and **boxplots** are great tools to explore regression targets

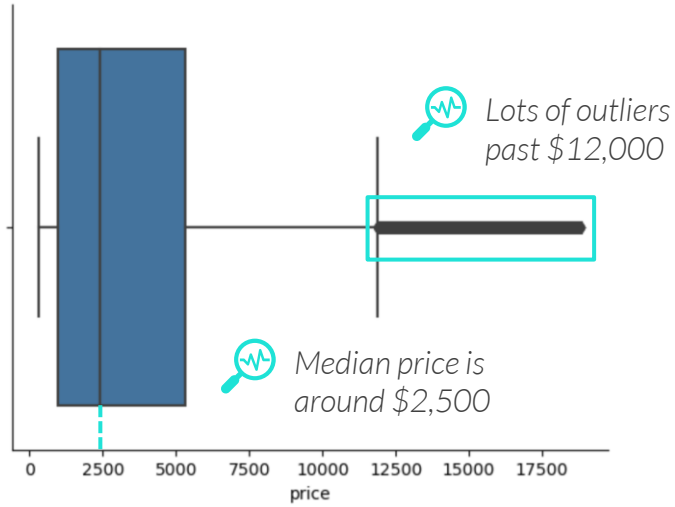
*The **seaborn** library is ideal for charts*

```
import seaborn as sns

sns.histplot(diamonds["price"]);
```




```
sns.boxplot(x=diamonds["price"])

sns.despine();
```

***sns.despine()** removes the top & right borders*



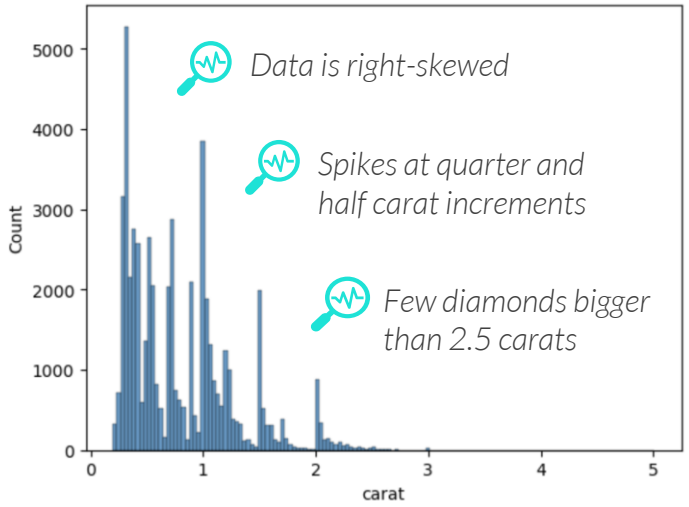
EXPLORING THE FEATURES

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

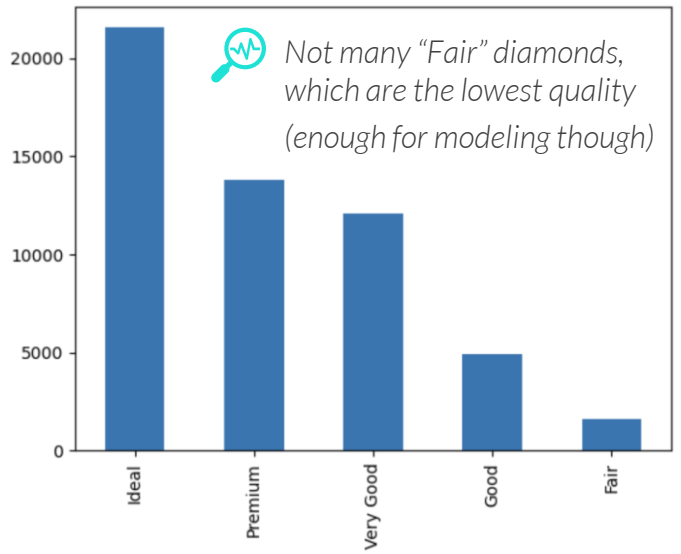
Exploring the features helps you understand them and start to get a sense of the transformations you may need to apply to each one

- **Histograms** and **boxplots** let you explore *numeric* features
- The **.value_counts()** method and **bar charts** are best for *categorical* features

```
sns.histplot(diamonds["carat"]);
```




```
diamonds["cut"].value_counts().plot.bar();
```



ASSIGNMENT: EXPLORING THE TARGET & FEATURES

NEW MESSAGE

June 28, 2023

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

Subject: **EDA**

Hi there, glad to have you on the team!

We've been getting a lot of partnership requests from various business units, so I could really use your help.

Can you explore the computer prices data set at a high level?

I want to see a boxplot and histogram of the "price" variable and histograms of the "speed", and "ram" variables as well.

Thanks!

01_EDA_assignments.ipynb

Reply | Forward

Key Objectives

1. Read in the "Computers.csv" file
2. Explore the target variable: "price"
3. Explore the features: "speed" and "RAM"

LINEAR RELATIONSHIPS

It's common for numeric variables to have **linear relationships** between them

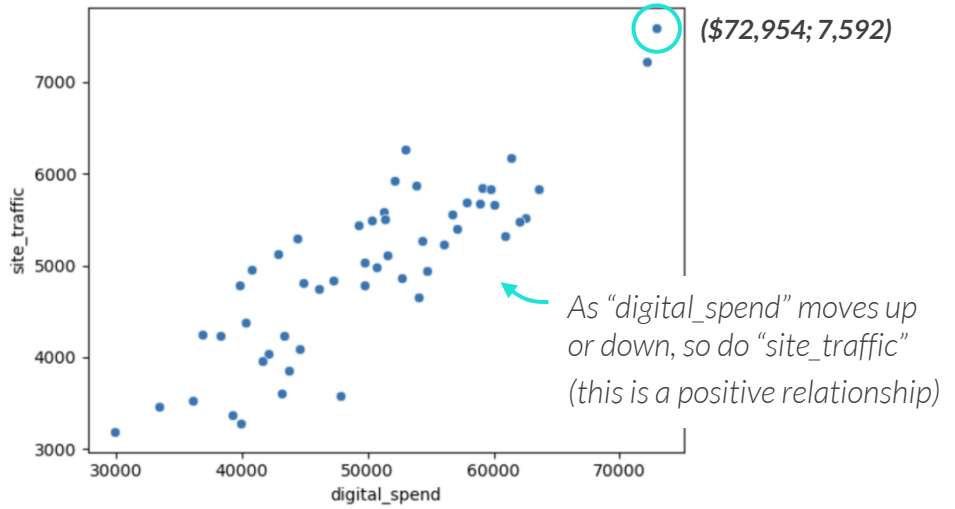
- When one variable changes, so does the other
- This relationship is commonly visualized with a **scatterplot**

EXAMPLE *Relationship between digital advertising spend and site traffic*

```
ads.head()
```

	week	offline_spend	digital_spend	site_traffic	site_load_time
0	1	35263	46125	4751	3.380876
1	2	53261	60007	5661	2.804510
2	3	43149	50314	5491	3.986187
3	4	47527	44432	5293	3.653095
4	5	37959	72954	7592	4.133515

```
sns.scatterplot(data=ads, x="digital_spend", y="site_traffic");
```

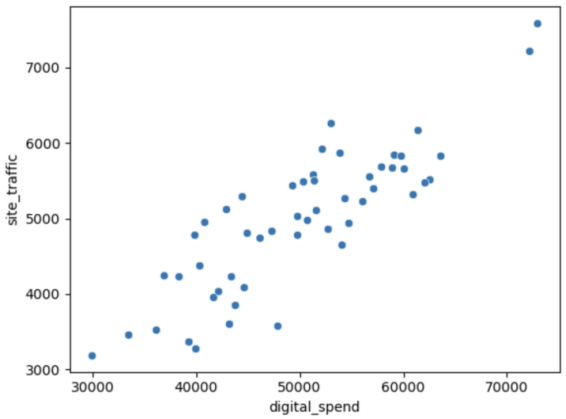


LINEAR RELATIONSHIPS

There are two possible linear relationships: **positive** & **negative**

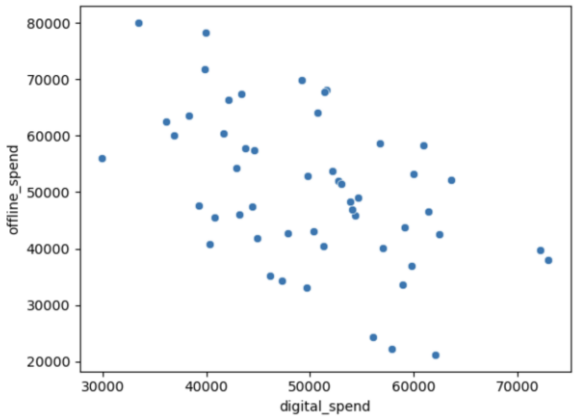
- Variables can also have **no relationship**

Positive Relationship



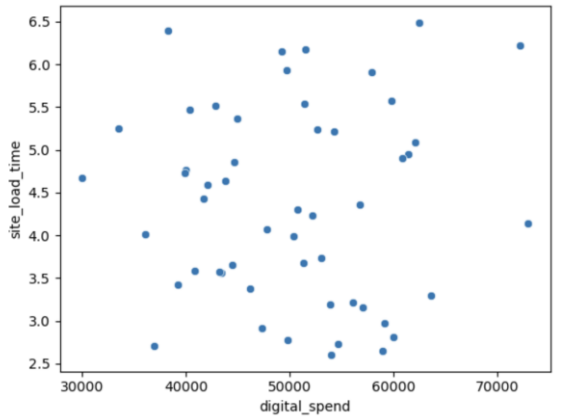
*As one changes, the other changes in the **same direction***

Negative Relationship



*As one changes, the other changes in the **opposite direction***

No Relationship



No association can be found between the changes in one variable and the other

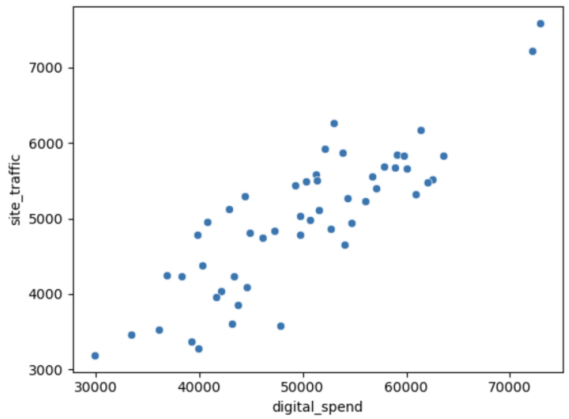
CORRELATION

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

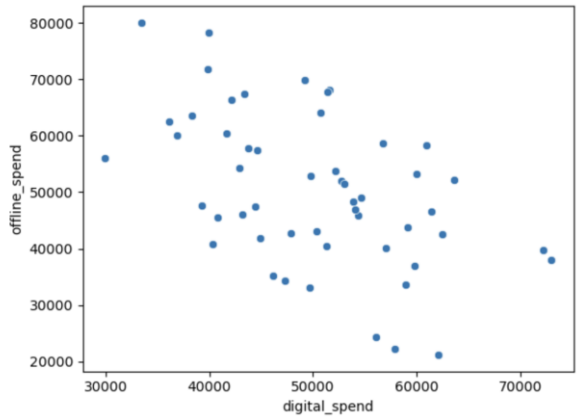
The **correlation (r)** measures the strength & direction of a linear relationship *(-1 to 1)*

- You can use the **.corr()** method to calculate correlations in Pandas – df["col1"].corr(df["col2"])

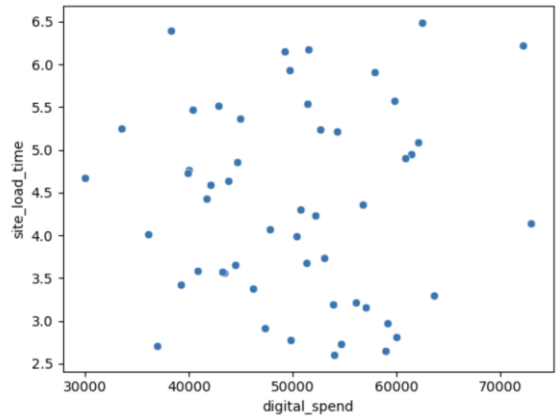
r = 0.858



r = -0.499



r = 0.008



-1	-0.9	-0.7	-0.4	-0.1	0.1	0.4	0.7	0.9	1
Very Strong	Strong	Moderate	Weak	None	Weak	Moderate	Strong	Very Strong	

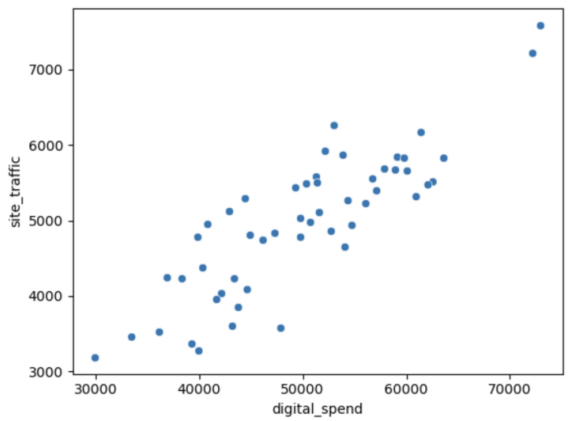
Negative Positive

CORRELATION

The **correlation (r)** measures the strength & direction of a linear relationship *(-1 to 1)*

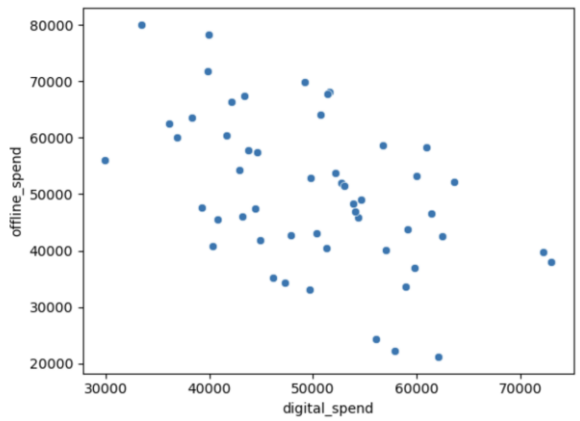
- You can use the **.corr()** method to calculate correlations in Pandas – df["col1"].corr(df["col2"])

r = 0.858



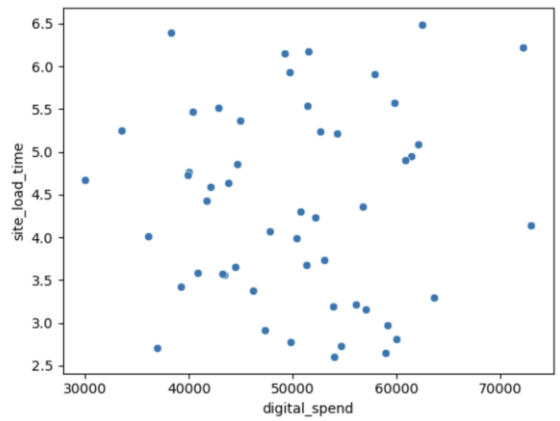
Strong positive correlation

r = -0.499



Moderate negative correlation

r = 0.008



No correlation

PRO TIP: Highly correlated variables tend to be the best candidates for your "baseline" regression model, but other variables can still be useful features after exploring non-linear relationships and fixing data issues

PRO TIP: CORRELATION MATRIX

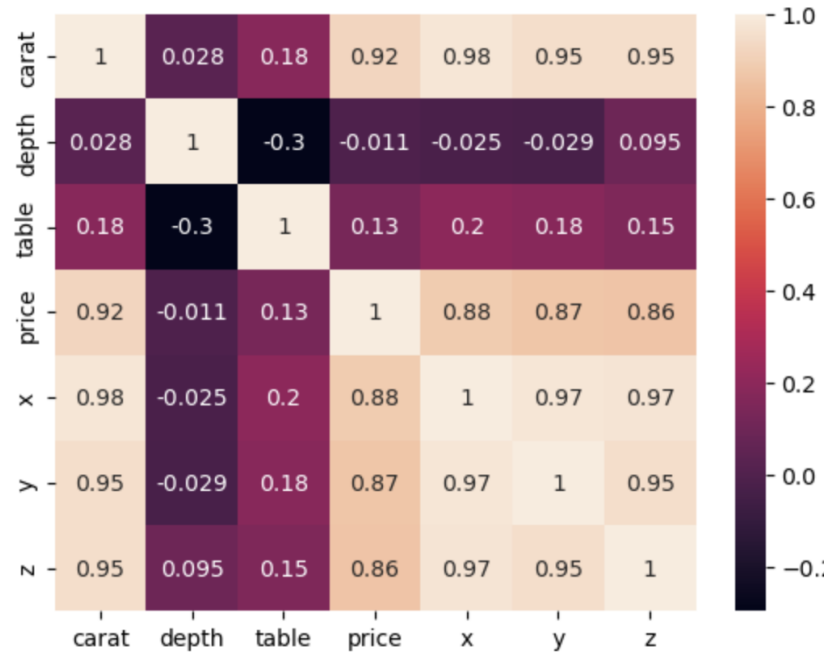
A **correlation matrix** returns the correlation between each column in a DataFrame

- Use **df.corr(numeric_only=True)** to only consider numeric columns in the DataFrame
- Wrap you correlation matrix in **sns.heatmap()** to make it easier to interpret

```
diamonds.corr(numeric_only=True)
```

	carat	depth	table	price	x	y	z
carat	1.000000	0.028234	0.181602	0.921591	0.975093	0.951721	0.953387
depth	0.028234	1.000000	-0.295798	-0.010630	-0.025289	-0.029340	0.094927
table	0.181602	-0.295798	1.000000	0.127118	0.195333	0.183750	0.150915
price	0.921591	-0.010630	0.127118	1.000000	0.884433	0.865419	0.861249
x	0.975093	-0.025289	0.195333	0.884433	1.000000	0.974701	0.970771
y	0.951721	-0.029340	0.183750	0.865419	0.974701	1.000000	0.952005
z	0.953387	0.094927	0.150915	0.861249	0.970771	0.952005	1.000000

```
sns.heatmap(diamonds.corr(numeric_only=True), annot=True);
```

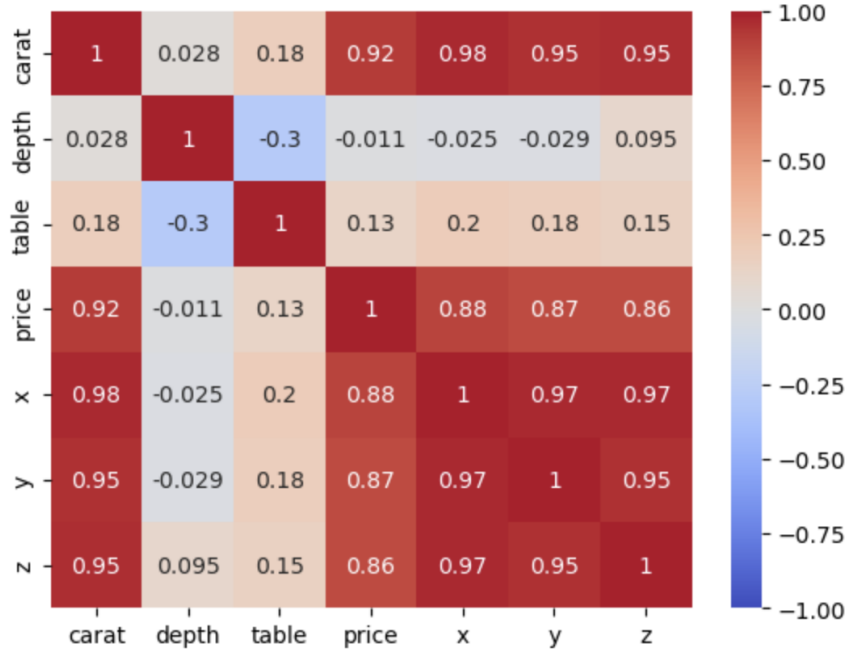


PRO TIP: CORRELATION MATRIX

A **correlation matrix** returns the correlation between each column in a DataFrame

- Use **df.corr(numeric_only=True)** to only consider numeric columns in the DataFrame
- Wrap you correlation matrix in **sns.heatmap()** to make it easier to interpret

```
sns.heatmap(diamonds.corr(numeric_only=True), annot=True, vmin=-1, vmax=1, cmap="coolwarm");
```



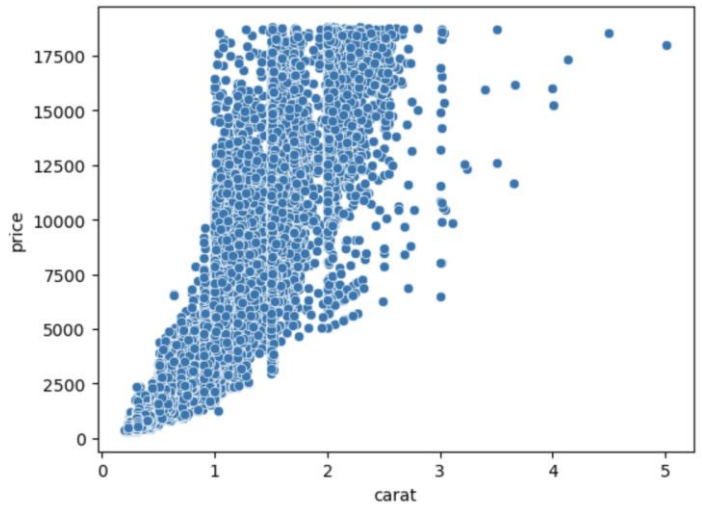
Setting the maximum and minimum values at -1 and 1 respectively, and adding a divergent color scale can help increase interpretability

FEATURE-TARGET RELATIONSHIPS

Exploring **feature-target relationships** helps identify potentially useful predictors

- Use scatterplots for numeric features and bar charts for categorical features

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

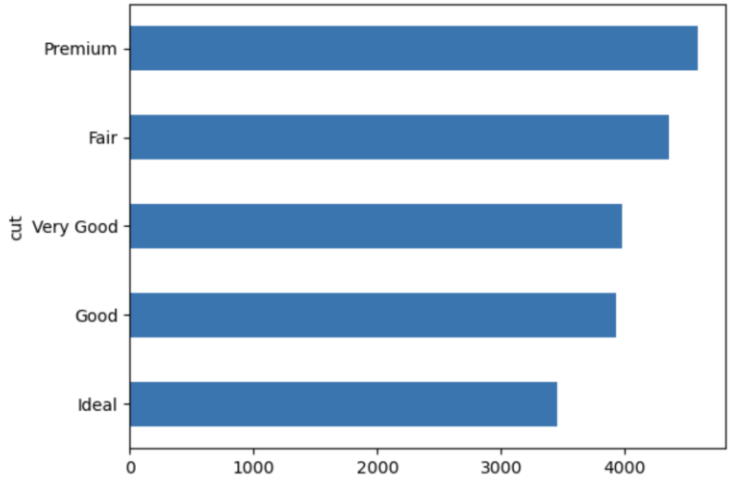



```
diamonds["carat"].corr(diamonds["price"])
```

```
0.9215912778016124
```

Strong positive relationship between carat & price (potentially exponential and not linear)

```
diamonds.groupby("cut")["price"].mean().sort_values().plot.barh();
```



Average prices differ between cuts:

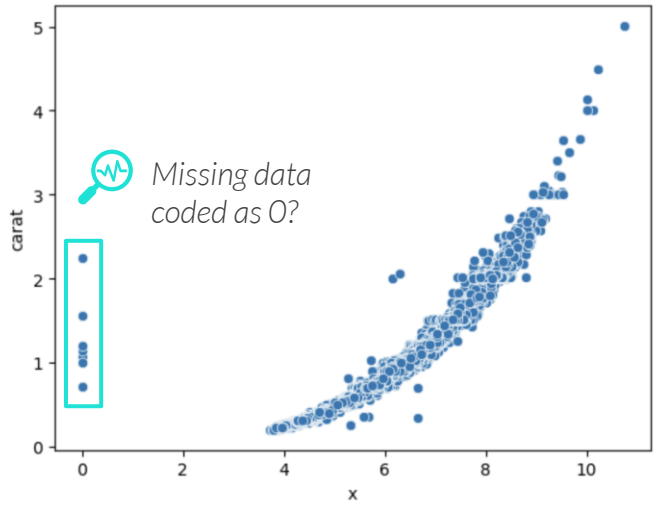
- *"Fair" cut diamonds (worst) have the 2nd highest average price*
- *"Ideal" cut diamonds (best) have the lowest average price*

FEATURE-FEATURE RELATIONSHIPS

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

Exploring **feature-feature relationships** helps identify highly correlated features, which can cause problems with regression models *(more on this later!)*

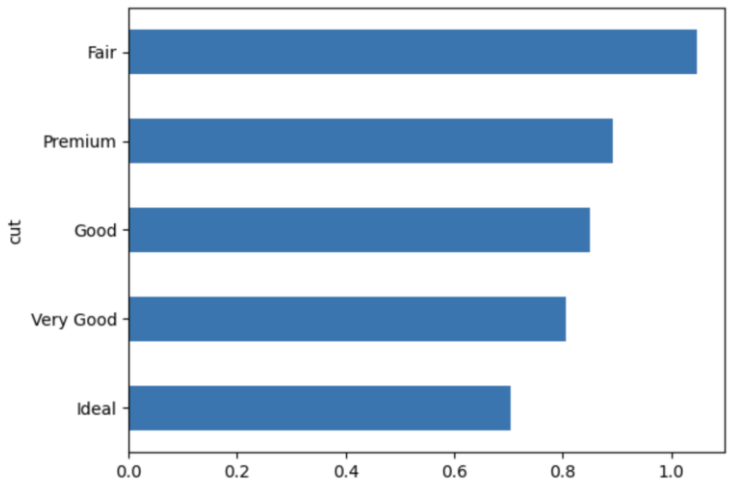
```
sns.scatterplot(diamonds, x="x", y="carat");
```



Strong positive relationship between diamond length (x) and carat

(we might not be able to include both features in our model – more later!)

```
diamonds.groupby("cut")["carat"].mean().sort_values().plot.barh();
```



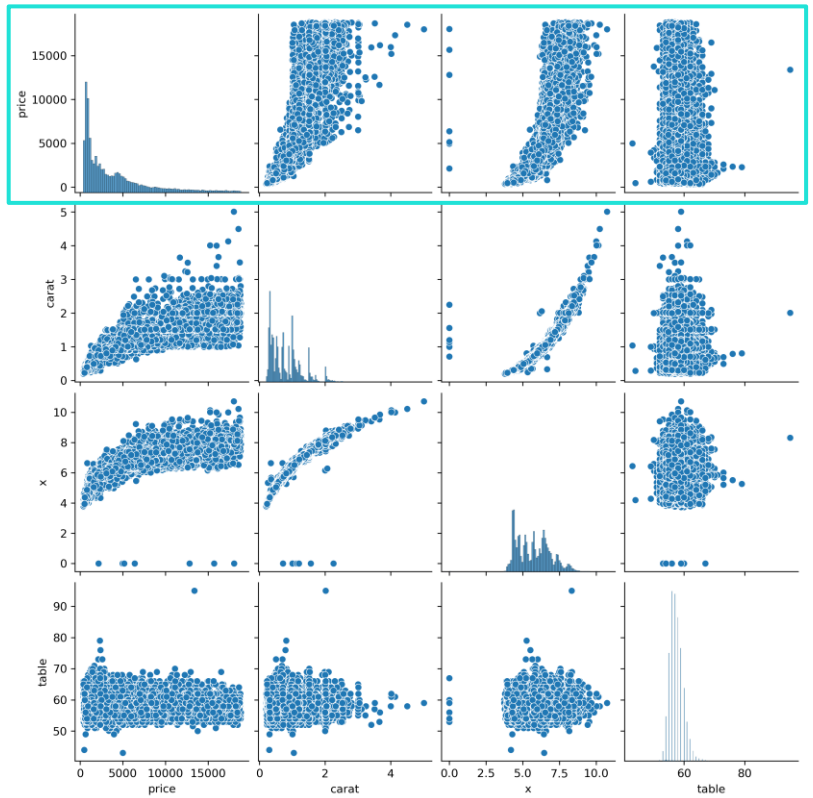
"Fair" cut diamonds have the largest average carat size, which explains why their average price is higher than "Ideal" cut diamonds, which have the smallest average carat size

PRO TIP: PAIRPLOTS

Use **sns.pairplot()** to create a pairplot that shows all the scatterplots and histograms that can be made using the numeric variables in a DataFrame

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

```
sns.pairplot(diamonds);
```



*The row with your **target** can help explore numeric feature-target relationships quickly!*

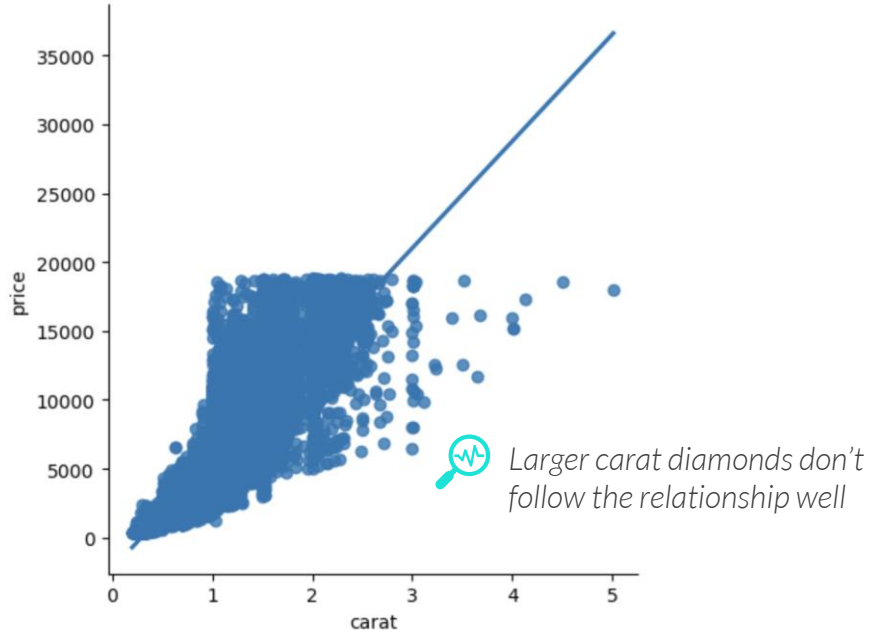
PRO TIP: It can take a long time to generate pairplots for large datasets, so consider using **df.sample()** to speed up the process significantly without losing high-level insights!

PRO TIP: LMPLOTS

Use **sns.lmplot()** to create a scatterplot with a fitted regression line *(more soon!)*

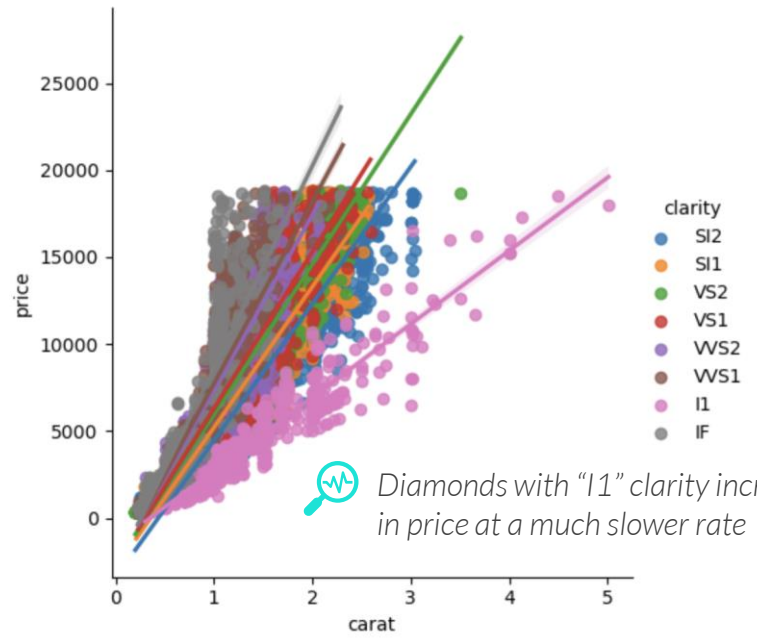
- This is commonly used to explore the impact of other variables on a linear relationship
- **sns.lmplot(df, x="feature", y="target", hue="categorical feature")**

```
sns.lmplot(diamonds, x="carat", y="price");
```



Larger carat diamonds don't follow the relationship well

```
sns.lmplot(diamonds, x="carat", y="price", hue="clarity");
```



Diamonds with "I1" clarity increase in price at a much slower rate

ASSIGNMENT: EXPLORING RELATIONSHIPS

NEW MESSAGE

June 28, 2023

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

Subject: **Exploring Variable Relationships**

Hi there!

Thanks for taking a quick look at those variables for me.

Now let's take it one step further and explore variable relationships in the data.

Thanks!

01_EDA_assignments.ipynb

Reply Forward

Key Objectives

1. Build a correlation matrix heatmap
2. Build a pair plot of numeric features
3. Build an lmplot of "RAM" vs. "price"

PREPARING FOR MODELING

- EDA for Regression
- Exploring the Target
- Exploring the Features
- Linear Relationships
- Exploring Relationships
- Preparing for Modeling

Preparing for modeling involves structuring the data as a valid input for a model:

- Stored in a single table *(DataFrame)*
- Aggregated to the right grain *(1 row per target)*
- Non-null *(no missing values)* and numeric

Data ready for EDA

Carat	Cut	Color	Price
0.90	Good	D	$3,423
0.31	Fair	E	$527
0.52	Very Good		$1,250
1.55	Ideal	D	$12,500

Data ready for modeling

Index	Carat	Cut	D	Missing	Price ($)
0	0.90	1	1	0	3423
1	0.31	0	0	0	527
2	0.52	2	0	1	1250
3	1.55	3	1	0	12500

Features *Target*

We'll revisit this during the **feature engineering** section of the course

KEY TAKEAWAYS

It's critical to **explore the features & target** in your data

- *Use histograms and boxplots to visualize and explore your target and any numeric features*
- *Use bar charts to visualize and explore categorical features*

Use scatterplots & correlations to **find linear relationships** in your data

- *Features that are highly correlated with your target are likely strong predictors of it*
- *Features that are highly correlated with each other can cause problems in a model*

Remember to **prepare you data for modeling** after performing EDA

- *The data must be stored in a single table, each column must be numeric, and missing values must be handled*

SIMPLE LINEAR REGRESSION

SIMPLE LINEAR REGRESSION

In this section we'll build our first **simple linear regression** models in Python and learn about the metrics and statistical tests that help evaluate their quality and output

TOPICS WE'LL COVER:

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

GOALS FOR THIS SECTION:

- Introduce the linear regression equation
- Visualize the the least squared error method for finding the line of best fit
- Build linear regression models in Python and use them to make predictions for the target
- Interpret the model summary statistics and use them to evaluate the model's accuracy

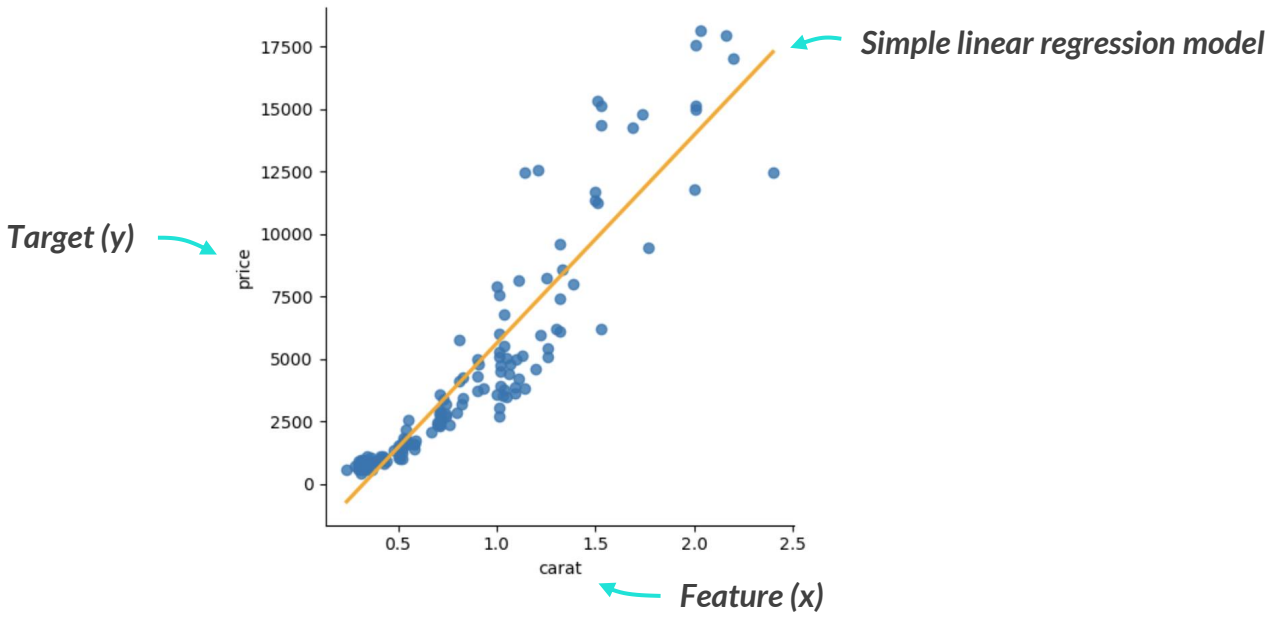
SIMPLE LINEAR REGRESSION

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

Simple linear regression models use a *single feature* to predict the target

- This is achieved by fitting a line through the data points in a scatterplot *(like an lmplot!)*

EXAMPLE | *Simple linear regression model using carat and price*



LINEAR REGRESSION MODEL

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

The **linear regression model** is an equation that best describes a linear relationship

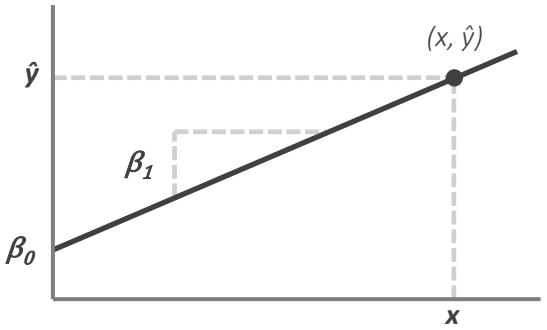
*The **predicted** value for the target*

The value for the feature

$$\hat{y} = \beta_0 + \beta_1 x$$

*The **y-intercept***

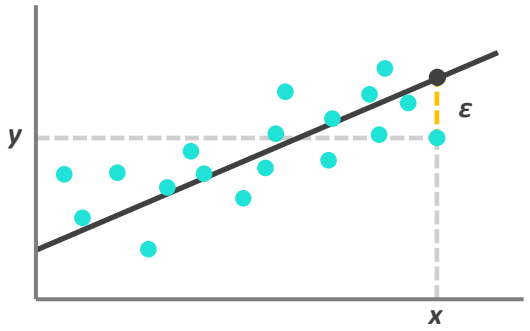
*The **slope** of the relationship*

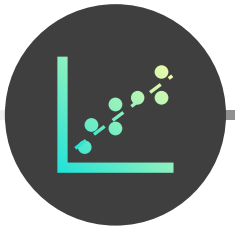


*The **actual** value for the target*

$$y = \beta_0 + \beta_1 x + \epsilon$$

*The **error**, or **residual**, caused by the difference between the actual and predicted values*





LEAST SQUARED ERROR

The **least squared error** method finds the line that best fits through the data

- It works by solving for the line that **minimizes the sum of squared error**
- The equation that minimizes error can be solved with **linear algebra**

Why "squared" error?

- Squaring the residuals converts them into **positive values**, and prevents positive and negative distances from cancelling each other out *(this makes the algebra to solve the line much easier, too)*
- One drawback of squared errors is that outliers can significantly impact the line *(more later!)*

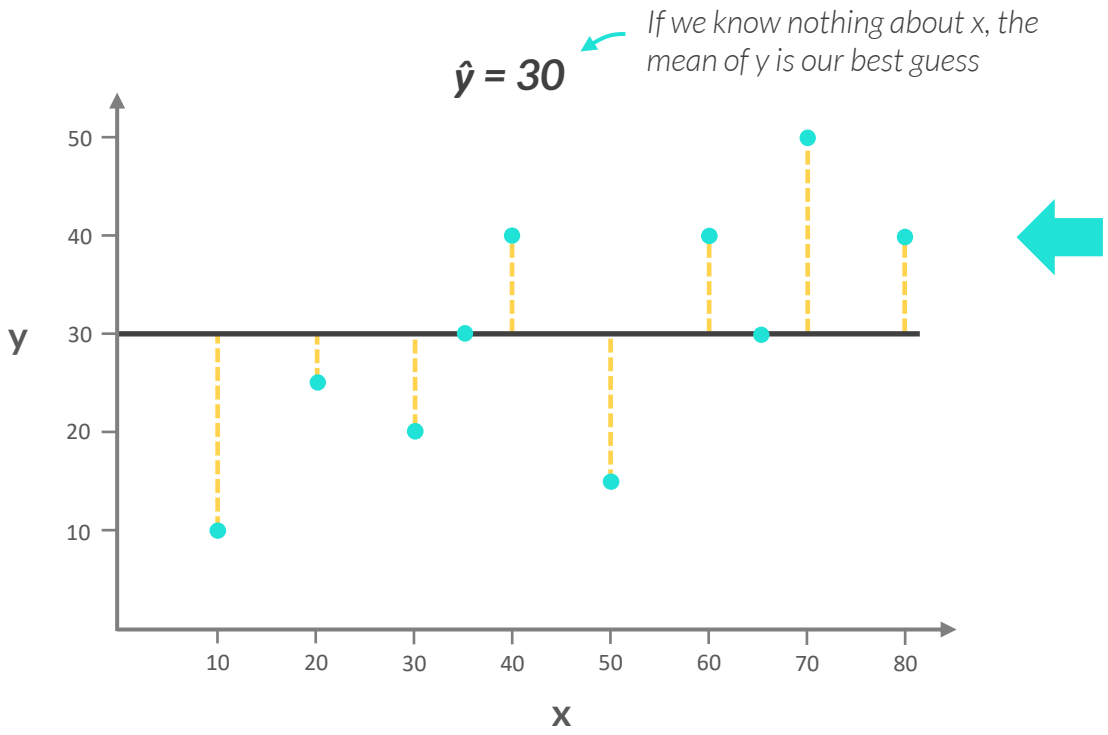
Ordinary Least Squares (OLS) is another term for traditional linear regression

There are other frameworks for linear regression that don't use least squared error, but they are rarely used outside of specialized domains

LEAST SQUARED ERROR

The **least squared error** method finds the line that best fits through the data

- It works by solving for the line that **minimizes the sum of squared error**
- The equation that minimizes error can be solved with **linear algebra**



x	y
10	10
20	25
30	20
35	30
40	40
50	15
60	40
65	30
70	50
80	40

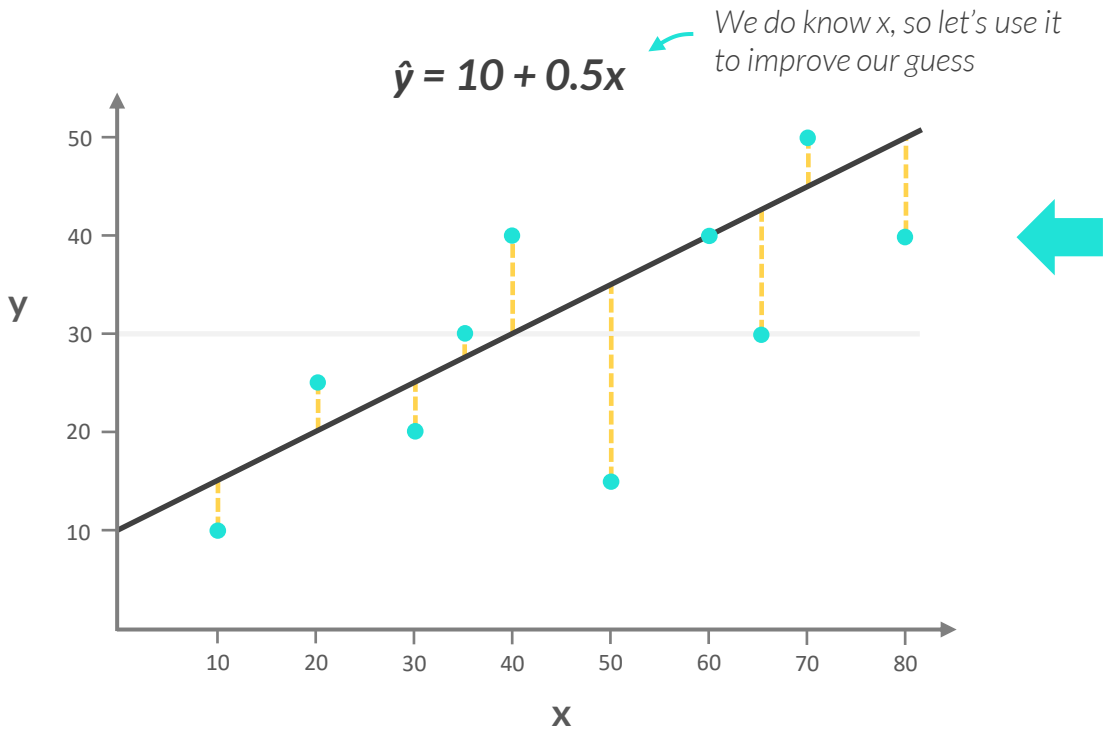
SUM OF SQUARED ERROR: = **1,450**

LEAST SQUARED ERROR

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

The **least squared error** method finds the line that best fits through the data

- It works by solving for the line that **minimizes the sum of squared error**
- The equation that minimizes error can be solved with **linear algebra**



x	y	ŷ	ε	ε^2
10	10	15	5	**25**
20	25	20	-5	**25**
30	20	25	5	**25**
35	30	27.5	-2.5	**6.25**
40	40	30	-10	**100**
50	15	35	20	**400**
60	40	40	0	**0**
65	30	42.5	12.5	**156.25**
70	50	45	-5	**25**
80	40	50	10	**100**
				II

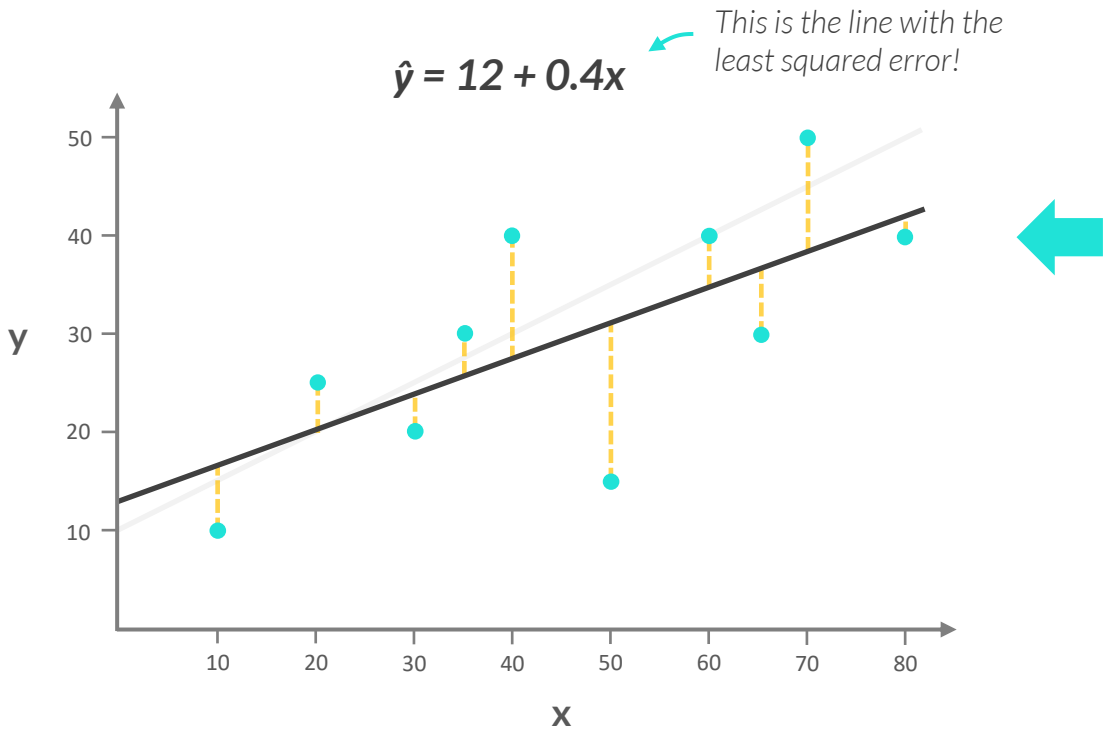
SUM OF SQUARED ERROR: **862.5**

LEAST SQUARED ERROR

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

The **least squared error** method finds the line that best fits through the data

- It works by solving for the line that **minimizes the sum of squared error**
- The equation that minimizes error can be solved with **linear algebra**



x	y	ŷ	ε	ε^2
10	10	16	6	**36**
20	25	20	-5	**25**
30	20	24	4	**16**
35	30	26	-4	**16**
40	40	28	-12	**144**
50	15	32	17	**289**
60	40	36	-4	**16**
65	30	38	8	**64**
70	50	40	-10	**100**
80	40	44	4	**16**
				II

SUM OF SQUARED ERROR: **722**

REGRESSION IN PYTHON

These Python libraries are used fit regression models: **statsmodels** & **scikit-learn**

statsmodels

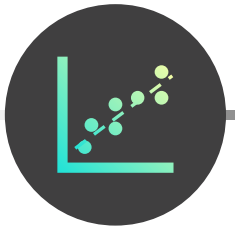
- **Ideal if your goal is inference**
- Similar output to other tools (SAS, R, Excel)
- Easy access to dozens of statistical tests
- Harder to leverage in production ML

scikit learn
machine learning in Python

- **Ideal if your goal is prediction**
- Most popular ML library in Python
- Has various models for easy comparison
- Designed to be deployed to production

Both libraries **use the same math** and return the same regression equation!

We will begin by focusing on statsmodels, but once we have the fundamentals of regression down, we'll introduce scikit-learn



REGRESSION IN STATSMODELS

You can fit a **regression in statsmodels** with just a few lines of code:

```
import statsmodels.api as sm

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

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

model.summary()
```

*1) Import **statsmodels.api** (standard alias is **sm**)*

*2) Create an "X" DataFrame with your **feature(s)** and **add a constant***

*3) Create a "y" DataFrame with your **target***

*4) Call **sm.OLS(y, X)** to set up the model, then use **.fit()** to build the model*

*5) Call **.summary()** on the model to review the model output*

Why do we need to add a constant?

- Statsmodels assumes you want to fit a model with a line that runs through the origin (0, 0)
- sm.add_constant() lets statsmodels calculate a y-intercept other than 0 for the model
- Most regression software *(like sklearn)* takes care of this step behind the scenes

REGRESSION IN STATSMODELS

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

You can fit a **regression in statsmodels** with just a few lines of code:

```
import statsmodels.api as sm

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

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

model.summary()
```

The model output can be intimidating the first time you see it, but we'll cover the important pieces in the next few lessons and later sections!

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.849
Model:	OLS	**Adj. R-squared:**	0.849
Method:	Least Squares	**F-statistic:**	3.041e+05
Date:	Wed, 02 Aug 2023	**Prob (F-statistic):**	0.00
Time:	10:47:03	**Log-Likelihood:**	-4.7276e+05
No. Observations:	53943	**AIC:**	9.455e+05
Df Residuals:	53941	**BIC:**	9.455e+05
Df Model:	1		
Covariance Type:	nonrobust		

Model summary statistics

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2256.3950	13.055	-172.840	0.000	-2281.983	-2230.807
carat	7756.4362	14.066	551.423	0.000	7728.866	7784.006

Variable summary statistics

Omnibus:	14027.005	**Durbin-Watson:**	1.999
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	153060.389
Skew:	0.939	**Prob(JB):**	0.00
Kurtosis:	11.036	**Cond. No.**	3.65

Residual (error) statistics

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

INTERPRETING THE MODEL

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

To **interpret the model**, use the "coef" column in the variable summary statistics

```
import statsmodels.api as sm

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

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

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2256.3950	13.055	-172.840	0.000	-2281.983	-2230.807
carat	7756.4362	14.066	551.423	0.000	7728.866	7784.006

$$\hat{y} = -2256 + 7756x$$

How do we interpret this?

- An increase of 1 carat in a diamond is *associated* with a $7,756 dollar increase in its price
- We **cannot** say 1 carat *causes* a $7,756 increase in price without a more rigorous experiment
- Technically, a 0-carat diamond is predicted to cost -$2,256 dollars

MAKING PREDICTIONS

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

The **.predict()** method returns model predictions for single points or DataFrames

```
import statsmodels.api as sm

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

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

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2256.3950	13.055	-172.840	0.000	-2281.983	-2230.807
carat	7756.4362	14.066	551.423	0.000	7728.866	7784.006

```
model.predict([1, 1.5])
```

```
array([9378.25919172])
```

The 1 adds a constant

$$-2256 + 7756(\mathbf{1.5}) = \mathbf{9378}$$

A 1.5 carat diamond has a predicted price of $9,378

MAKING PREDICTIONS

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

The **.predict()** method returns model predictions for single points or DataFrames

```
import statsmodels.api as sm

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

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

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2256.3950	13.055	-172.840	0.000	-2281.983	-2230.807
carat	7756.4362	14.066	551.423	0.000	7728.866	7784.006

```
carats = sm.add_constant(pd.DataFrame({"carat": [.5, 1, 1.5, 2, 2.5]}))
model.predict(carats)
```

```
0     1621.823032
1     5500.041112
2     9378.259192
3    13256.477271
4    17134.695351
dtype: float64
```

R-SQUARED

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

R-squared, or coefficient of determination, measures how much better the model is at predicting the target than using its mean *(our best guess without using features)*

- R-squared values are bounded between 0 and 1 on training data

```
import statsmodels.api as sm

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

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

model.summary()
```

```
diamonds["carat"].corr(diamonds["price"])**2
```

```
0.8493304833200086
```

Squaring the correlation *between the feature and target yields R^2 in simple linear regression (this doesn't hold in multiple linear regression)*

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.849
Model:	OLS	**Adj. R-squared:**	0.849
Method:	Least Squares	**F-statistic:**	3.041e+05
Date:	Wed, 02 Aug 2023	**Prob (F-statistic):**	0.00
Time:	10:47:03	**Log-Likelihood:**	-4.7276e+05
No. Observations:	53943	**AIC:**	9.455e+05
Df Residuals:	53941	**BIC:**	9.455e+05
Df Model:	1		
Covariance Type:	nonrobust		

*The model explains **84.9%** of the variation in price not explained by the mean of price*

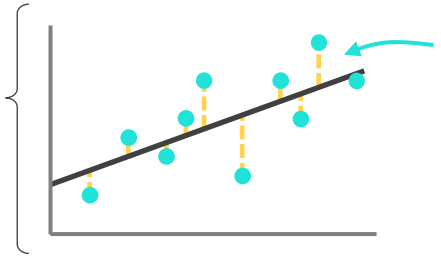
R-SQUARED

R-squared, or coefficient of determination, measures how much better the model is at predicting the target than using its mean *(our best guess without using features)*

- R-squared values are bounded between 0 and 1 on training data

$$R^2 = 1 - \frac{SSE}{SST}$$

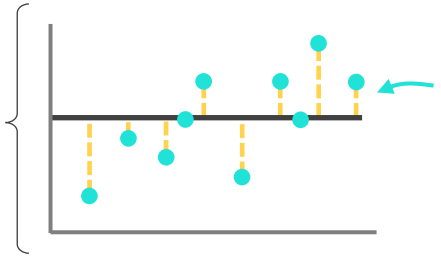
*This is the **sum of squared error** (distance between values & line)*



This is the variation of "y" not explained by "x"

- *Reducing error will increase R^2*
- *A perfect model has an error of 0, or R^2 of 1*

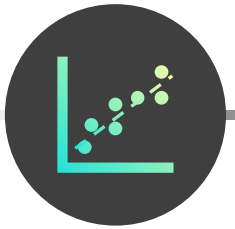
*This is the **sum of squared total** (distance between values & mean)*



This is the variation of "y" around its mean

- *If R^2 = 0, the model is no better than the mean of y*

A "good" value for R^2 is relative to your data – an R^2 of .05 might be industry leading in sports analytics, but if you were trying to prove a physics theory with an experiment, an R^2 of .95 might be a disappointment!



HYPOTHESIS TEST

Regression models include several **hypothesis tests**, including the F-test, that indicates whether our model is significantly better at predicting our target than using the mean of the target as the model

- In other words, you're trying to find significant evidence that your model isn't useless

Steps for the hypothesis test:

1) State the **null** and **alternative hypotheses**
2) Set a **significance level** (α)
3) Calculate the **test statistic** and **p-value**
4) Draw a **conclusion** from the test
    a) If p≤**α**, reject the null hypothesis *(you're confident the model isn't useless)*
    b) If p>**α**, don't reject it *(the model is probably useless, and needs more training)*

HYPOTHESES & SIGNIFICANCE LEVEL

1) For F-Tests, the **null & alternative hypotheses** are always the same:

- **H_o: F=0** – the model is NOT significantly better than the mean *(naïve guess)*
- **H_a: F≠0** – the model is significantly better than the mean *(naïve guess)*

*The hope is to **reject the null hypothesis***
(and therefore, accept the alternative)

2) The **significance level** is the threshold you set to determine when the evidence against your null hypothesis is considered "strong enough" to prove it wrong

- This is set by **alpha (α)**, which is the accepted probability of error
- The industry standard is **α = .05** *(this is what we'll use in the course)*

Some teams and industries set a much higher bar, such as .01 or even .001, making the null hypothesis **less likely to be rejected**

F-STATISTIC & P-VALUE

3) The **F-statistic** and associated **p-value** are part of the model summary and help understand the predictive power of the regression model *as a whole*

- The **F-statistic** is the ratio of variability the model explains vs the variability it doesn't
- The **p-value**, or F-significance, is the probability that your model predicts poorly

```
import statsmodels.api as sm

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

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

model.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.849
Model:	OLS	**Adj. R-squared:**	0.849
Method:	Least Squares	**F-statistic:**	3.041e+05
Date:	Wed, 02 Aug 2023	**Prob (F-statistic):**	0.00
Time:	10:47:03	**Log-Likelihood:**	-4.7276e+05
No. Observations:	53943	**AIC:**	9.455e+05
Df Residuals:	53941	**BIC:**	9.455e+05
Df Model:	1		
Covariance Type:	nonrobust		

The F-statistic is primarily used as a stepping-stone to calculate the p-value, which is **easier to interpret** and **more commonly used** in model diagnostics

HYPOTHESIS TEST CONCLUSION

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

4) Comparing the p-value and alpha lets us draw a **conclusion** from the test

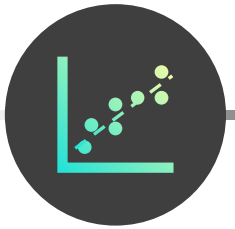
- **$p \leq \alpha$** – reject the null hypothesis *(you're confident the model isn't useless)*
- **$p > \alpha$** – don't reject it *(the model is probably useless, and needs more training)*

```
import statsmodels.api as sm

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

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

model.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.849
Model:	OLS	**Adj. R-squared:**	0.849
Method:	Least Squares	**F-statistic:**	3.041e+05
Date:	Wed, 02 Aug 2023	**Prob (F-statistic):**	0.00
Time:	10:47:03	**Log-Likelihood:**	-4.7276e+05
No. Observations:	53943	**AIC:**	9.455e+05
Df Residuals:	53941	**BIC:**	9.455e+05
Df Model:	1		
Covariance Type:	nonrobust		

Our F-statistic is much greater than 0, and the p-value is less than 0.05, so we can reject the null hypothesis (carat is a good predictor of a diamond's price!)



T-STATISTICS & P-VALUES

The **T-statistics** and associated **p-values** are part of the model summary and help understand the predictive power of *individual model coefficients*

- It's essentially another hypothesis test designed to find which coefficients are useful

```
import statsmodels.api as sm

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

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

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2256.3950	13.055	-172.840	0.000	-2281.983	-2230.807
carat	7756.4362	14.066	551.423	0.000	7728.866	7784.006

Since the p-value is lower than our alpha of 0.05, we can conclude that carat is a good predictor of price

(the constant also has a p-value lower than 0.05, but we can generally ignore insignificant p-values for the intercept term)

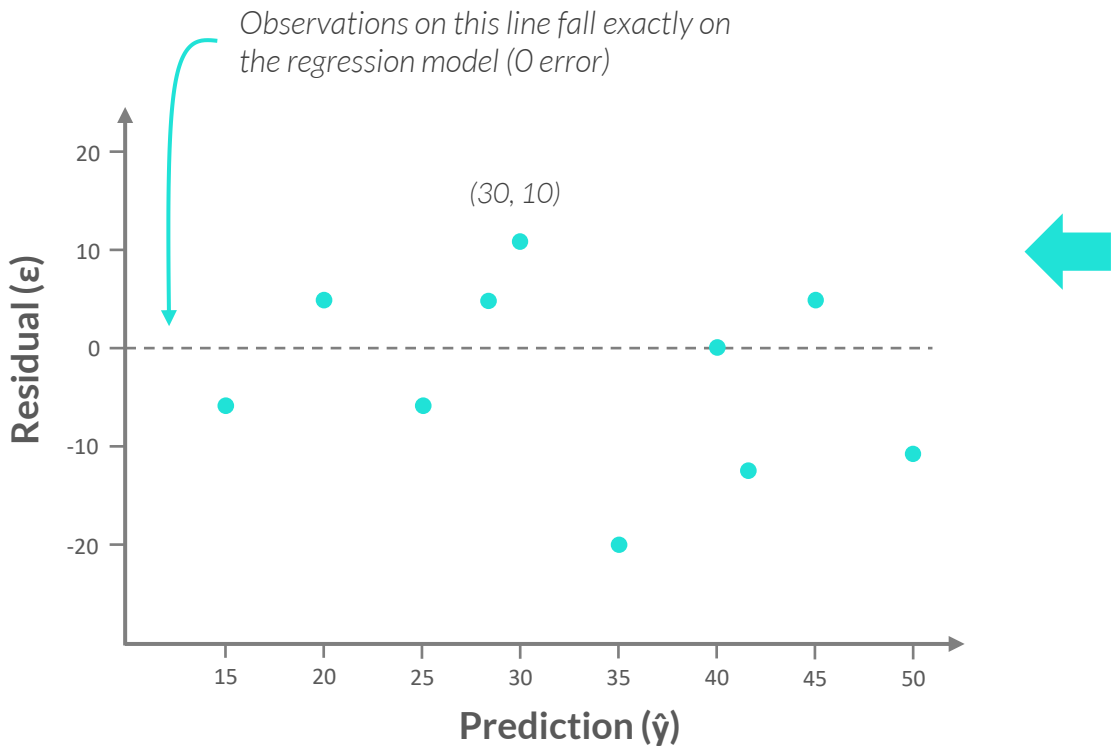
This will become more relevant when performing **variable selection** in multiple linear regression models (up next!)

RESIDUAL PLOTS

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

Residual plots show how well a model performs across the range of predictions

- Ideally, residual plots should be normally distributed around 0



x	y	ŷ	ε	ε^2
10	10	**15**	**-5**	25
20	25	**20**	**5**	25
30	20	**25**	**-5**	25
35	30	**27.5**	**2.5**	6.25
40	40	**30**	**10**	100
50	15	**35**	**-20**	400
60	40	**40**	**0**	0
65	30	**42.5**	**-12.5**	156.25
70	50	**45**	**5**	25
80	40	**50**	**-10**	100

RESIDUAL PLOTS

- Linear Regression Model
- Least Squared Error
- Regression in Python
- Making Predictions
- Evaluation Metrics

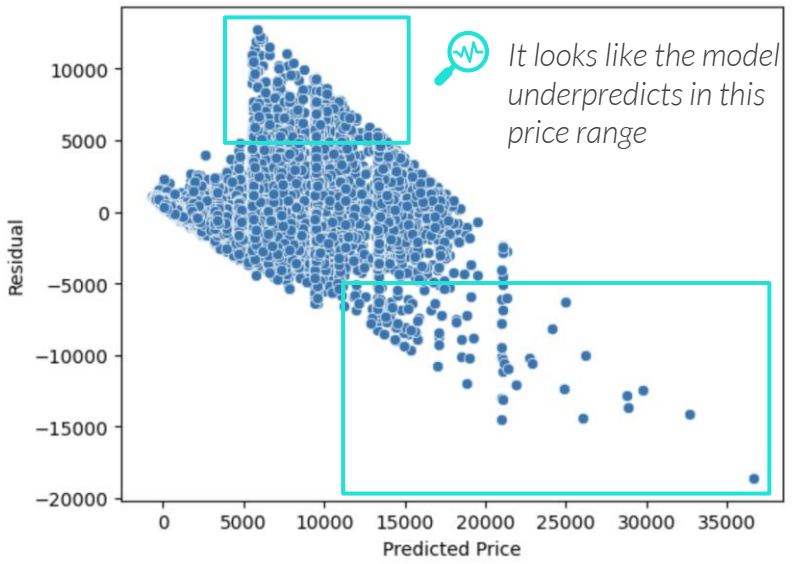
Residual plots show how well a model performs across the range of predictions

- Ideally, residual plots should show errors to be normally distributed around 0
- **model.resid** returns a series with the residuals *(actual value - predicted value)*

```
residuals = pd.DataFrame(
    {
        "Carat": diamonds["carat"],
        "Price": diamonds["price"],
        "Predicted Price": model.predict(),
        "Residual": model.resid
    }
)

residuals.head()
```

	Carat	Price	Predicted Price	Residual
0	0.40	666	846.179416	-180.179416
1	0.77	1940	3716.060795	-1776.060795
2	0.58	2036	2242.337925	-206.337925
3	0.41	705	923.743778	-218.743778
4	0.81	3250	4026.318242	-776.318242

```
sns.scatterplot(residuals, x="Predicted Price", y="Residual");
```



CASE STUDY: HEALTH INSURANCE

THE SITUATION

You've just been hired as a Data Science Intern for **Maven National Insurance**, a large private health insurance provider in the United States

THE ASSIGNMENT

The company is looking at updating their **insurance pricing model** and want you to start a new one from scratch using only a handful of variables

If you're successful, they can reduce the complexity of their model while maintaining its accuracy *(the data you've been provided has already been QA'd and cleaned)*

THE OBJECTIVES

1. Identify the strongest predictor of insurance prices using correlation
2. Build a simple linear regression model using this feature
3. Predict insurance prices for new customers using the model

ASSIGNMENT: SIMPLE LINEAR REGRESSION

NEW MESSAGE

June 30, 2023

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

Subject: **Price predictions**

Hi there!

Looks a few variables are promising!

Can you build a linear regression model with the target as "price" and the most correlated variable as the feature?

I'd also like to see a few predictions for common values of the feature that you selected.

Thanks!

02_simple_regression_assignments.ipynb

Reply Forward

Key Objectives

1. Build a simple linear regression model with "price" as the target and the most correlated variable as the feature
2. Interpret the model summary
3. Visualize the model residuals
4. Predict new "price" values with the model

KEY TAKEAWAYS

- A **simple linear regression** model is the line that best fits a scatterplot
  - *The line can be described using an equation with a slope and y-intercept, plus an error term*
  - *The least squared error method is used to find the line of best fit*

- Python uses the **statsmodels** & **scikit-learn** libraries to fit regression models
  - *Statsmodels is ideas if your goal is inference, while scikit-learn is optimal for prediction workflows*

- Linear regression models can be used to **predict new data**
  - *These predictions can be used to assess if our model performance holds on new data and help make decisions*

- The model summary contains metrics used to **evaluate the model**
  - *The model's R^2 value and the F-test's p-value help determine if the regression model is useful*

MULTIPLE LINEAR REGRESSION

MULTIPLE LINEAR REGRESSION

In this section we'll build **multiple linear regression** models using more than one feature, evaluate the model fit, perform variable selection, and compare models using error metrics

TOPICS WE'LL COVER:

- Multiple Regression
- Variable Selection
- Mean Error Metrics

GOALS FOR THIS SECTION:

- Learn how to fit and interpret multiple linear regression models in Python
- Walk through methods of performing variable selection, ensuring model variables add value
- Compare the predictive accuracy across models using mean error metrics like MAE and RMSE

MULTIPLE LINEAR REGRESSION MODEL

Multiple linear regression models use *multiple features* to predict the target

- In other words, it's the same linear regression model, but with additional "x" variables

SIMPLE LINEAR REGRESSION MODEL:

$$y = \beta_0 + \beta_1 x + \epsilon$$

MULTIPLE LINEAR REGRESSION MODEL:

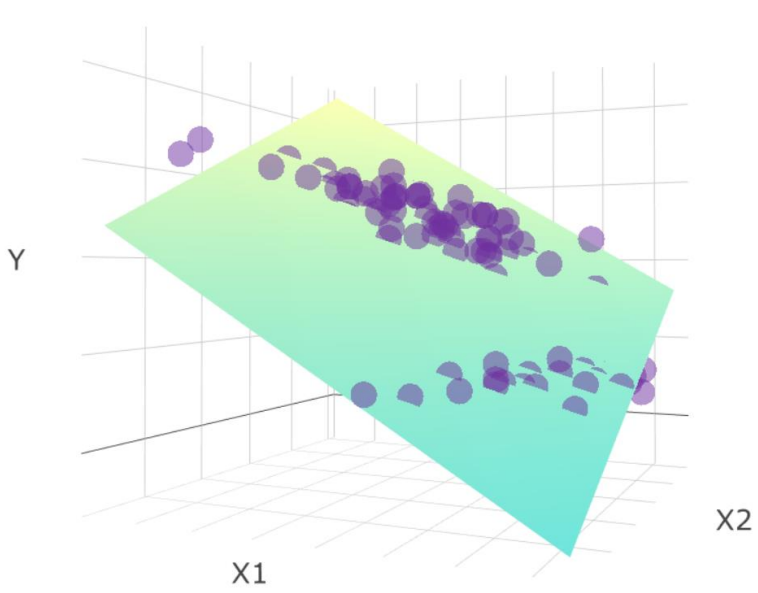
$$y = \beta_0 + \underbrace{\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k}_{} + \epsilon$$

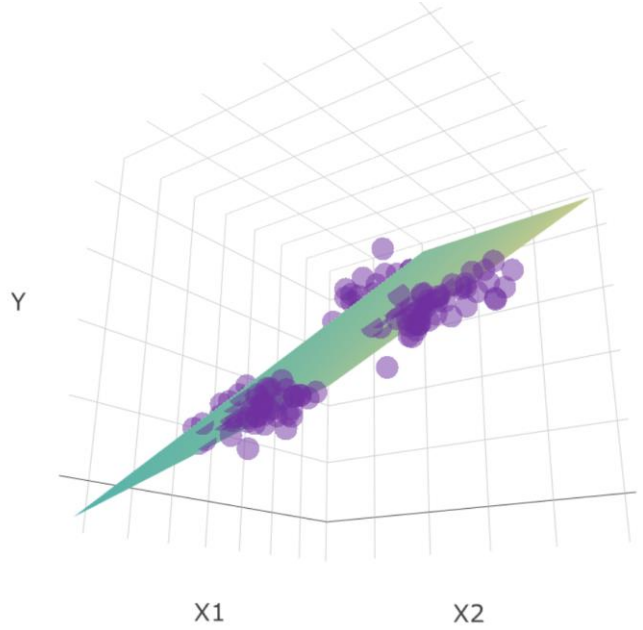
*Instead of just one "x", we have a **whole set of features** (and associated coefficients) to help predict the target (y)*

MULTIPLE LINEAR REGRESSION MODEL

- Multiple Regression
- Variable Selection
- Mean Error Metrics

To visualize how a multiple linear regression model works with 2 features, imagine fitting a plane *(instead of a line)* through a 3D scatterplot:





Multiple regression can scale well beyond 2 variables, but this is where visual analysis breaks down – and one reason why we need **machine learning**!

FITTING A MULTIPLE REGRESSION

To **fit a multiple regression**, include the desired features in the "X" DataFrame

- Multiple Regression
- Variable Selection
- Mean Error Metrics

```
diamonds.corr(numeric_only=True)
```

	carat	depth	table	price	x	y	z
carat	1.000000	0.028234	0.181602	0.921591	0.975093	0.951721	0.953387
depth	0.028234	1.000000	-0.295798	-0.010630	-0.025289	-0.029340	0.094927
table	0.181602	-0.295798	1.000000	0.127118	0.195333	0.183750	0.150915
price	0.921591	-0.010630	0.127118	1.000000	0.884433	0.865419	0.861249
x	0.975093	-0.025289	0.195333	0.884433	1.000000	0.974701	0.970771
y	0.951721	-0.029340	0.183750	0.865419	0.974701	1.000000	0.952005
z	0.953387	0.094927	0.150915	0.861249	0.970771	0.952005	1.000000

"carat" and "x" are the most correlated features with "price"

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

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

model.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.853
Model:	OLS	**Adj. R-squared:**	0.853
Method:	Least Squares	**F-statistic:**	1.570e+05
Date:	Thu, 03 Aug 2023	**Prob (F-statistic):**	0.00
Time:	10:05:44	**Log-Likelihood:**	-4.7201e+05
No. Observations:	53943	**AIC:**	9.440e+05
Df Residuals:	53940	**BIC:**	9.441e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	1738.1187	103.618	16.774	0.000	1535.026	1941.211
carat	1.013e+04	62.551	161.886	0.000	1e+04	1.02e+04
x	-1026.9048	26.432	-38.851	0.000	-1078.711	-975.099

Omnibus:	14014.880	**Durbin-Watson:**	2.001
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	317574.692
Skew:	0.717	**Prob(JB):**	0.00
Kurtosis:	14.800	**Cond. No.**	112.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

R^2 increased from 0.849

$p < 0.05$, so the model isn't useless

$p < 0.05$, so all variables are useful

"x" has a negative coefficient but a positive correlation (this is a concern we'll cover shortly)

INTERPRETING MULTIPLE REGRESSION

Interpreting multiple regression is like simple regression, with a slight twist

- Multiple Regression
- Variable Selection
- Mean Error Metrics

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

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

model.summary()
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	1738.1187	103.618	16.774	0.000	1535.026	1941.211
carat	1.013e+04	62.551	161.886	0.000	1e+04	1.02e+04
x	-1026.9048	26.432	-38.851	0.000	-1078.711	-975.099

$$\hat{y} = 1738 + 10130(carat) - 1027(x)$$

How do we interpret this?

- Technically, a 0-carat diamond with 0 length (x) is predicted to cost $1,738 dollars
- An increase in carat by 1 corresponds to a $10,130 increase in price, **holding x constant**
- An increase in x by 1 corresponds to a $1,027 decrease in price, **holding carat constant**

VARIABLE SELECTION

Multiple Regression

Variable Selection

Mean Error Metrics

Variable selection is a critical part of the modeling process that helps reduce complexity by only including features that help meet the model's goal

- Do new variables improve model accuracy? *(critical for prediction)*
- Is each variable statistically significant? *(critical for inference)*
- Do new variables make the model more challenging to interpret or explain to stakeholders?

EXAMPLE

Using all the possible features to predict diamond price

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.859
Model:	OLS	**Adj. R-squared:**	0.859

	coef	std err	t	P>|t|	[0.025	0.975]
const	2.085e+04	447.545	46.584	0.000	2e+04	2.17e+04
carat	1.069e+04	63.198	169.094	0.000	1.06e+04	1.08e+04
depth	-203.1414	5.504	-36.909	0.000	-213.929	-192.354
table	-102.4407	3.084	-33.217	0.000	-108.485	-96.396
x	-1315.7397	43.069	-30.550	0.000	-1400.155	-1231.324
y	66.3300	25.522	2.599	0.009	16.306	116.354
z	41.6285	44.303	0.940	0.347	-45.207	128.464

R^2 improved to .859 compared to the simple regression model (.849)
If the goal is inference, a single variable model may be a good choice!

"x" still has a negative coefficient
This helps R^2 but makes the model hard to explain

"z" has a p-value greater than alpha (0.347 > 0.05)
We should drop this variable from the model

ADJUSTED R-SQUARED

Multiple Regression

Variable Selection

Mean Error Metrics

A criticism of r-squared is that it will never decrease as new variables are added

Adjusted r-squared corrects this by penalizing new variables added to a model

- This measure has no meaning whatsoever, but it helps as a variable selection tool

EXAMPLE | *Adding a random column to a sample of 100 diamonds*

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.782
Model:	OLS	**Adj. R-squared:**	0.780
Method:	Least Squares	**F-statistic:**	2501

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2517.2645	355.136	-7.088	0.000	-3222.020	-1812.509
carat	8631.6433	459.978	18.765	0.000	7718.831	9544.455

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.784
Model:	OLS	**Adj. R-squared:**	0.780
Method:	Least Squares	**F-statistic:**	176.2

R^2 increased but adjusted R^2 didn't

	coef	std err	t	P>|t|	[0.025	0.975]
const	-2263.3000	452.073	-5.006	0.000	-3160.540	-1366.060
carat	8673.9033	462.726	18.745	0.000	7755.520	9592.287
random	-5.5127	6.063	-0.909	0.366	-17.547	6.521

In this case, the p-value could also tell us to remove this variable (other times, a variable can be significant and lower adjusted R^2)

ASSIGNMENT: MULTIPLE LINEAR REGRESSION

NEW MESSAGE

July 2, 2023

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

Subject: **Multiple Regression Model**

Hi there!

Let's include more features in the model.

Can you start by adding all of the PC component variables, ram, screen, hd, and speed?

Then take a look at including trend, which accounts for cost decreases over time, and ads, which measure marketing spend on each model.

Thanks!

03_multiple_regression_assignments.ipynb

Reply Forward

Key Objectives

1. Build and interpret two multiple linear regression models
2. Evaluate model fit and coefficient values for both models

MEAN ERROR METRICS

Multiple Regression

Variable Selection

Mean Error Metrics

Mean error metrics measure how well your regression model *fits in in the units of our target*, as opposed to how well it explains variance (like R-Squared)

- The most common are **Mean Absolute Error** (MAE) and **Root Mean Squared Error** (RMSE)
- They are used to compare model fit across models *(the lower the better!)*

MAE

*Average of the **absolute** distance between actual & predicted values*

$$\frac{\sum_i |y_i - \hat{y}_i|}{n}$$

MSE

*Average of the **squared** distance between actual & predicted values*

$$\frac{\sum_i (y_i - \hat{y}_i)^2}{n}$$

RMSE

***Square root** of Mean Squared Error, to return to the target's units (like MAE)*

$$\sqrt{\frac{\sum_i (y_i - \hat{y}_i)^2}{n}}$$

RMSE is more sensitive to large outliers, so it is preferred over MAE in situations where they are particularly undesirable

MEAN ERROR METRICS

Multiple Regression

Variable Selection

Mean Error Metrics

You can use **sklearn.metrics** to calculate MAE and MSE for your model

- sklearn.metrics.**mean_absolute_error**(y_actual, y_predicted)
- sklearn.metrics.**mean_squared_error**(y_actual, y_predicted)

```
from sklearn.metrics import mean_absolute_error as mae
from sklearn.metrics import mean_squared_error as rmse

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

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

print(f"MAE: {mae(y, model.predict())}")
print(f"RMSE {rmse(y, model.predict(), squared=False)}")
```

```
MAE: 1007.4339350399421
RMSE 1548.4940983743945
```

*This returns **RMSE** instead of MSE*

The simple linear regression model has an average prediction error of ~$1,000

The outliers in the dataset are making RMSE around 50% larger than MAE

```
x = sm.add_constant(diamonds[["carat", "depth", "table", "x", "y"]])
y = diamonds["price"]

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

print(f"MAE: {mae(model.predict(x), y)}")
print(f"RMSE: {rmse(model.predict(x), y, squared=False)}")
```

```
MAE: 889.2135691786077
RMSE: 1496.8311707830426
```

RMSE will always be bigger than MAE

The multiple linear regression model performs better across both metrics

ASSIGNMENT: MEAN ERROR METRICS

NEW MESSAGE

July 3, 2023

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

Subject: **Error Metrics**

Hi there!

Thanks for building those models, they're definitely showing some potential!

Can you calculate MAE and RMSE for the model with all variables and our simple regression model that just included RAM?

This will help me communicate the improvement in fit better to our stakeholders.

Thanks!

03_multiple_regression_assignments.ipynb

Reply Forward

Key Objectives

1. Calculate MAE and RMSE for fitted regression models

KEY TAKEAWAYS

Multiple linear regression models use **multiple features** to predict the target

- *Each new feature comes with an associated coefficient that forms part of the regression equation*

Variable selection methods help you identify valuable features for the model

- *Coefficients with p-values greater than alpha (0.05) indicate that a coefficient isn't significantly different than 0*
- *Contrary to R^2, the adjusted R^2 metric penalizes new variables added to a model*

Mean error metrics let you compare predictive accuracy across models

- *Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) quantify a model's inaccuracy in the target's units*
- *RMSE is more sensitive to large errors, so it is preferred in situations where large errors are undesirable*

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

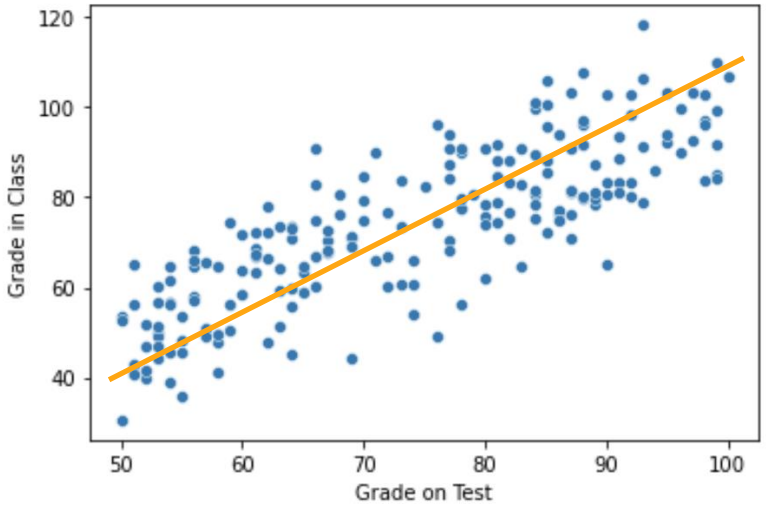
- Assumptions Overview
- Linearity
- Independence of Errors
- Normality of Errors
- No Perfect Multicollinearity
- Equal Variance of Errors
- Outliers & Influence

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

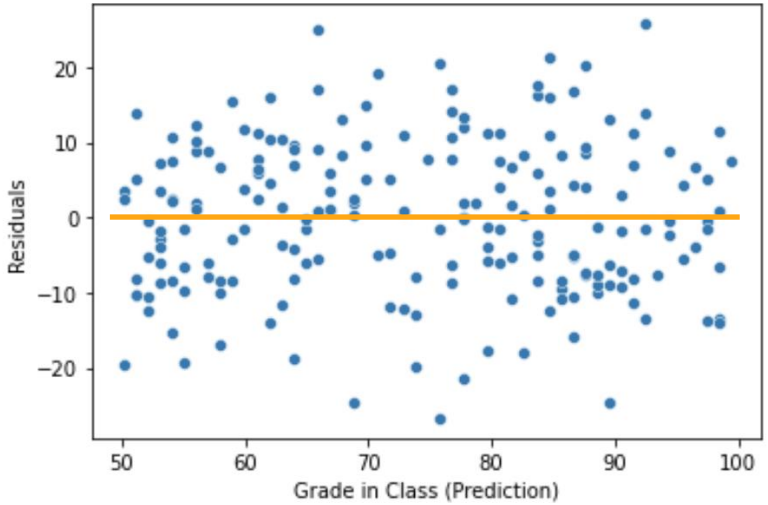
- If this assumption is violated, it means the model isn't capturing the underlying relationship between the variables, which could lead to inaccurate predictions

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

Ideal Scatterplot



Ideal Residual Plot



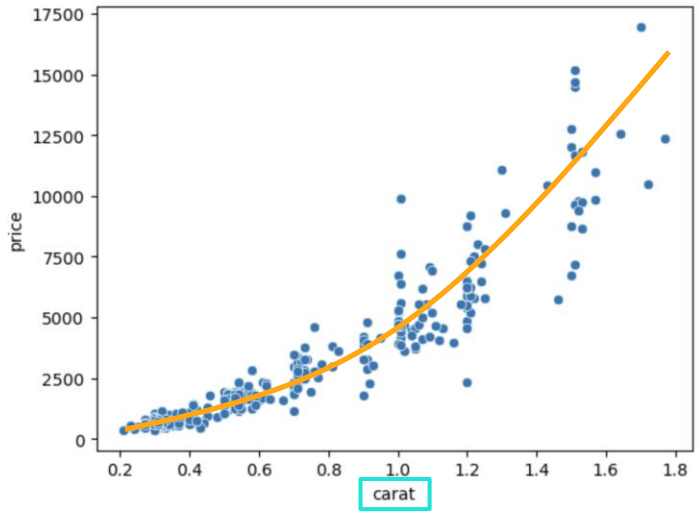
LINEARITY

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)

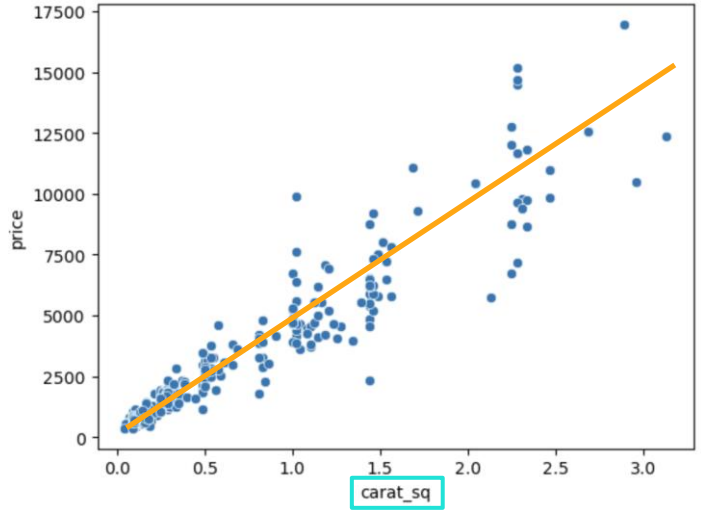
```
d_sample = diamonds.sample(300)

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

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

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

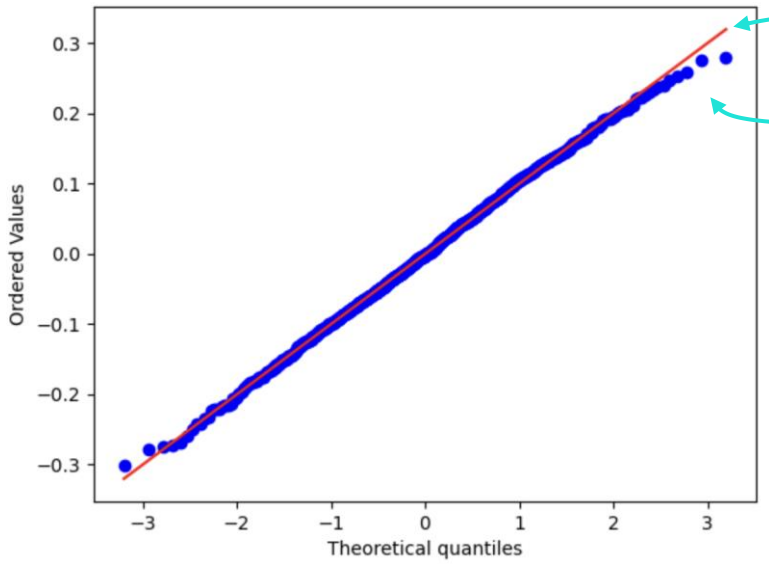
Randomizing the order brings things back to normal

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

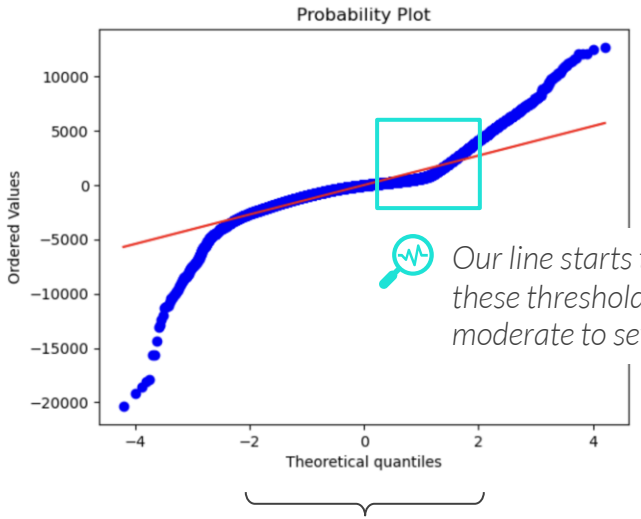
NORMALITY OF ERRORS

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);
```

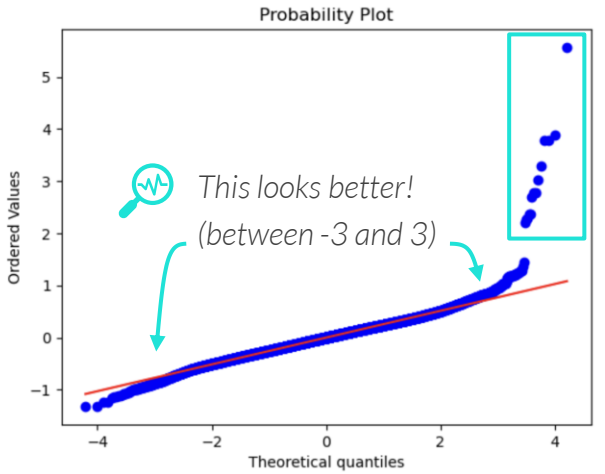


Our line starts to curve in between these thresholds, which indicates moderate to severe non-normality

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!)

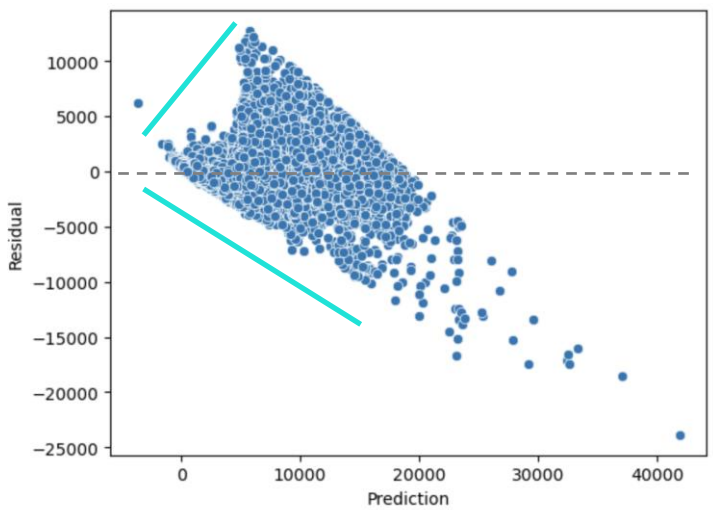
EQUAL VARIANCE OF ERRORS

Equal variance of errors assumes the residuals are consistent across predictions

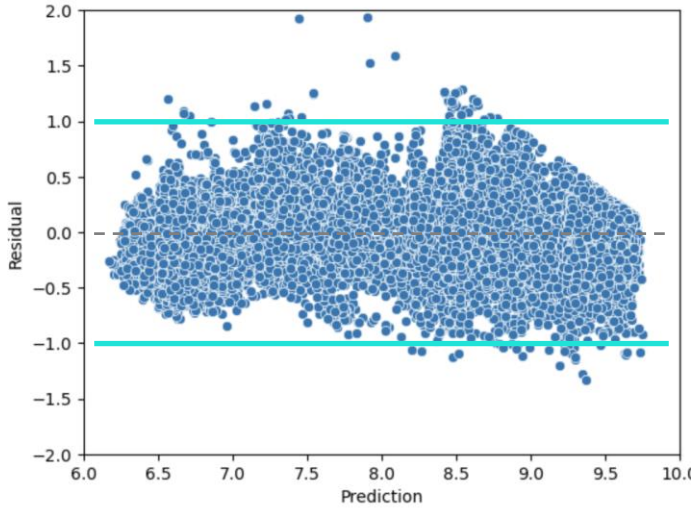
- In other words, average error should stay roughly the same across the range of the target
- Equal variance is known as **homoskedasticity**, and non-equal variance is **heteroskedasticity**

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

Heteroskedasticity
(original regression model)



Homoskedasticity
(after fixing violated assumptions)



Our model predicts much better now!

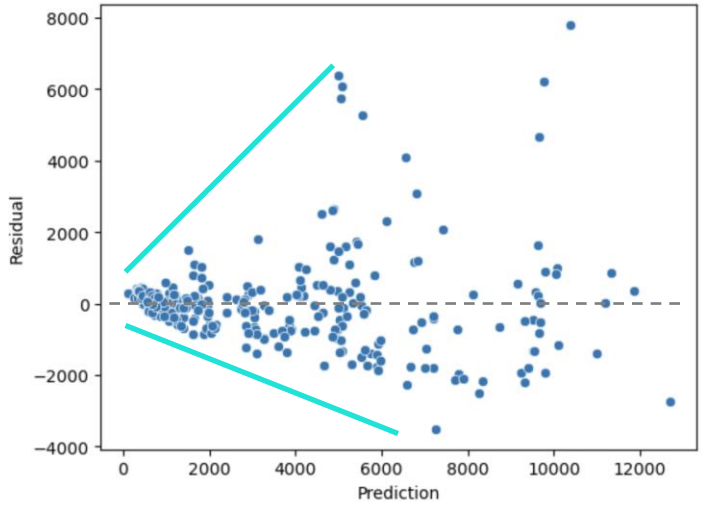
EQUAL VARIANCE OF ERRORS

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

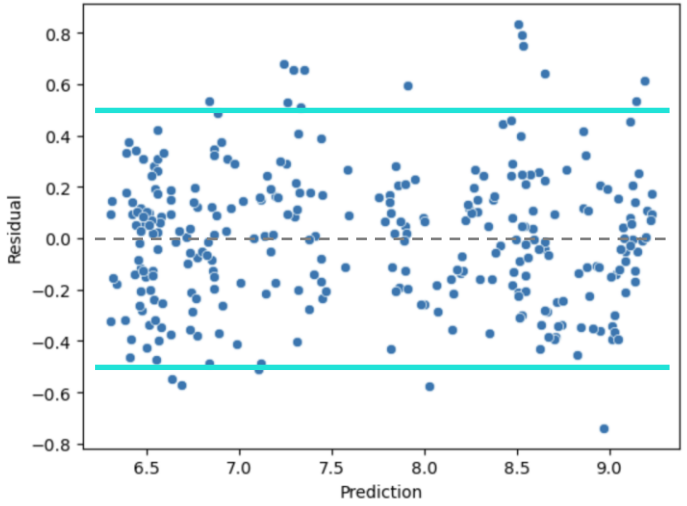
```
X = sm.add_constant(d_sample.loc[:, features])
y = d_sample["price"]

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*

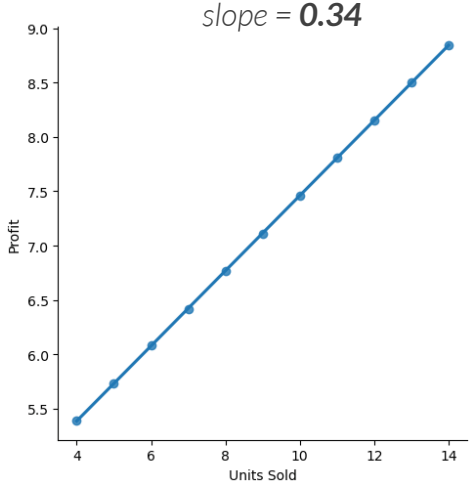
OUTLIERS & INFLUENCE

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**

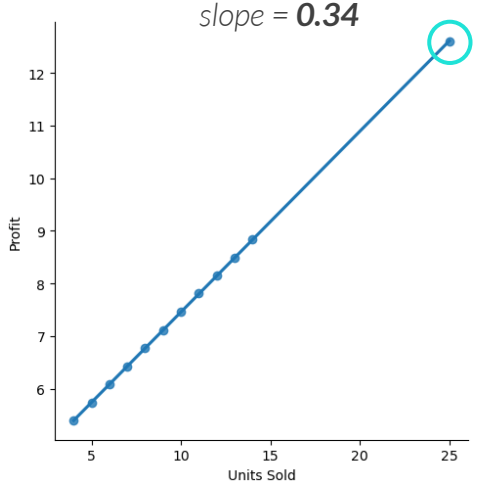
No Outliers

*slope = **0.34***



Non-influential Outlier

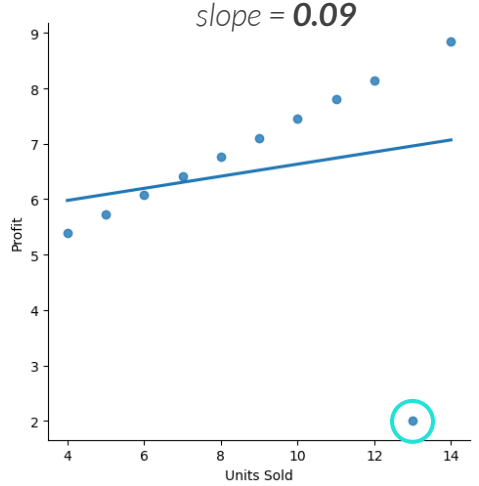
*slope = **0.34***



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)*

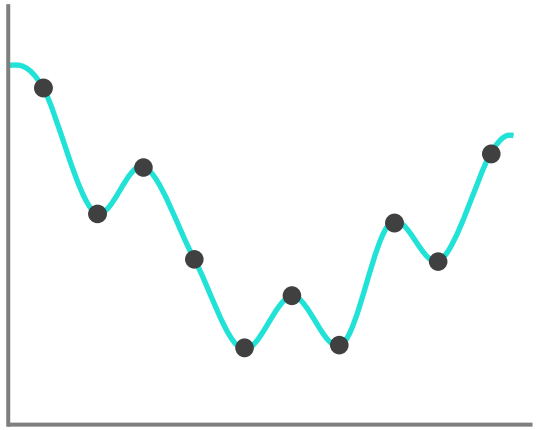
***4 inputs**:*

- *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

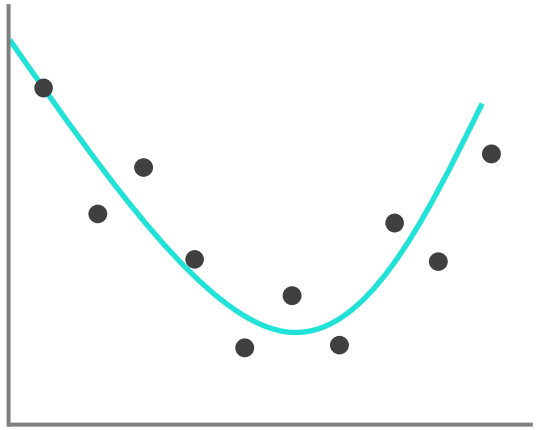
Data splitting is primarily used to avoid **overfitting**, which is when a model predicts known *(Training)* data very well but unknown *(Test)* data poorly

- 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



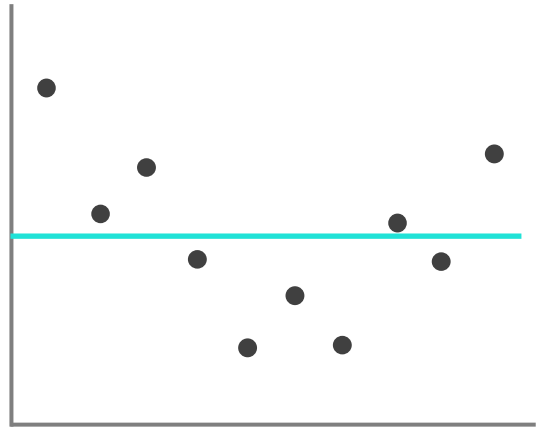
OVERFIT model

- *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!

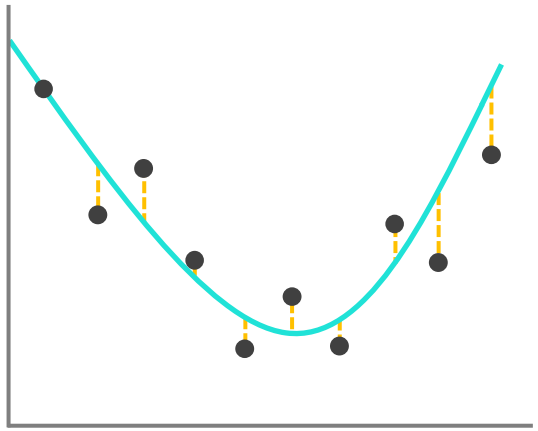
THE BIAS-VARIANCE TRADEOFF

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

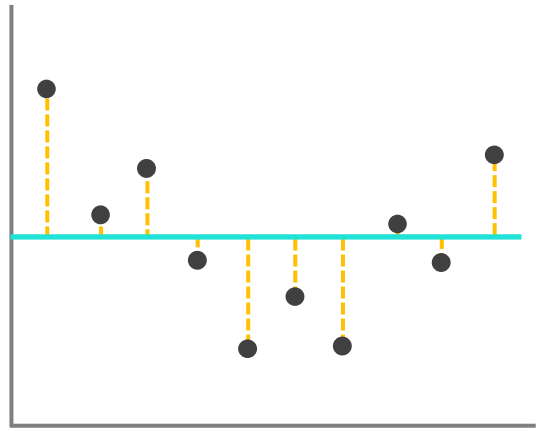
OVERFIT model

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



WELL-FIT model

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



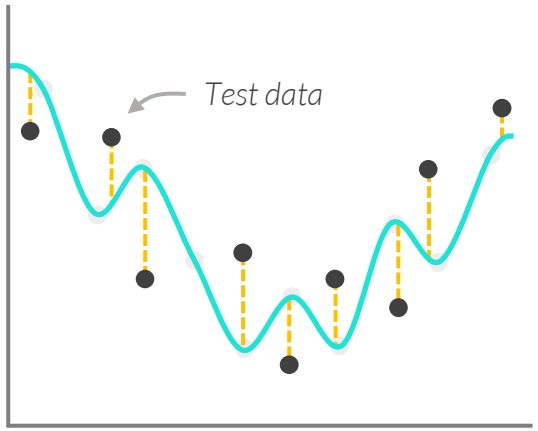
UNDERFIT model

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

THE BIAS-VARIANCE TRADEOFF

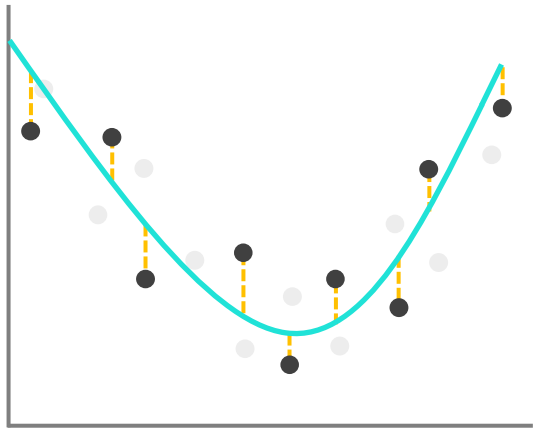
When splitting data for regression, there are two types of errors that can occur:

- **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

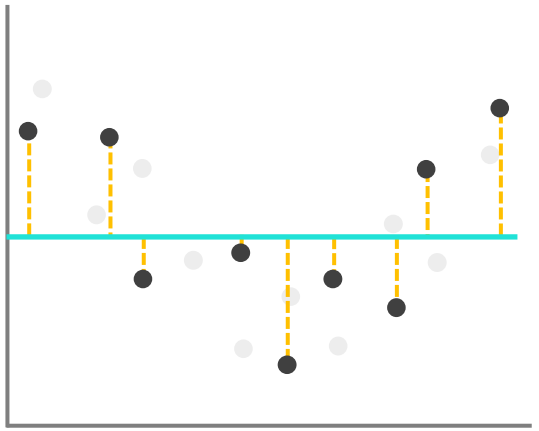
- ***Low bias*** *(no error)*
- ***High variance*** *(much more error)*



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

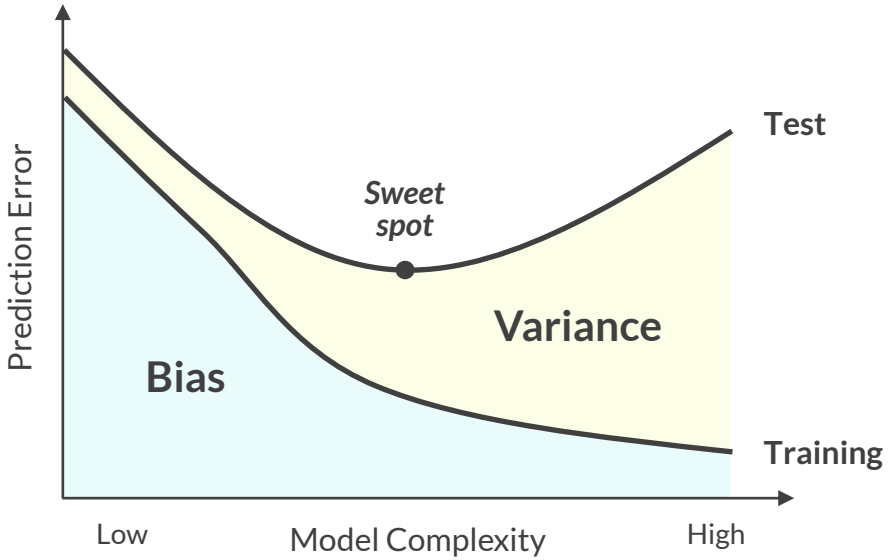
- Model Scoring Steps
- Data Splitting
- Overfitting
- Bias-Variance Tradeoff
- Validation
- Cross Validation

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

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	Model #2	Model #3
3 variables	*20 variables*	*3 variables + 1 squared term*
• Training R^2 = **0.67** • Validation R^2 = **0.66**	• Training R^2 = **0.89** • Validation R^2 = **0.71**	• Training R^2 = **0.81** • Validation R^2 = **0.79**
Both R^2 values seem relatively low with a small gap between them (underfit – high bias, low variance)	*Training R^2 improved but the gap between them grew considerably (overfit – low bias, high variance)*	*Training R^2 dropped a bit but the gap between them closed down (best fit – balanced bias & variance)*
Action: Add complexity	**Action**: Simplify	**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))}")
```

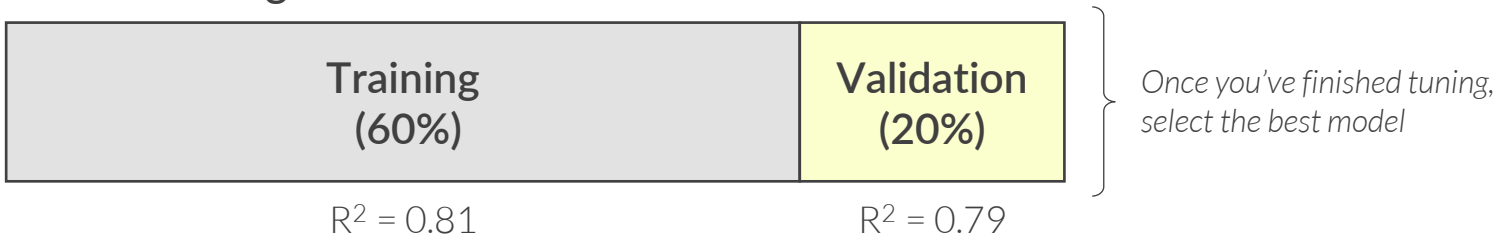
```
Training R2: 0.856811264298398
Validation R2: 0.8262889176864605
```

MODEL SCORING

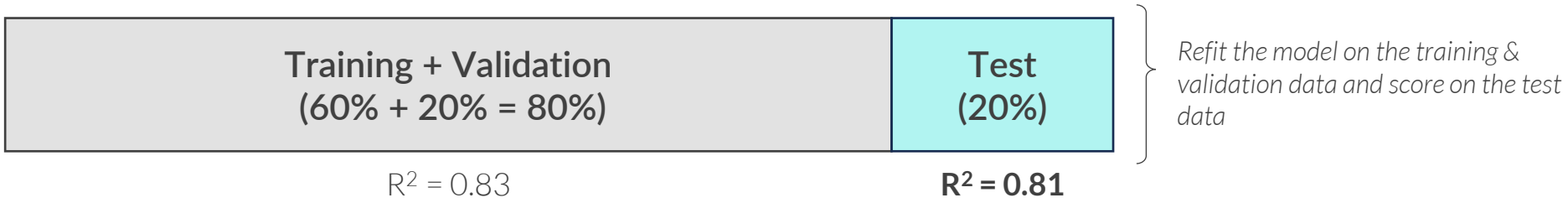
Once you've tuned and selected the best model, you need to refit the model on both the training and validation data, then **score the model** on the test data

- 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

Math Calculations

Category Mappings

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

	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

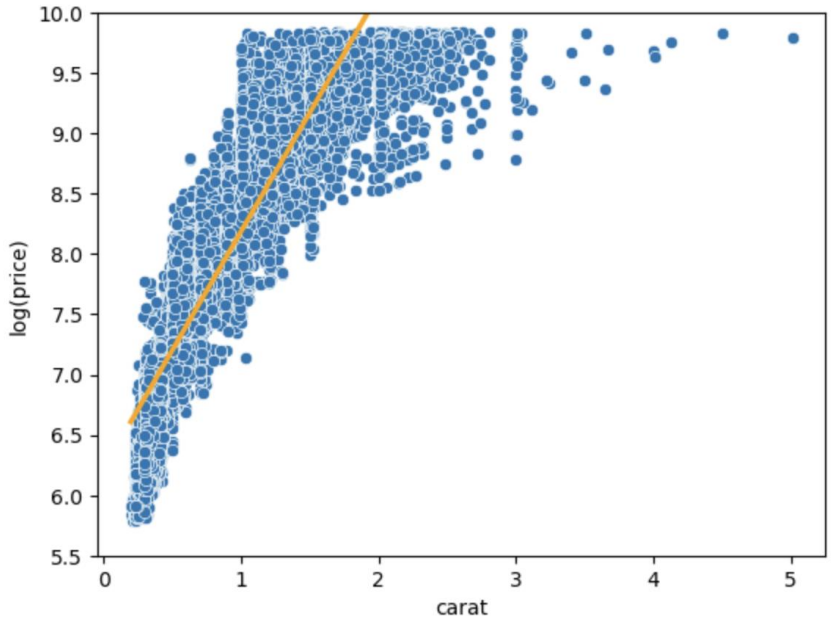
Category Mappings

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

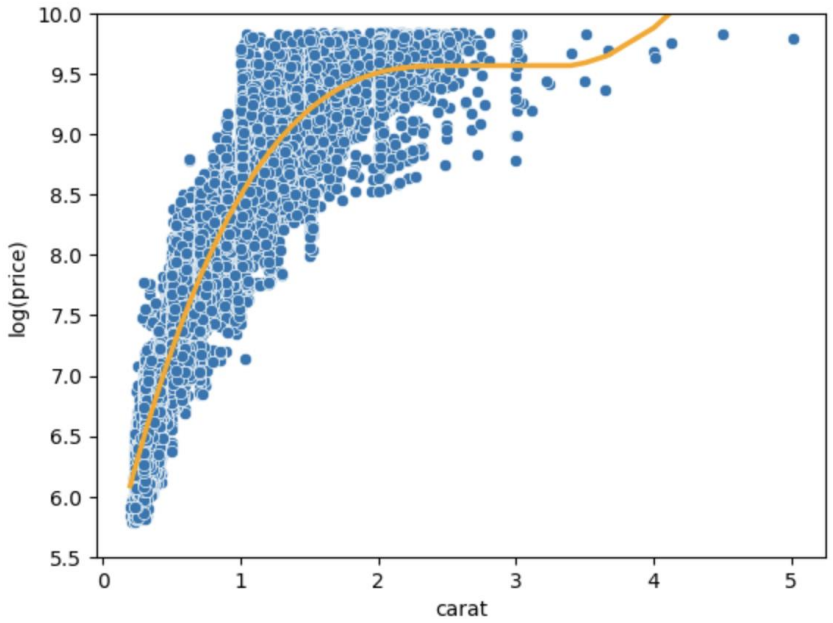
Adding **polynomial terms** (x^2, x^3, etc.) to regression models can improve fit when spotting "curved" feature-target relationships during EDA

- 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

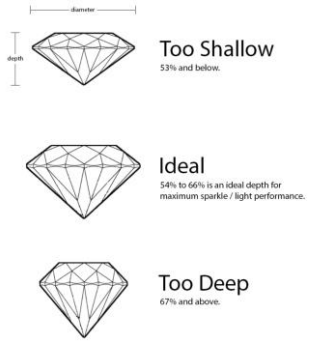
Carat weight is our strongest variable alone, but can we engineer some stronger features?

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

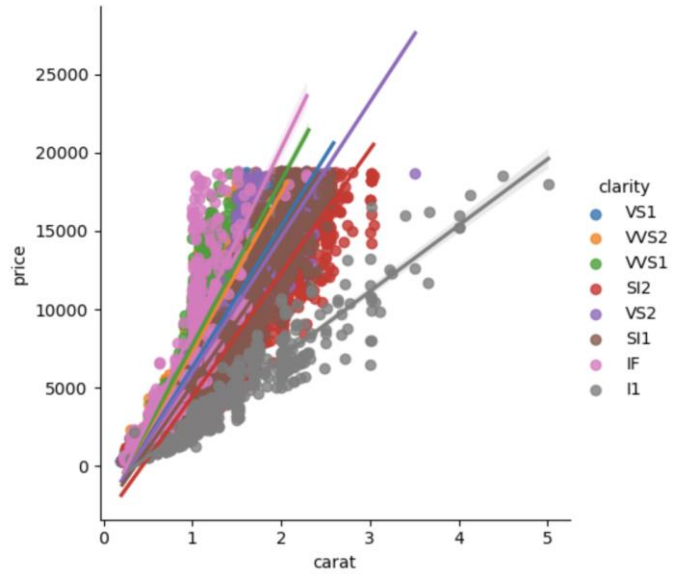
Feature Engineering

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

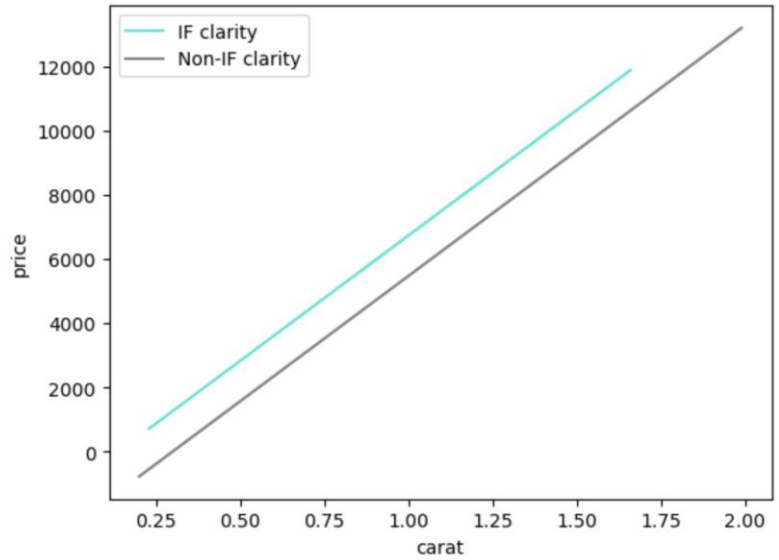
Interpreting coefficients for binary columns:

```
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"

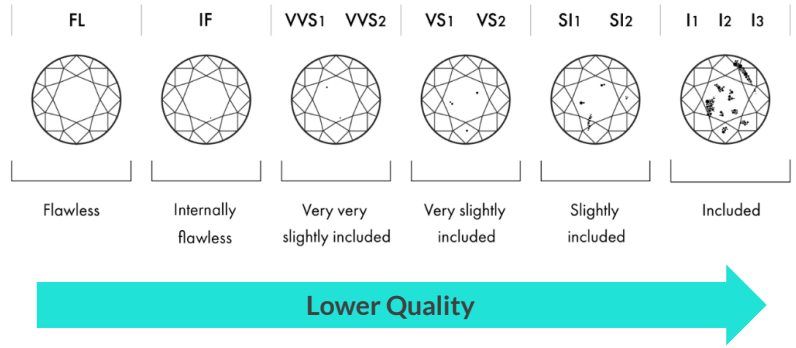
Interpreting coefficients for dummy variables:

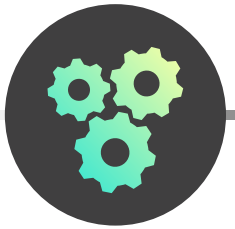
	coef	std err	t	P>|t|	[0.025	0.975]
const	-6959.7458	56.310	-123.596	0.000	-7070.115	-6849.376
carat	8436.8446	14.191	594.519	0.000	8409.030	8464.659
clarity_IF	5571.4246	64.005	87.047	0.000	5445.973	5696.876
clarity_SI1	3783.4198	55.091	68.676	0.000	3675.440	3891.399
clarity_SI2	2925.0475	55.439	52.761	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

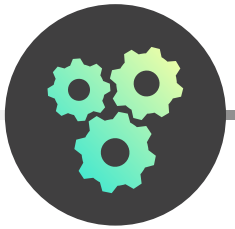
- After binning, create dummy variables for the groups *(which should be fewer than before)*

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

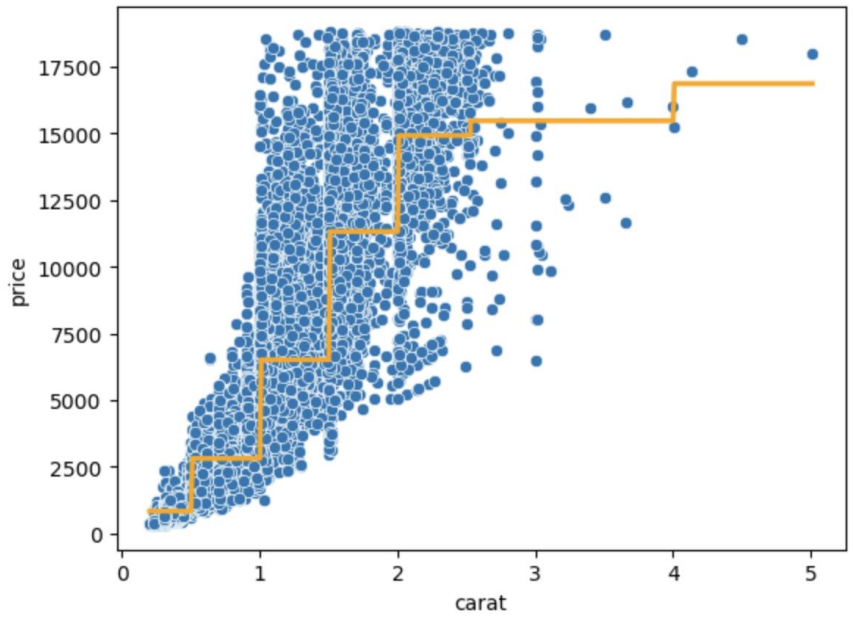
Category Mappings

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*

PROJECT 1: LINEAR REGRESSION

PROJECT DATA: SAN FRANCISCO RENT PRICES

```
apartments_df.head()
```

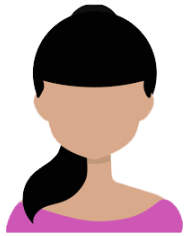
	price	sqft	beds	bath	laundry	pets	housing_type	parking	hood_district
0	6800	1600.0	2.0	2.0	(a) in-unit	(d) no pets	(c) multi	(b) protected	7.0
1	3500	550.0	1.0	1.0	(a) in-unit	(a) both	(c) multi	(b) protected	7.0
2	5100	1300.0	2.0	1.0	(a) in-unit	(a) both	(c) multi	(d) no parking	7.0
3	9000	3500.0	3.0	2.5	(a) in-unit	(d) no pets	(c) multi	(b) protected	7.0
4	3100	561.0	1.0	1.0	(c) no laundry	(a) both	(c) multi	(d) no parking	7.0

```
apartments_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 989 entries, 0 to 988
Data columns (total 9 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   price          989 non-null    int64
 1   sqft           989 non-null    float64
 2   beds           989 non-null    float64
 3   bath           989 non-null    float64
 4   laundry        989 non-null    object
 5   pets           989 non-null    object
 6   housing_type   989 non-null    object
 7   parking        989 non-null    object
 8   hood_district  989 non-null    float64
dtypes: float64(4), int64(1), object(4)
memory usage: 69.7+ KB
```

MAVEN CONSULTING GROUP

PROJECT BRIEF: LINEAR REGRESSION



NEW MESSAGE

July 20, 2023

From: **Cathy Coefficient** *(Data Science Lead)*

Subject: **Apartment Rental Prices**

Hi there,

Cam has told me some great things about your work. We have a new client that has a modelling project they need help with.

The client works in the real estate industry in San Francisco and wants to understand the key factors affecting rental prices. More importantly, they hope be able to use your model to predict an appropriate price range for the apartments they build in the city.

You'll find more info in the attached notebook, thanks!

07_regression_modelling_project.ipynb

Reply Forward

Key Objectives

1. Perform EDA on a modelling dataset
2. Split your data and choose a validation framework
3. Fit and tune a linear regression model by checking model assumptions and performing feature engineering
4. Interpret a linear regression model

MAVEN CONSULTING GROUP

REGULARIZED REGRESSION

REGULARIZED REGRESSION

In this section we'll cover **regularized regression models,** also known as "penalized" regression models, which focus on reducing model variance to improve predictive accuracy

TOPICS WE'LL COVER:

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

GOALS FOR THIS SECTION:

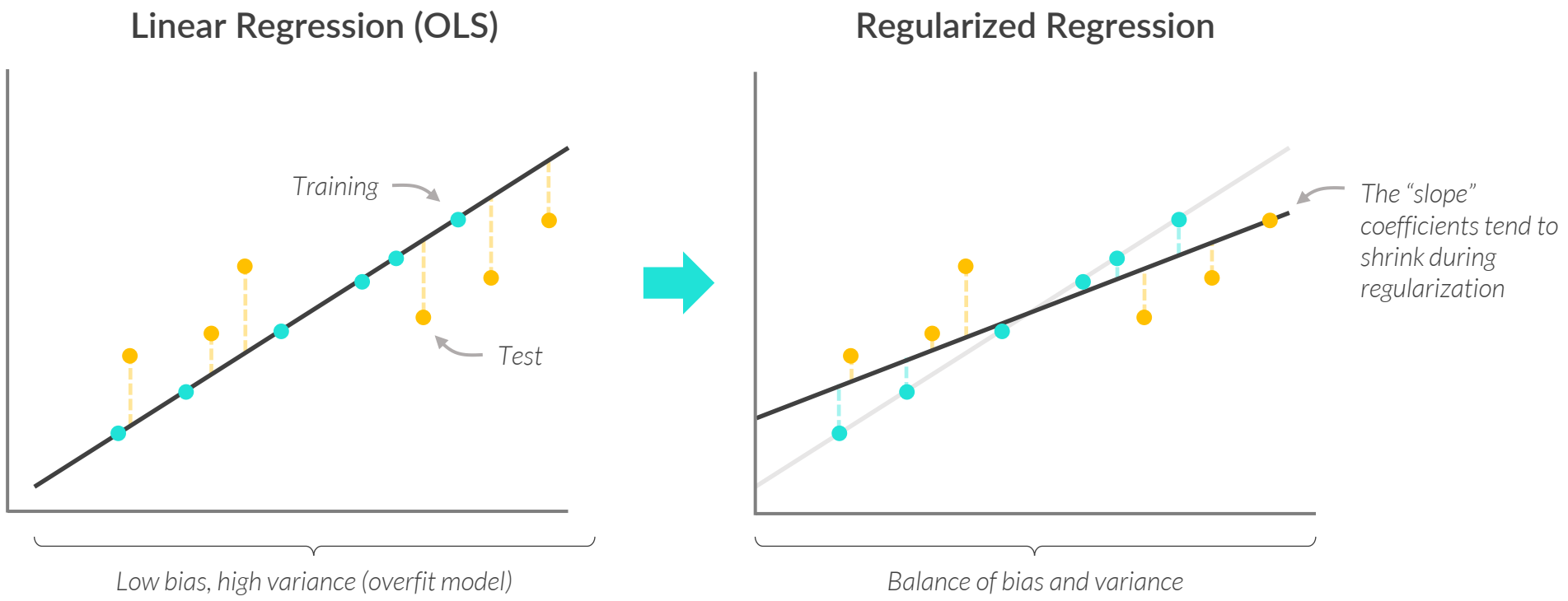
- Understand the difference between linear regression and regularized regression models
- Introduce the cost function and the impact of the regularization term
- Review the steps for fitting and training regularized regression models, including standardization and hyperparameter tuning
- Discuss the similarities and differences between Ridge Regression, Lasso Regression and Elastic Net

REGULARIZED REGRESSION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Regularized regression models add bias by NOT choosing the line of best fit for the training data with the purpose of reducing the variance in the test data

- This helps reduce overfitting and leads to better predictions *(especially in smaller data sets)*



THE COST FUNCTION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Regularized regression models modify the "line of best fit" by adding additional elements to the **cost function** used to find the optimal model

In linear regression, the cost function is just the sum squared error (SSE):

$$J = SSE$$

*The **cost** you're trying to minimize*

*The **sum of squared error***

In regularized regression, you add a regularization term controlled by alpha:

$$J = SSE + \alpha R$$

*The **alpha** term, which controls the impact of the regularization term*

*The **regularization** term*

TYPES OF REGULARIZED REGRESSION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Different **types of regularized regression** have different regularization equations

$$J = SSE + \alpha R$$

Ridge Regression

*Sum of the **squared** coefficient values*

$$\sum_{j=1}^{p} \beta_j^2$$

Lasso Regression

*Sum of the **absolute** coefficient values*

$$\sum_{j=1}^{p} |\beta_j|$$

Elastic Net Regression

*Sum of the **ridge & lasso regularization terms**, weighted by lambda (λ)*

$$\lambda \sum_{j=1}^{p} \beta_j^2 + (1 - \lambda) \sum_{j=1}^{p} |\beta_j|$$

RIDGE REGRESSION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Ridge regression adds a regularization term, or complexity penalty, that's equal to the sum of the model's coefficients (β) squared, also known as an L2 penalty

- The L2 penalty shrinks the coefficients towards 0 *(but never to 0 exactly)*
- The higher alpha, the more the coefficients get shrunk
- The features that reduce SSE the most will shrink at a slower rate than less useful ones

$$J = SSE + \alpha \sum_{j=1}^{p} \beta_j^2$$

When fitting a ridge regression, the model is trying to minimize this cost function by both:

- *Minimizing training error with SSE*
- *Keeping the coefficient values small*

The idea is to **incorporate the bias-variance tradeoff** directly into the algorithm!

We want to reduce the sum of squared error, but also constrain the magnitude of the coefficients to prevent overfitting'

Because of this, ridge regression tends to produce much more accurate models when working with multicollinear features.

RIDGE REGRESSION WORKFLOW

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

The **ridge regression workflow** has a few additional steps that are required:

Linear regression:

1. Fit a linear regression model on the training data and score on the validation data
2. Tune the model by adding or removing inputs
3. Fit the model on the train & validation data and score on the test data

Ridge regression:

1. **Standardize all inputs into the model**
2. Fit a ridge regression model on the training data and score on the validation data
3. Tune the model by adding or removing inputs and **modifying the α hyperparameter**
4. Fit the model on the train & validation data and score on the test data

STANDARDIZATION

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

Standardization sets all input features on the same scale by transforming all values to have a mean of 0 and a standard deviation of 1

- Ridge regression is fit using the size of the coefficients, so they need to have the same units
- The **StandardScaler()** function from scikit-learn is the preferred way to do this

```
from sklearn.preprocessing import StandardScaler

std = StandardScaler()
X_tr = std.fit_transform(X_train.values)
X_val = std.transform(X_valid.values)
X_te = std.transform(X_test.values)
```

*Use **.fit_transform** on the training data to calculate the mean & standard deviation and apply the standardization*

*Use **.transform** on the validation & test data to apply the same standardization*

As a best practice, only retrieve the mean & standard deviation from the **training data**

Doing this before splitting the data would be collecting information from the test data and using it for training the model, which will lead to inflated test performance

STANDARDIZATION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Standardization sets all input features on the same scale by transforming all values to have a mean of 0 and a standard deviation of 1

- Ridge regression is fit using the size of the coefficients, so they need to have the same units
- The **StandardScaler()** function from scikit-learn is the preferred way to do this

```
X_train["carat"].mean()
```

```
0.7923543951799783
```

```
X_train["carat"].std()
```

```
0.46527357460975394
```

```
X_train["carat"].head()
```

```
10018    2.03
5076     0.76
8512     0.54
681      0.29
5874     0.56
Name: carat, dtype: float64
```

```
pd.DataFrame(X_tr, columns=X.columns).head()["carat"]
```

```
0    2.660244
1   -0.069544
2   -0.542420
3   -1.079780
4   -0.499432
Name: carat, dtype: float64
```

A 0.29-carat diamond is 1.08 standard deviations (0.46) smaller than the mean carat weight (0.79)

FITTING A RIDGE REGRESSION

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

You can fit a ridge regression with the **Ridge()** function in sklearn.linear_model

- Use the **.coef_** attribute to return the model coefficients

```
from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=1).fit(X_tr, y_train)

print(
    f"Train Score: {round(ridge_model.score(X_tr, y_train), 4)} "
    f"Valid Score: {round(ridge_model.score(X_val, y_valid), 4)} "
)
```

Fit on the standardized features and alpha = 1

Calculate R^2 for the training and validation sets

```
Train Score: 0.8697 Valid Score: 0.3325
```

The large gap indicates the model is overfit, so we should try different values for alpha

```
list(zip(X.columns, ridge_model.coef_))
```

```
[('carat', 5865.414053107809),
 ('carat2', -267.1919322677817),
 ('x', -2193.9527084181086),
 ('y', 3374.5590294124545),
 ('z', -3148.560785227803),
 ('depth', 50.660046305595486),
 ('table', -226.6380042668729)]
```

Note that the magnitudes differ from previous estimates due to standardization

TUNING ALPHA

Regularization

Ridge Regression

Lasso Regression

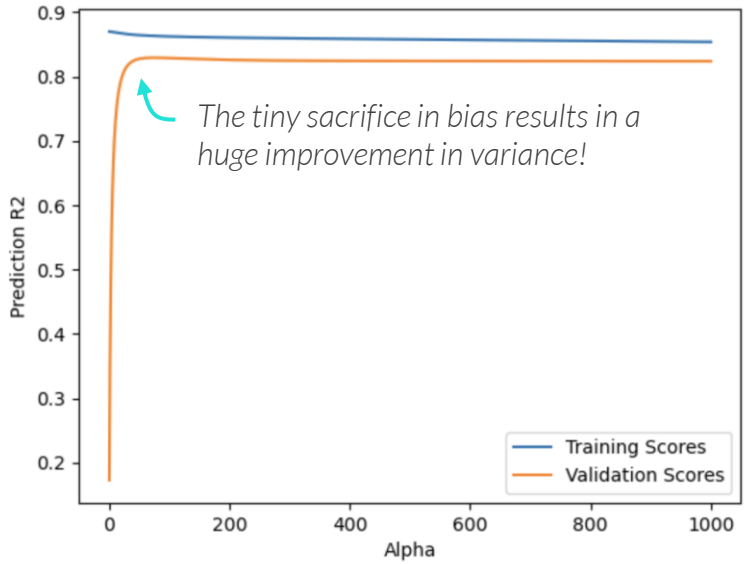
Elastic Net Regression

We need to **test different values of α** to see how it affects model performance

- In sklearn, the default value is 1, but this should be tuned
- The ideal value is different for every model and can be found via trial & error with validation

```
# generate 200 alphas between .001 and 1000
n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

train_scores = []
val_scores = []

# Loop through alphas and fit Ridge for each value
for alpha in alphas:
    ridge_model = Ridge(alpha=alpha).fit(X_tr, y_train)
    train_scores.append(ridge_model.score(X_tr, y_train))
    val_scores.append(ridge_model.score(X_val, y_valid))

# Find alpha with highest validation score
pd.DataFrame({
    "alpha": alphas,
    "training": train_scores,
    "validation": val_scores
}).sort_values("validation", ascending=False).head(1)
```

	alpha	training	validation
161	71.49429	0.863131	0.829146



The training score went down slightly from 0.869, but validation improved significantly from 0.332 with an alpha of 71.49 (instead of 1)

TUNING ALPHA

Regularization

Ridge Regression

Lasso Regression

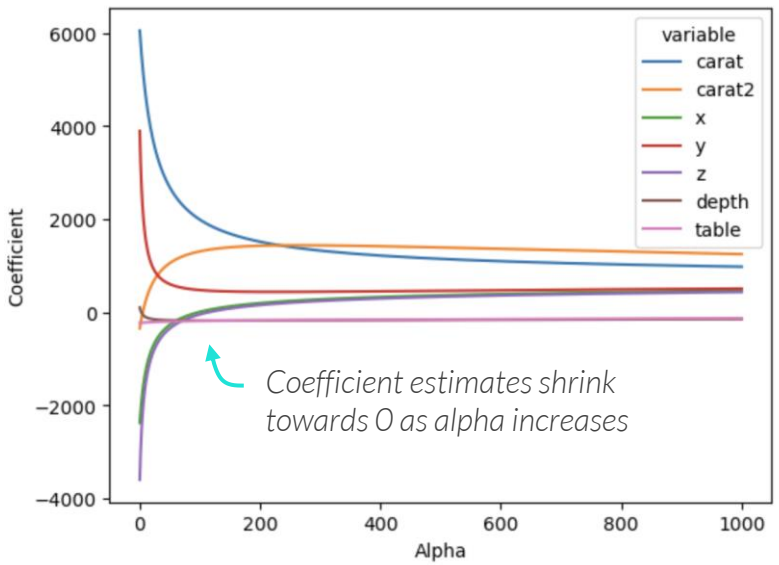
Elastic Net Regression

We need to **test different values of α** to see how it affects model performance

- In sklearn, the default value is 1, but this should be tuned
- The ideal value is different for every model and can be found via trial & error with validation

```
# generate 200 alphas between .001 and 1000
n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

train_scores = []
val_scores = []

# Loop through alphas and fit Ridge for each value
for alpha in alphas:
    ridge_model = Ridge(alpha=alpha).fit(X_tr, y_train)
    train_scores.append(ridge_model.score(X_tr, y_train))
    val_scores.append(ridge_model.score(X_val, y_valid))

# Find alpha with highest validation score
pd.DataFrame({
    "alpha": alphas,
    "training": train_scores,
    "validation": val_scores
}).sort_values("validation", ascending=False).head(1)
```

	alpha	training	validation
161	71.49429	0.863131	0.829146



PRO TIP: RIDGECV

The **RidgeCV()** function performs a cross validation loop that returns the best-performing value for alpha *(instead of writing a loop from scratch)*

- RidgeCV(alphas=list_of_alphas, cv=5_or_10)
- Use **.alpha_** to return the best-performing value for alpha

Cross validation with RidgeCV

```
from sklearn.linear_model import RidgeCV

X_m = std.transform(X)

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

ridge_model = RidgeCV(alphas=alphas, cv=5)

ridge_model.fit(X_m, y)
print(f"Alpha: {ridge_model.alpha_}")
print(f"Train R2: {ridge_model.score(X_m, y)}")
print(f"Test R2: {ridge_model.score(X_te, y_test)}")
```

```
Alpha: 41.0265810582719
Train R2: 0.8622763072049645
Test R2: 0.8607693938572964
```

RidgeCV() used an alpha of 41.02 and achieved a better test score than the alpha of 71.49 from the normal validation loop

Validation loop

```
ridge_model = Ridge(alpha=71.49).fit(X_m, y)

print(f"Train R2: {ridge_model.score(X_m, y)}")
print(f"Test R2: {ridge_model.score(X_te, y_test)}")
```

```
Train R2: 0.8595599407160773
Test R2: 0.859724346204797
```

RidgeCV() will usually yield a different optimal regularization strength than simple validation, and tends to yield **better test performance** as it looks at multiple cross validation folds

ASSIGNMENT: RIDGE REGRESSION

NEW MESSAGE

July 22, 2023

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

Subject: **Ridge Regression**

Hi there!

I love the model you built for our computer price data set.

Can you try fitting a ridge regression and compare the model's accuracy?

Thanks!

08_regularization_assignments.ipynb

Reply Forward

Key Objectives

1. Fit a ridge regression model in Python
2. Tune alpha through cross validation
3. Compare model performance with traditional regression

LASSO REGRESSION

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

Lasso regression adds a regularization term equal to the sum of the magnitude (absolute value) of the model's coefficients (β), also known as an L1 penalty

- The L1 penalty shrinks the coefficients towards 0 *(they can drop to 0, unlike ridge regression)*
- The higher alpha, the more the coefficients get shrunk
- The features that reduce SSE the most will shrink at a slower rate than less useful ones

$$J = SSE + \alpha \sum_{j=1}^{p} |\beta_j|$$

When fitting a lasso regression, the model is trying to minimize this cost function by both:

- *Minimizing training error with SSE*
- *Keeping the coefficient values small*

PRO TIP: If you have a lots of features in your model, fitting a moderately penalized lasso regression can help inform which features are strongest by dropping the rest to 0

FITTING A LASSO REGRESSION

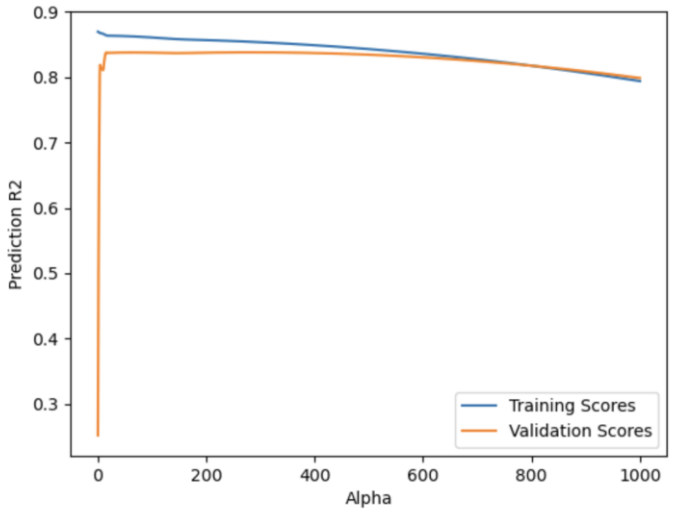
- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

You can fit a lasso regression with the **Lasso()** function in sklearn.linear_model

```
from sklearn.linear_model import Lasso

lasso_model = Lasso(alpha=286.61).fit(X_tr, y_train)

print(f"Train R2: {lasso_model.score(X_tr, y_train)}")
print(f"Validation R2: {lasso_model.score(X_val, y_valid)}")
```

```
Train R2: 0.854133796001005
Validation R2: 0.8381812194410436
```

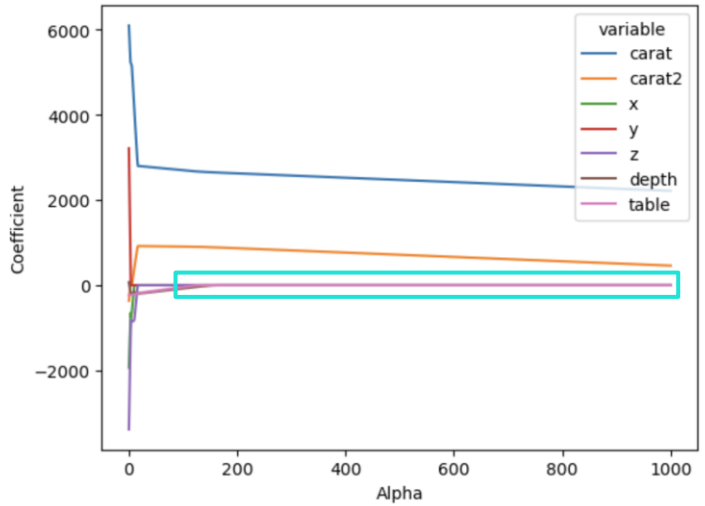
You can tune alpha with the same validation loop!




```
list(zip(X.columns, lasso_model.coef_))
```

```
[('carat', 2581.63588097972),
 ('carat2', 823.6784298610111),
 ('x', 0.0),
 ('y', 0.0),
 ('z', 0.0),
 ('depth', -0.0),
 ('table', -0.0)]
```

These coefficients all dropped to 0!



PRO TIP: LASSOCV

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

Like RidgeCV(), the **LassoCV()** function performs a cross validation loop that returns the best-performing value for alpha

- LassoCV(alphas=list_of_alphas, cv=5_or_10)
- Use **.alpha_** to return the best-performing value for alpha

```
from sklearn.linear_model import LassoCV

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

lasso_model = LassoCV(alphas=alphas, cv=5)

lasso_model.fit(X_m, y)
print(f"Alpha: {lasso_model.alpha_}")
print(f"Train R2: {lasso_model.score(X_m, y)}")
print(f"Test R2: {lasso_model.score(X_te, y_test)}")
```

```
Alpha: 8.907354638610439
Train R2: 0.8637391077142467
Test R2: 0.8606971727189058
```

ASSIGNMENT: LASSO REGRESSION

NEW MESSAGE

July 23, 2023

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

Subject: **RE: Ridge Regression**

Hi there!

I was hoping ridge regression would increase performance more than it did, but given that we didn't have any highly correlated features, it makes sense.

I meant to ask you to check the performance of the lasso model as well, can you try a lasso model and compare it with the others?

Thanks!

08_regularization_assignments.ipynb

Reply Forward

Key Objectives

1. Fit a lasso regression model in Python
2. Tune alpha through cross validation
3. Compare model performance with traditional regression

ELASTIC NET REGRESSION

- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

Elastic net regression combines the ridge and lasso penalties into a single model and introduces a new hyperparameter (**λ**) that controls the balance between them

- When lambda is 1, the model is equivalent to lasso; when it is 0, it is equivalent to ridge

Ridge penalty *Lasso penalty*

$$J = SSE + \alpha((1-\lambda)\sum_{j=1}^{p}\beta_j^2 + (\lambda)\sum_{j=1}^{p}|\beta_j|)$$

*Total **regularization strength***

*Controls the **balance** between ridge & lasso*

PRO TIP: Elastic net regression combines the effects of both lasso and ridge regression and can be a lifesaver for challenging modeling problems

FITTING AN ELASTIC NET REGRESSION

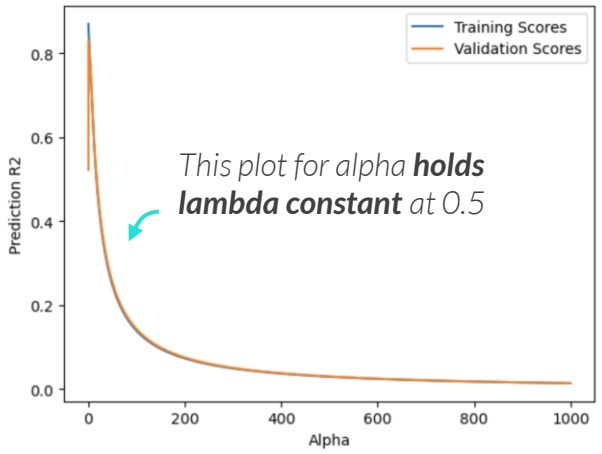
- Regularization
- Ridge Regression
- Lasso Regression
- Elastic Net Regression

You can fit an elastic net with the **ElasticNet()** function in sklearn.linear_model

- ElasticNet(alpha=1, l1_ratio=.5)
- The "l1_ratio" controls the balance of L1 to L2 penalty *(the default is 0.5, or equal weight)*

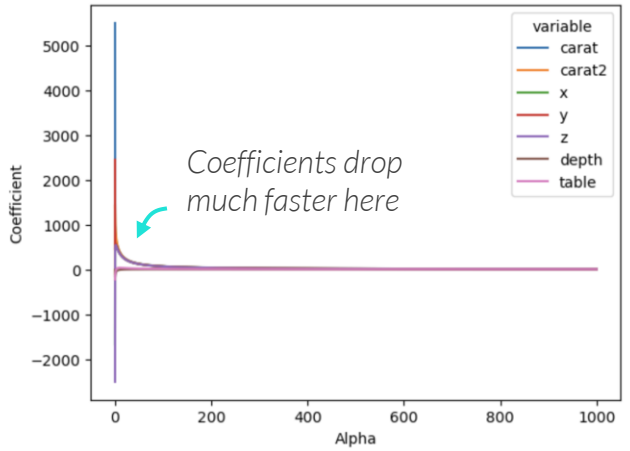
```
from sklearn.linear_model import ElasticNet

enet_model = ElasticNet().fit(X_tr, y_train)

print(f"Training R2: {enet_model.score(X_tr, y_train)}")
print(f"Validation R2: {enet_model.score(X_val, y_valid)}")
```

```
Training R2: 0.8386421298117831
Validation R2: 0.8175744926112136
```




```
list(zip(X.columns, enet_model.coef_))
```

```
[('carat', 779.3937390279258),
 ('carat2', 927.2606705791363),
 ('x', 552.2821166392038),
 ('y', 567.9872690861856),
 ('z', 536.4803214170591),
 ('depth', -95.55161623634689),
 ('table', -60.435649026085905)]
```



TUNING LAMBDA

Regularization

Ridge Regression

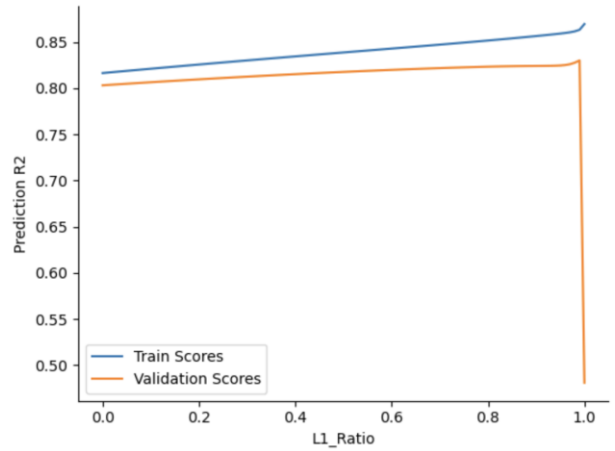
Lasso Regression

Elastic Net Regression

You need to **tune lambda** in order to find the best ridge / lasso balance

- To do so, you can try fixing alpha at 1 and evaluating performance across "l1_ratios"

```
train_scores = []
val_scores = []

l1_ratios = [x * .01 for x in range(101)]


for l1_ratio in l1_ratios:
    enet_model = ElasticNet(alpha=1, l1_ratio=l1_ratio).fit(X_tr, y_train)
    train_scores.append(enet_model.score(X_tr, y_train))
    val_scores.append(enet_model.score(X_val, y_valid))
```



Tuning lambda while keeping alpha constant, or vice versa, misses out on many possible combinations of these hyperparameters that might perform better – we need to **try multiple combinations of both**!

PRO TIP: ELASTICNETCV

Regularization

Ridge Regression

Lasso Regression

Elastic Net Regression

The **ElasticNetCV()** function performs a cross validation loop that returns the best-performing values for *both* alpha and lambda

- ENetCV(alphas=list_of_alphas, l1_ratio=list_of_lambdas, cv=5_or_10)
- Use **.alpha_** and **.l1_ratio_** to return the best-performing values for alpha and lambda

```
from sklearn.linear_model import ElasticNetCV

alphas = 10 ** np.linspace(-3, 3, 200)
l1_ratios = [x * .1 for x in range(10)]

enet_model = ElasticNetCV(alphas=alphas, l1_ratio=l1_ratios, cv=5)

enet_model = enet_model.fit(X_m, y)

print(f"Tuned Alpha: {enet_model.alpha_}")
print(f"Tuned Lambda: {enet_model.l1_ratio_}")
print(f"Training Score: {enet_model.score(X_m, y)}")
print(f"Test Score: {enet_model.score(X_te, y_test)}")
```

```
Tuned Alpha: 0.05607169938205458
Tuned Lambda: 0.9
Training Score: 0.8614868346319197
Test Score: 0.8605300373289071
```

Low regularization strength, and 90% skewed towards a Lasso penalty

This model slightly underperformed the Lasso model, but it is still quite good

ASSIGNMENT: ELASTIC NET REGRESSION

NEW MESSAGE

July 25, 2023

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

Subject: **Re: Re: Ridge Regression**

Hi there!

Ok, last request for a bit, I promise!

Can you try an elastic net as well?

Just want to make sure we're checking all of our possible options here.

Thanks!

08_regularization_assignments.ipynb

Reply Forward

Key Objectives

1. Fit an elastic net regression model in Python
2. Tune alpha and lambda through cross validation
3. Compare model performance with other linear regression models

RECAP: REGULARIZED REGRESSION MODELS

Model	Cost Function	Penalty Type	Hyper-parameters	Details
Linear Regression (OLS)	SSE	None	None	Fits a line of best fit by minimizing SSE
Ridge Regression	$SSE + \alpha \sum_{j=1}^{p} \beta_j^2$	L2 penalty	α	Helps with overfitting by shrinking coefficients towards zero, but never reaching it. Great for modelling highly correlated features
Lasso Regression	$SSE + \alpha \sum_{j=1}^{p} \lvert\beta_j\rvert$	L1 penalty	α	Helps with overfitting by dropping some coefficients to 0, making it a good variable selection technique
Elastic Net Regression	$SSE + \alpha((1-\lambda) \sum_{j=1}^{p} \beta_j^2 + \lambda \sum_{j=1}^{p} \lvert\beta_j\rvert)$	L2 and L1 penalty	α, λ	Helps with overfitting by balancing ridge and lasso regression

KEY TAKEAWAYS

Regularized regression **adds bias** to a model to help **reduce variance**

- *To achieve this, it strays away from the line of best fit by introducing a regularization term to the cost equation*
- *The goal is now to minimize the sum of squared error AND keep the coefficient estimates small*

There are **3 types** of regularized regression models you can try

- *Any regularization technique combats overfitting, with Ridge regression reducing the coefficient values to be closer to 0, Lasso regression dropping some coefficients down to 0, and Elastic Net balancing the two*

All features need to be **standardized** before being input into these models

- *This allows the resulting coefficients to be fairly compared with one another*
- *Use the training data set to calculate the mean and standard deviation values, then apply to the test data set*

Tuning the **hyperparameters** is key in achieving the best performance

- *The alpha and lambda hyperparameters modify the regularization penalty amount in the cost equation*

PROJECT 2: REGULARIZATION

PROJECT DATA: SAN FRANCISCO RENT PRICES

```
apartments_df.head()
```

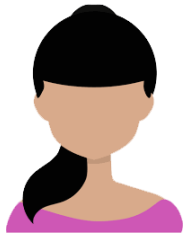
	price	sqft	beds	bath	laundry	pets	housing_type	parking	hood_district
0	6800	1600.0	2.0	2.0	(a) in-unit	(d) no pets	(c) multi	(b) protected	7.0
1	3500	550.0	1.0	1.0	(a) in-unit	(a) both	(c) multi	(b) protected	7.0
2	5100	1300.0	2.0	1.0	(a) in-unit	(a) both	(c) multi	(d) no parking	7.0
3	9000	3500.0	3.0	2.5	(a) in-unit	(d) no pets	(c) multi	(b) protected	7.0
4	3100	561.0	1.0	1.0	(c) no laundry	(a) both	(c) multi	(d) no parking	7.0

```
apartments_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 989 entries, 0 to 988
Data columns (total 9 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   price          989 non-null    int64
 1   sqft           989 non-null    float64
 2   beds           989 non-null    float64
 3   bath           989 non-null    float64
 4   laundry        989 non-null    object
 5   pets           989 non-null    object
 6   housing_type   989 non-null    object
 7   parking        989 non-null    object
 8   hood_district  989 non-null    float64
dtypes: float64(4), int64(1), object(4)
memory usage: 69.7+ KB
```

MAVEN CONSULTING GROUP

PROJECT BRIEF: REGULARIZATION



NEW MESSAGE

July 27, 2023

From: **Cathy Coefficient** *(Data Science Lead)*

Subject: **RE: Apartment Rental Prices**

Hi again,

I just reviewed your modelling work and, overall, I'm quite pleased with the results.

In order to make sure we're doing everything we can to exceed our client expectations, I'd like to see if regularized regression can improve the model.

Please build on your previous work to fit regularized regression models and compare them with your original model, then select the final model based on accuracy.

09_regularized_regression_project.ipynb

Reply Forward

Key Objectives

1. Fit Lasso, Ridge, and Elastic Net Regression Models
2. Tune the models to the optimal regularization strength based on validation results
3. Select the model which has the highest accuracy on out of sample data

MAVEN CONSULTING GROUP

TIME SERIES ANALYSIS

TIME SERIES ANALYSIS

In this section we'll cover **time series analysis** and forecasting, specialized techniques applied to time series data to extract patterns & trends and predict future values

TOPICS WE'LL COVER:

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

GOALS FOR THIS SECTION:

- Learn how to transform data into a format that is ready for time series analysis and forecasting
- Become familiar with the various ways to smooth time series data to visualize patterns
- Understand how decomposition works and how it can be useful for looking at trends and seasonality
- Apply time series forecasting techniques, including OLS and Facebook Prophet

TIME SERIES DATA

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

Time series data requires each row to represent a unique moment in time

- This can be in any unit of time (seconds, hours, days, months, years, etc.)

Raw Data

Row ID	Order Date	Customer ID	Product Name	Sales
541	2015-01-02	BD-11500	Enermax Aurora Lite Keyboard	468.900
5714	2015-01-03	HR-14770	Sannysis Cute Owl Design Soft Skin Case Cover ...	5.940
158	2015-01-03	DB-13060	Global Deluxe High-Back Manager's Chair	457.568
6549	2015-01-03	VF-21715	Black Print Carbonless 8 1/2" x 8 1/4" Rapid M...	17.472
7950	2015-01-03	SC-20380	DAX Black Cherry Wood-Tone Poster Frame	63.552
...	...	...	...	...
909	2018-12-30	PO-18865	Wilson Jones Legal Size Ring Binders	52.776
646	2018-12-30	CC-12430	Eureka The Boss Plus 12-Amp Hard Box Upright V...	209.300
908	2018-12-30	PO-18865	Gear Head AU3700S Headset	90.930
907	2018-12-30	PO-18865	Bush Westfield Collection Bookcases, Fully Ass...	323.136
1298	2018-12-30	EB-13975	GBC Binding covers	20.720

*This is **NOT time series data**, as each row represents a transaction, not a point in time*

Time Series Data

Daily

Order Date	Sales
2015-01-02	468.900
2015-01-03	2203.151
2015-01-04	119.888
2015-01-06	5188.520
2015-01-07	601.024

Monthly

Order Date	Sales
2015-01-31	28828.254
2015-02-28	12588.484
2015-03-31	54027.692
2015-04-30	24710.016
2015-05-31	29520.490

Yearly

Order Date	Sales
2015-12-31	479856.2081
2016-12-31	459436.0054
2017-12-31	600192.5500
2018-12-31	722052.0192

*Aggregating the data by date converts it into **time series data***

Deciding **which unit of time to analyze** is an important first step. Does your company need a daily forecast to help plan staffing? Or is does it need a monthly forecast to help finance make budgeting plans?

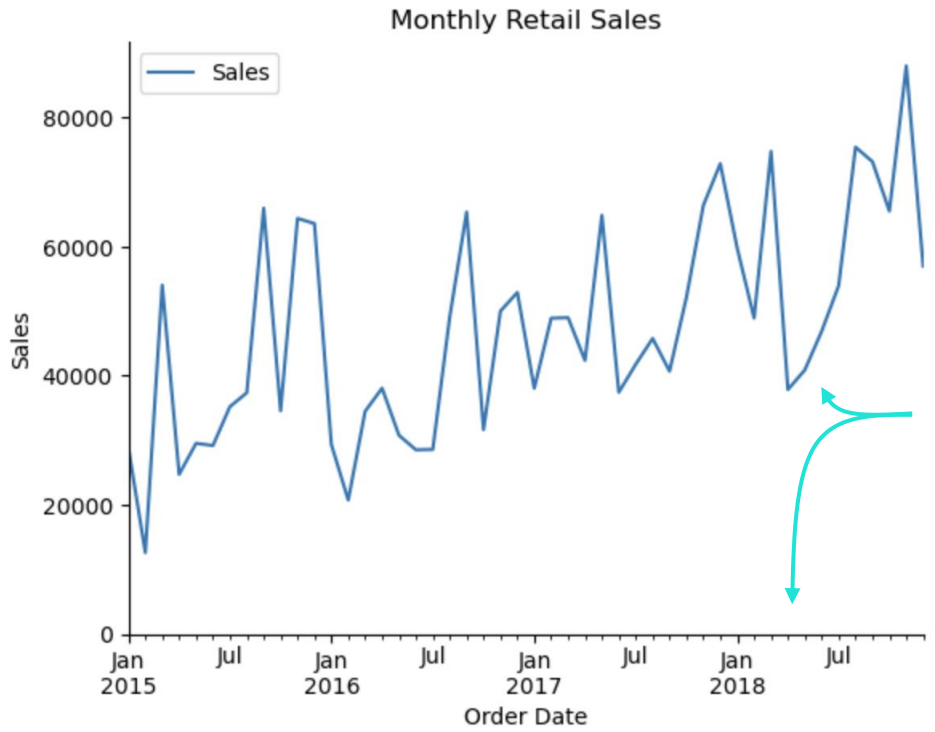
TIME SERIES DATA

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

Time series data requires each row to represent a unique moment in time

- This can be in any unit if time (seconds, hours, days, months, years, etc.)

Order Date	Sales
2015-01-31	28828.254
2015-02-28	12588.484
2015-03-31	54027.692
2015-04-30	24710.016
2015-05-31	29520.490



*Time series data is often visualized using a **line chart** with time as the x-axis*

TYPES OF TIME SERIES ANALYSIS

Time Series Data

Smoothing

Decomposition

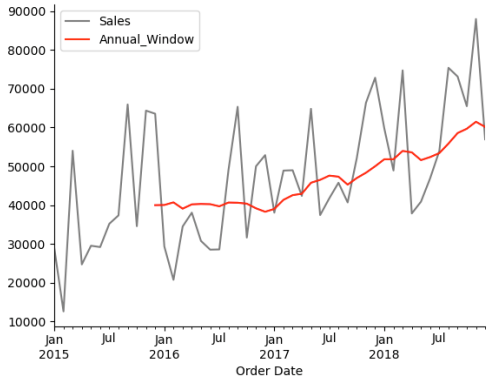
Forecasting

There are many **types of time series analysis**, including:

Smoothing

Reduces volatility *to reveal underlying trends & patterns*

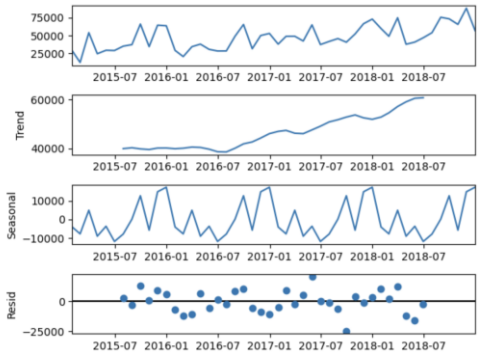
- Moving average
- Exponential smoothing



Decomposition

Breaks down data into ***seasonality, trend, and noise*** *components*

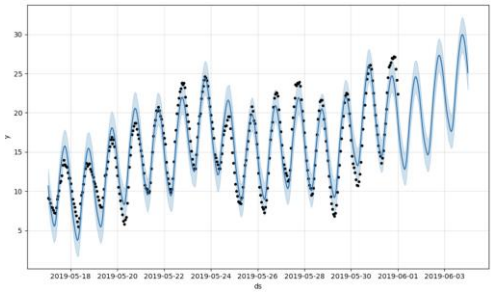
- Additive model
- Multiplicative model



Forecasting

Predicts future values *in time using historical time series data*

- Linear Regression
- Facebook Prophet*



MOVING AVERAGE

Time Series Data

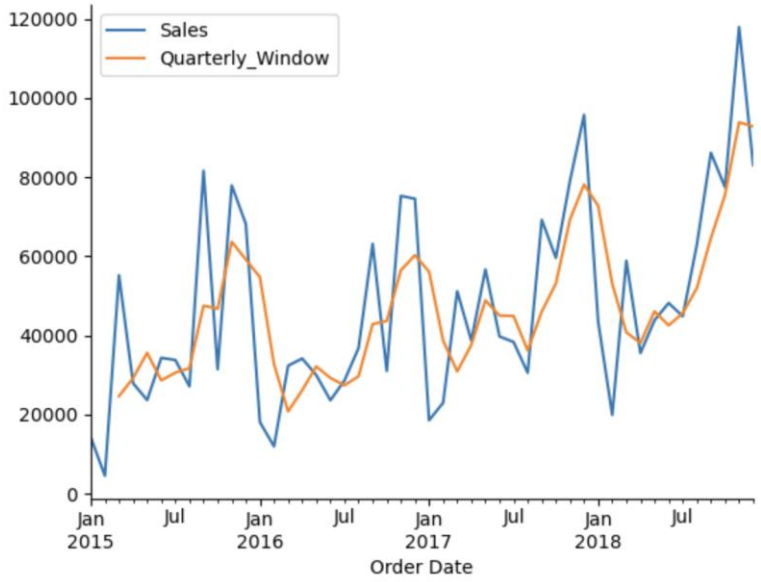
Smoothing

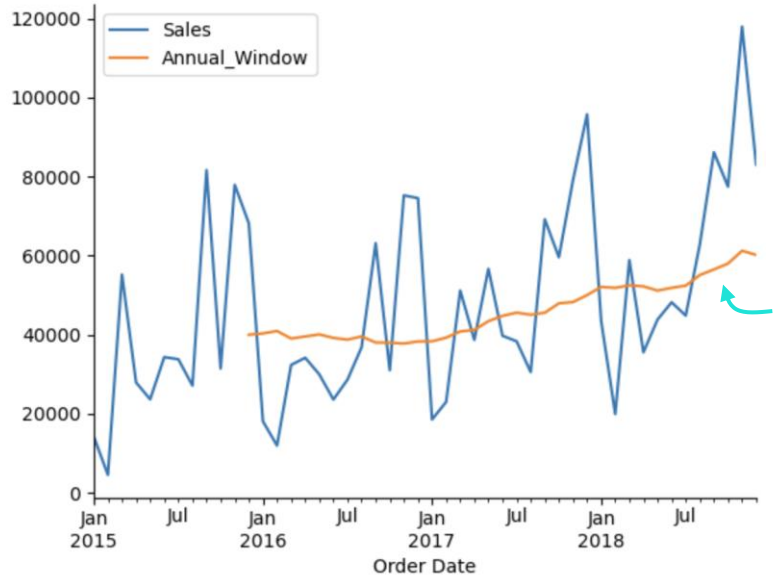
Decomposition

Forecasting

Time series smoothing is the process of reducing volatility in time series data to help identify trends and patterns that are otherwise challenging to see

The simplest way to smooth time series data is by calculating a **moving average**





The larger the window, the smoother the data

PRO TIP: Align your windows with intuitive seasons: with monthly data, look at quarters or years, with daily data look at weekly or monthly, etc.

MOVING AVERAGE

Time Series Data

Smoothing

Decomposition

Forecasting

The **rolling()** method lets you calculate moving averages for a specified window

- df["col"].rolling(window).mean()

```
monthly_sales.assign(
    quarterly=monthly_sales["Sales"].rolling(window=3).mean(),
    half_year=monthly_sales["Sales"].rolling(window=6).mean(),
    annual=monthly_sales["Sales"].rolling(window=12).mean()
).head(12)
```

This creates new columns with a 3-month, 6-month, and 12-month moving average

Order Date	Sales	quarterly	half_year	annual
2015-01-31	14205.7070	NaN	NaN	NaN
2015-02-28	4519.8920	NaN	NaN	NaN
2015-03-31	55205.7970	24643.798667	NaN	NaN
2015-04-30	27906.8550	29210.848000	NaN	NaN
2015-05-31	23644.3030	35585.651667	NaN	NaN
2015-06-30	34322.9356	28624.697867	26634.248267	NaN
2015-07-31	33781.5430	30582.927200	29896.887600	NaN
2015-08-31	27117.5365	31740.671700	33663.161683	NaN
2015-09-30	81623.5268	47507.535433	38066.116650	NaN
2015-10-31	31453.3930	46731.485433	38657.206317	NaN
2015-11-30	77907.6607	63661.526833	47701.099267	NaN
2015-12-31	68167.0585	59176.037400	53341.786417	39988.017342

How does it work?

- Each "quarterly" value represents the average sales from the current month and 2 previous

MOVING AVERAGE

Time Series Data

Smoothing

Decomposition

Forecasting

The **rolling()** function lets you calculate moving averages for a specified window

- df["col"].rolling(window).mean()

```
monthly_sales.assign(
    quarterly=monthly_sales["Sales"].rolling(window=3).mean(),
    half_year=monthly_sales["Sales"].rolling(window=6).mean(),
    annual=monthly_sales["Sales"].rolling(window=12).mean()
).head(12)
```

This creates new columns with a 3-month, 6-month, and 12-month moving average

Order Date	Sales	quarterly	half_year	annual
2015-01-31	14205.7070	NaN	NaN	NaN
2015-02-28	4519.8920	NaN	NaN	NaN
2015-03-31	55205.7970	24643.798667	NaN	NaN
2015-04-30	27906.8550	29210.848000	NaN	NaN
2015-05-31	23644.3030	35585.651667	NaN	NaN
2015-06-30	34322.9356	28624.697867	26634.248267	NaN
2015-07-31	33781.5430	30582.927200	29896.887600	NaN
2015-08-31	27117.5365	31740.671700	33663.161683	NaN
2015-09-30	81623.5268	47507.535433	38066.116650	NaN
2015-10-31	31453.3930	46731.485433	38657.206317	NaN
2015-11-30	77907.6607	63661.526833	47701.099267	NaN
2015-12-31	68167.0585	59176.037400	53341.786417	39988.017342

How does it work?

- Each "quarterly" value represents the average sales from the current month and 2 previous
- Each "half_year" value represents the average sales from the current month and 5 previous

MOVING AVERAGE

Time Series Data

Smoothing

Decomposition

Forecasting

The **rolling()** function lets you calculate moving averages for a specified window

- df["col"].rolling(window).mean()

```
monthly_sales.assign(
    quarterly=monthly_sales["Sales"].rolling(window=3).mean(),
    half_year=monthly_sales["Sales"].rolling(window=6).mean(),
    annual=monthly_sales["Sales"].rolling(window=12).mean()
).head(12)
```

This creates new columns with a 3-month, 6-month, and 12-month moving average

Order Date	Sales	quarterly	half_year	annual
2015-01-31	14205.7070	NaN	NaN	NaN
2015-02-28	4519.8920	NaN	NaN	NaN
2015-03-31	55205.7970	24643.798667	NaN	NaN
2015-04-30	27906.8550	29210.848000	NaN	NaN
2015-05-31	23644.3030	35585.651667	NaN	NaN
2015-06-30	34322.9356	28624.697867	26634.248267	NaN
2015-07-31	33781.5430	30582.927200	29896.887600	NaN
2015-08-31	27117.5365	31740.671700	33663.161683	NaN
2015-09-30	81623.5268	47507.535433	38066.116650	NaN
2015-10-31	31453.3930	46731.485433	38657.206317	NaN
2015-11-30	77907.6607	63661.526833	47701.099267	NaN
2015-12-31	68167.0585	59176.037400	53341.786417	39988.017342

How does it work?

- Each "quarterly" value represents the average sales from the current month and 2 previous
- Each "half_year" value represents the average sales from the current month and 5 previous
- Each "annual" value represents the average sales from the current month and 11 previous

EXPONENTIAL SMOOTHING

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

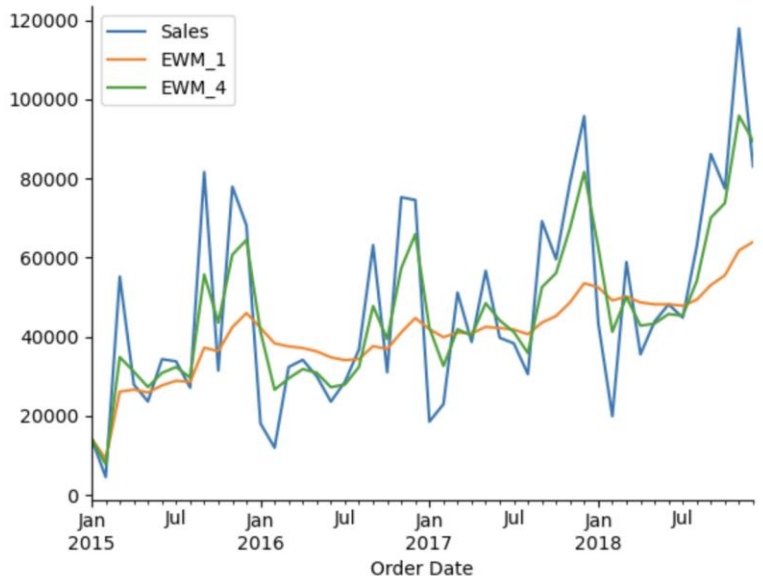
The **exponential smoothing** technique is similar to moving average, but it gives more weight to recent data points within a window

- The weight is controlled by alpha, which is a value between 0 and 1
- The higher alpha, the more weight is given to recent points *(which also increases volatility)*
- df["col].**ewm**(alpha).mean()

```
monthly_sales.assign(
    EWM_1=monthly_sales["Sales"].ewm(alpha=.1).mean(),
    EWM_4=monthly_sales["Sales"].ewm(alpha=.4).mean()
).head()
```

Order Date	Sales	EWM_1	EWM_4
2015-01-31	14205.707	14205.707000	14205.707000
2015-02-28	4519.892	9107.909632	8152.072625
2015-03-31	55205.797	26118.200173	32159.074857
2015-04-30	27906.855	26638.309166	30204.929702
2015-05-31	23644.303	25907.189983	27359.411528

No NaN values!



ASSIGNMENT: SMOOTHING

NEW MESSAGE

August 1, 2023

From: **Tammy Tiempo** *(Financial Analyst)*

Subject: **Smoothing**

Hi there!

We're working with an electric utility in Morocco on a forecasting model for electric consumption. The data is quite noisy due to lots of seasonality. Can you calculate a weekly and monthly rolling average so we can see broader patterns?

Thanks!

10_time_series_assignments.ipynb

Reply Forward

Key Objectives

1. Explore and manipulate time series data
2. Modify smoothing parameters to reveal different patterns

DECOMPOSITION

Time Series Data

Smoothing

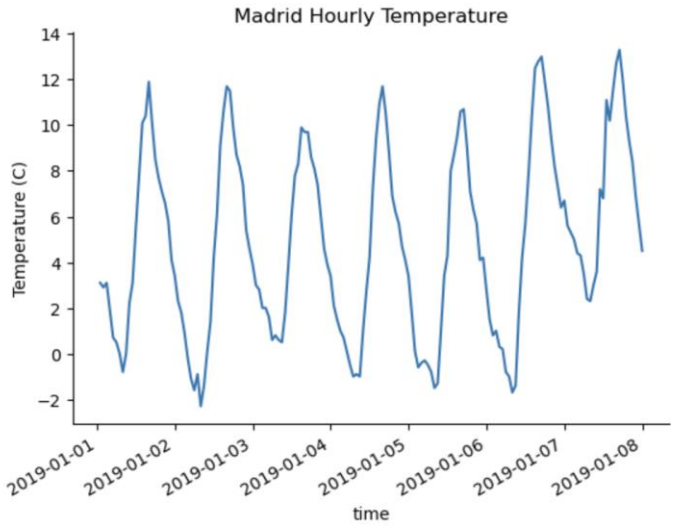
Decomposition

Forecasting

Time series data can be **decomposed** into trend, seasonality, and random noise

- **Trend**: Are values trending up, down, or staying flat over time?
- **Seasonality**: Do values display a cyclical pattern? *(like more customers buying on weekends)*
- **Random noise**: What volatility exists outside the trend and seasonal patterns?

This data has a positive trend, unclear seasonality, and lots of random noise



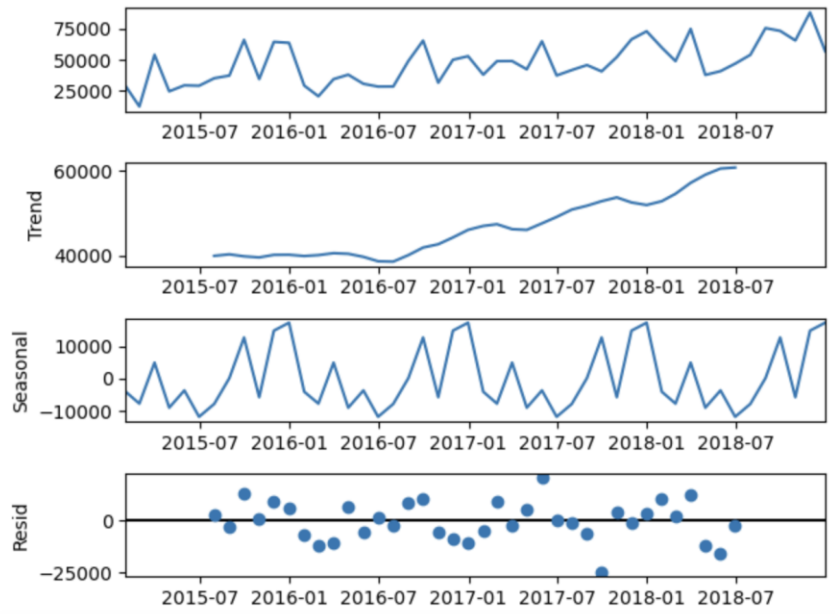
This data has a flat trend, clear hourly seasonality, and relatively little noise

DECOMPOSITION

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

You can use statsmodels' **seasonal_decompose()** to decompose time series data

```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(monthly_sales)

result.plot();
```



The positive trend started around July 2016

There appears to be a yearly seasonal pattern, but it doesn't seem to match the data that well

After taking away the trend and seasonality, the residuals look somewhat random over time

TYPES OF DECOMPOSITION

Time Series Data

Smoothing

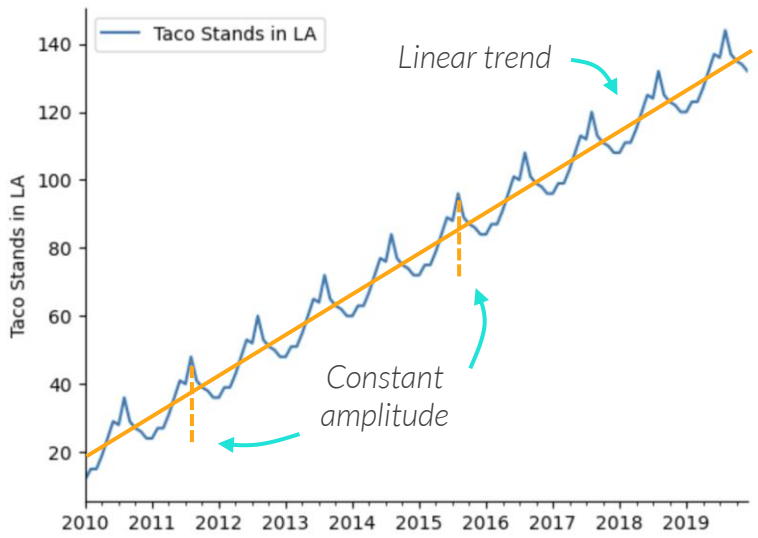
Decomposition

Forecasting

There are two types of decomposition: **additive** and **multiplicative**

- **Additive** decomposition assumes the trend and seasonality remain constant
- **Multiplicative** decomposition assumes the trend and seasonality increase over time

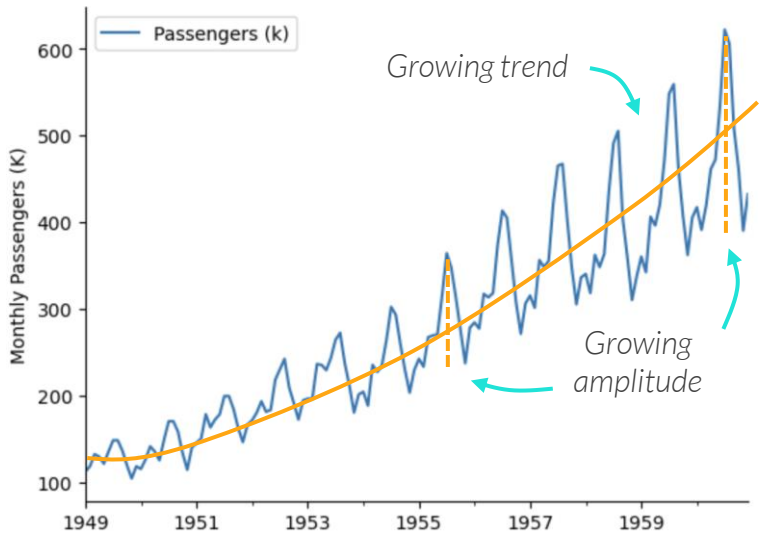
Additive



$$y_t = T_t + S_t + R_t$$

Time + seasonality + random

Multiplicative



$$y_t = T_t \times S_t \times R_t$$

*Time * seasonality * random*

TYPES OF DECOMPOSITION

Time Series Data

Smoothing

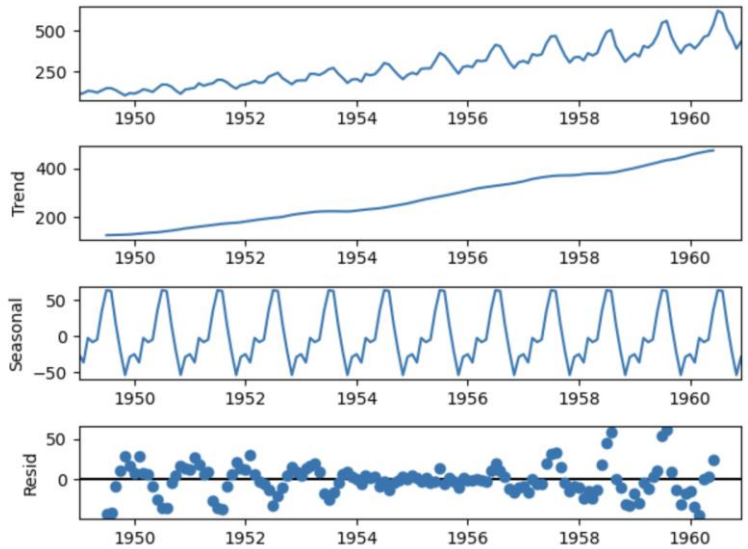
Decomposition

Forecasting

There are two types of decomposition: **additive** and **multiplicative**

- **Additive** decomposition assumes the trend and seasonality remain constant
- **Multiplicative** decomposition assumes the trend and seasonality increase over time

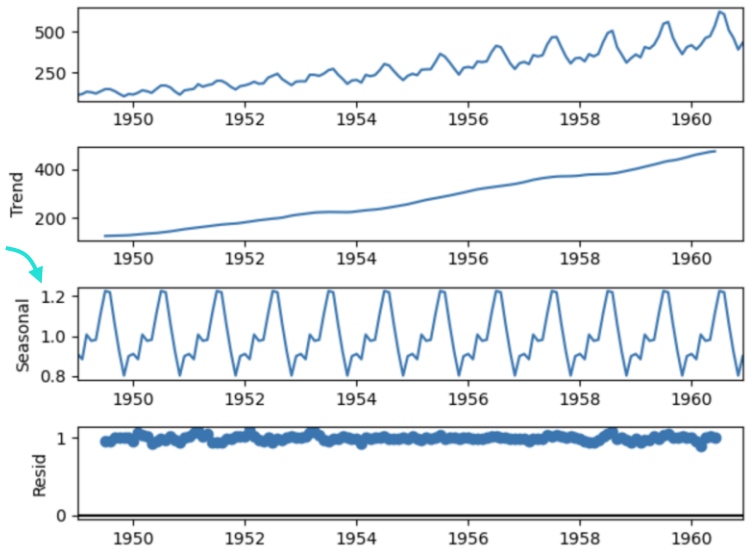
```
result = seasonal_decompose(air_traffic, model="additive")

result.plot();
```



The residuals look good in the middle but get worse towards the end, which indicates we need a multiplicative model

Multiplicators

```
result = seasonal_decompose(air_traffic, model="multiplicative")

result.plot();
```



We can no longer see a pattern in the residuals

PRO TIP: AUTOCORRELATION CHART

Time Series Data

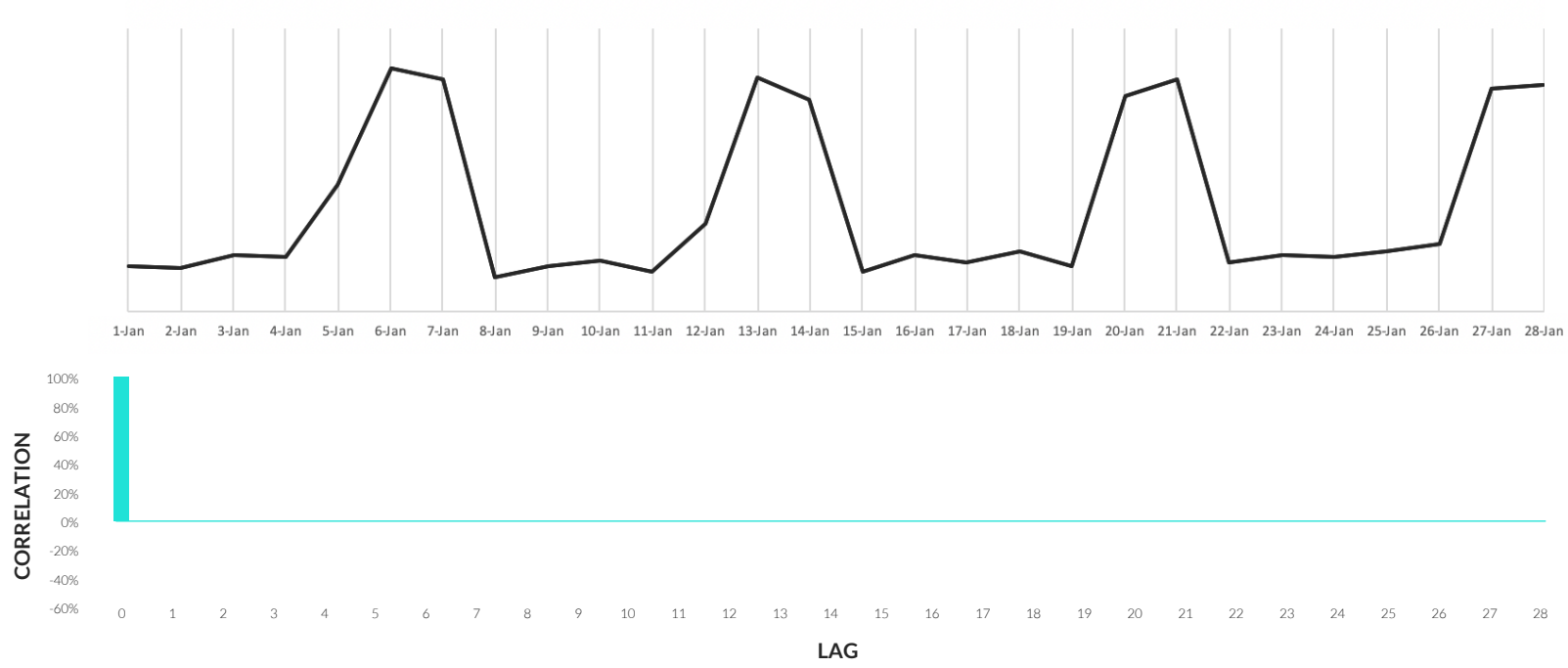
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

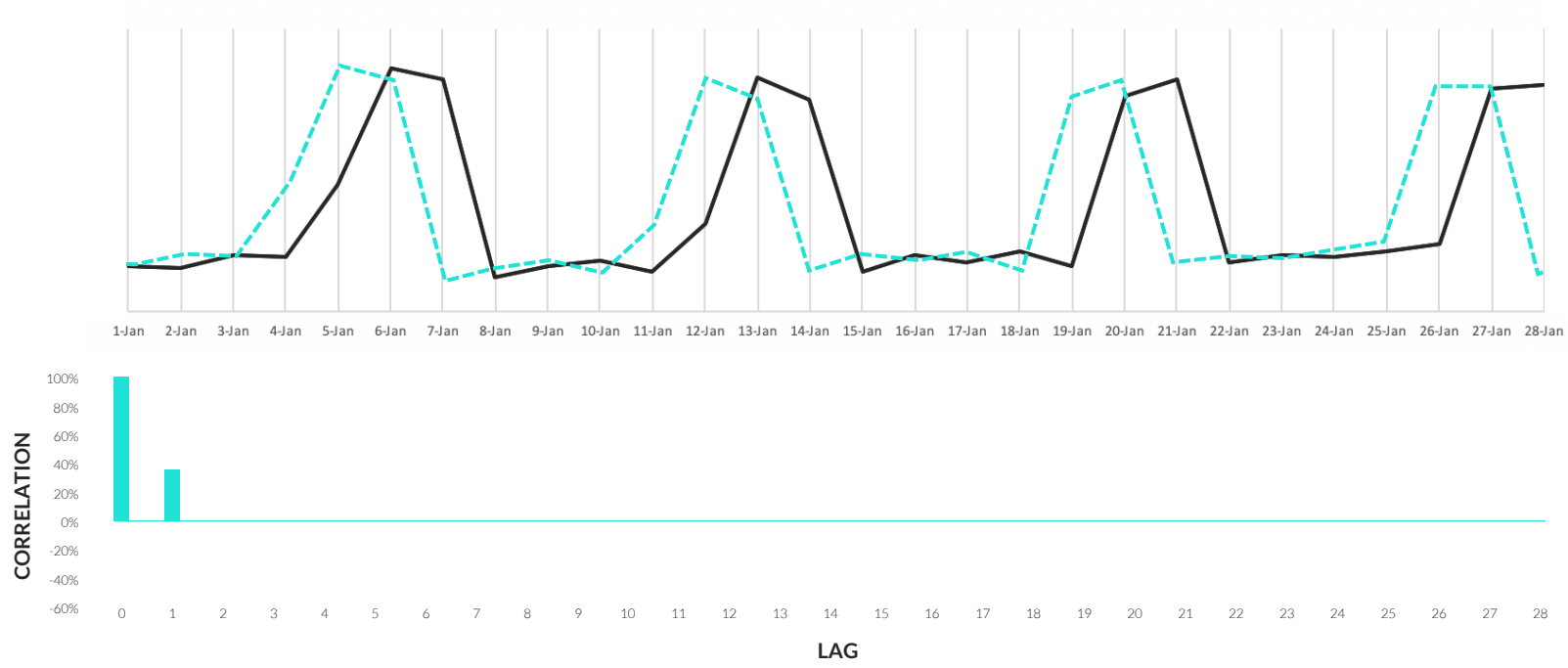
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

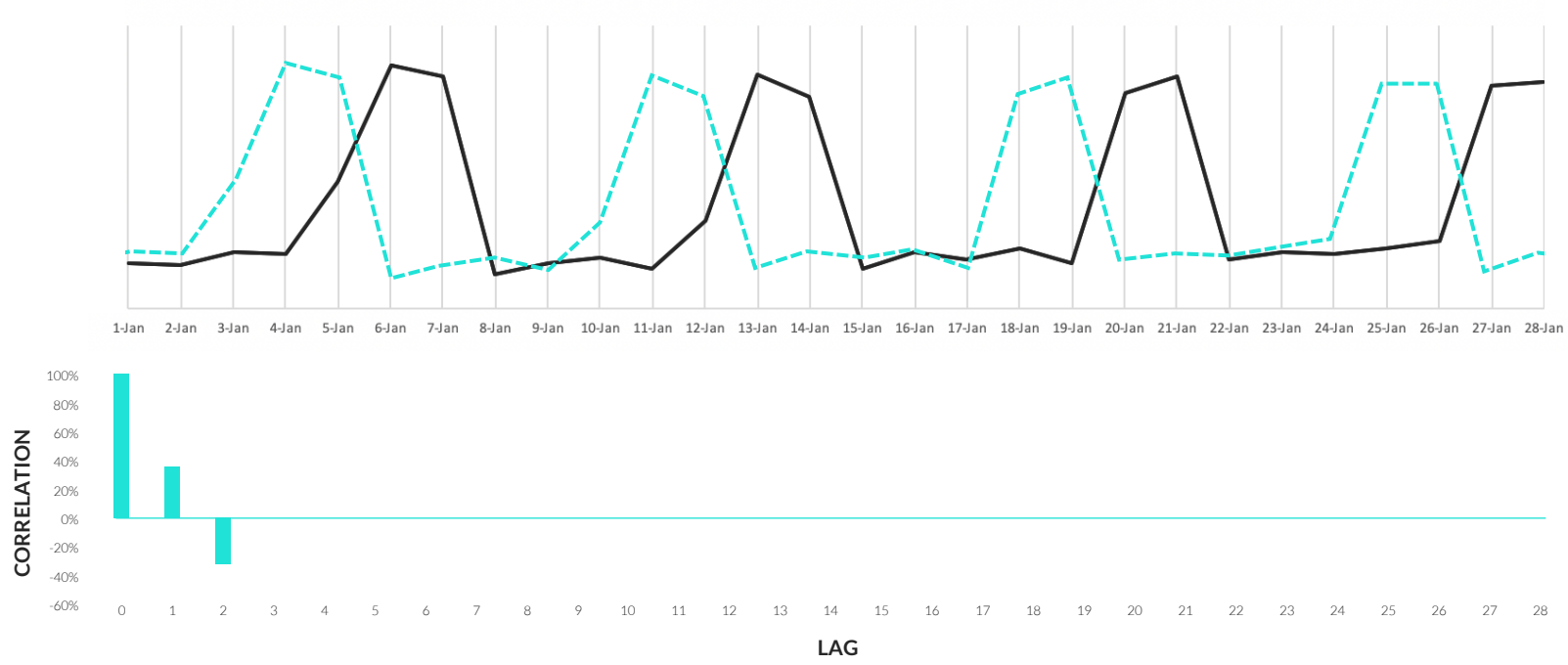
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

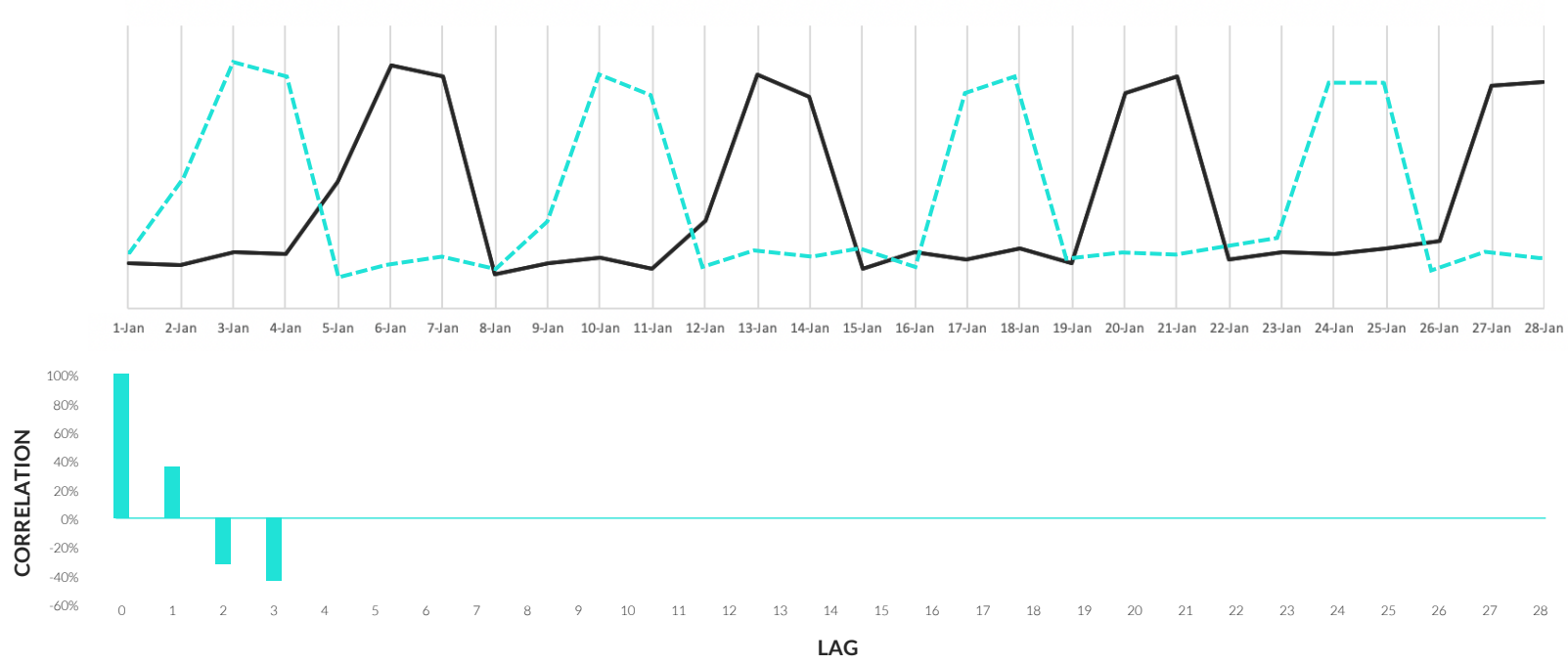
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

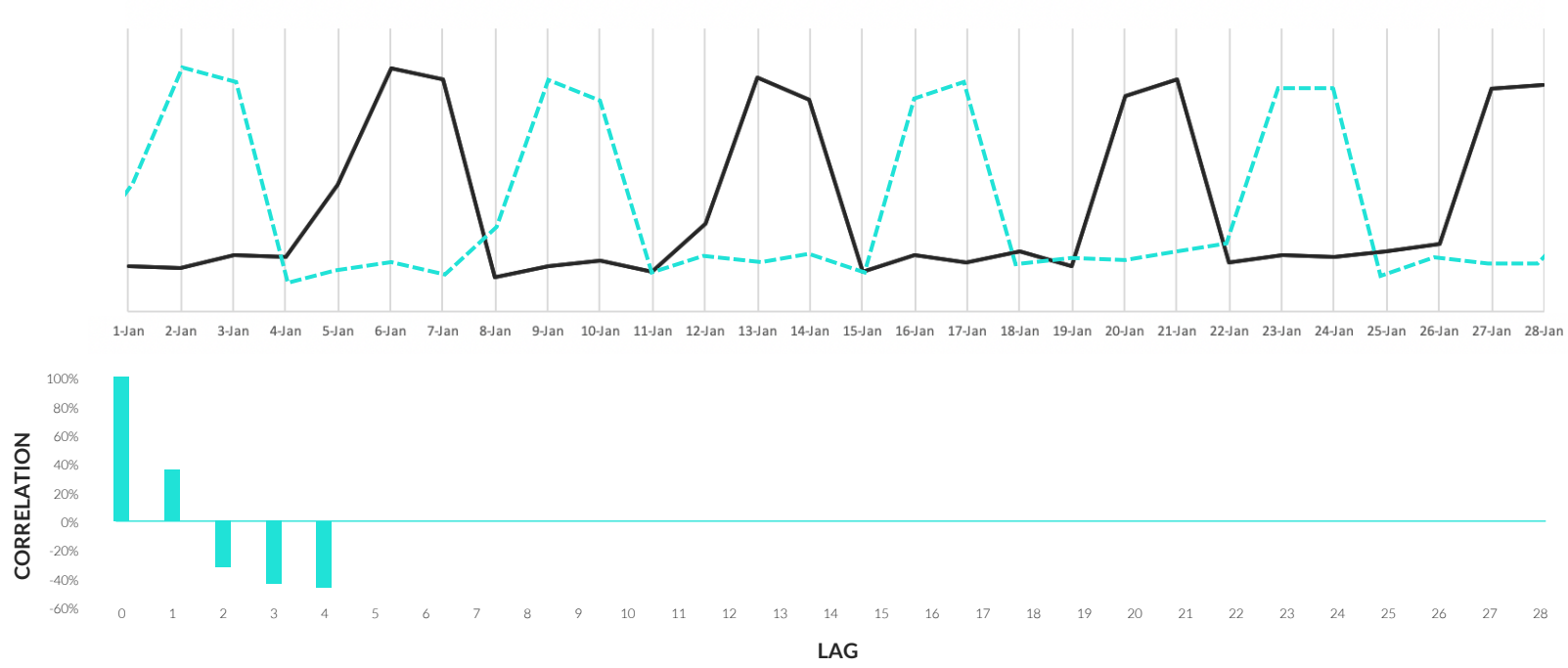
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

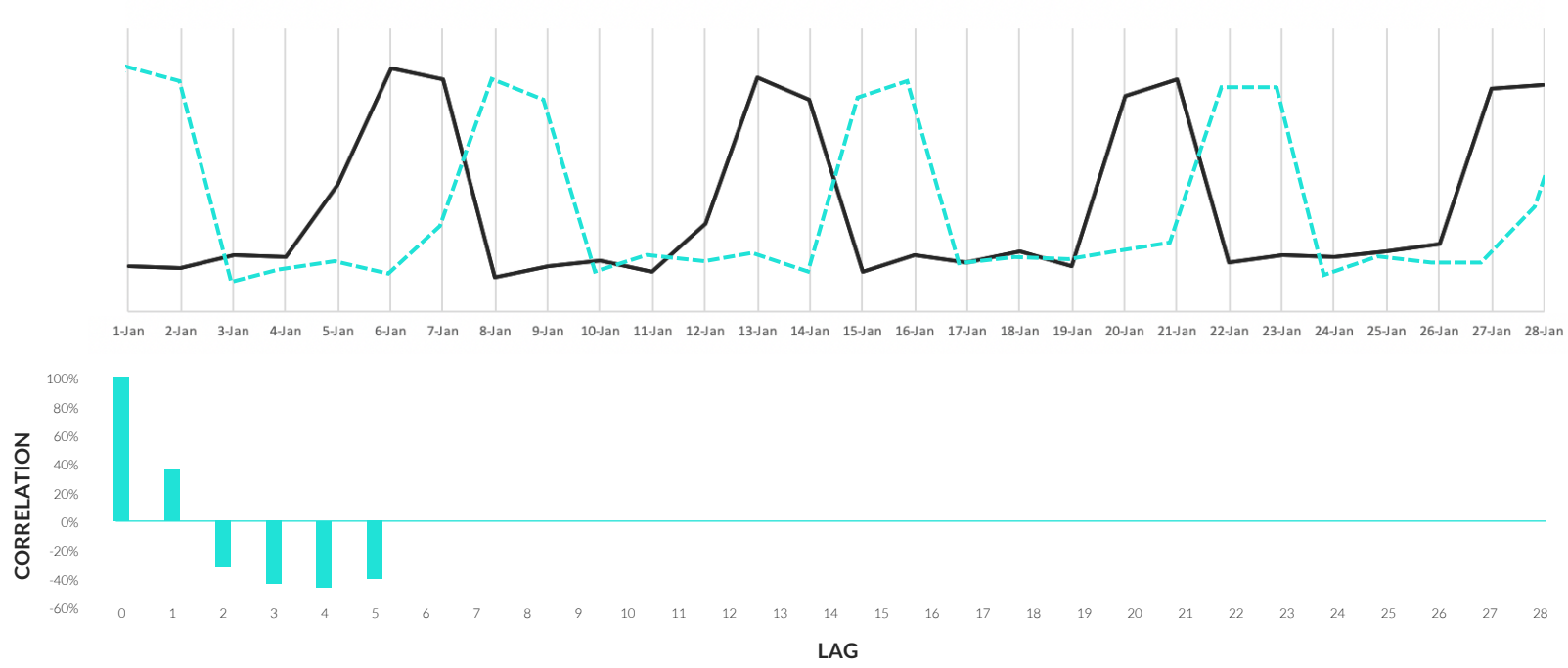
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

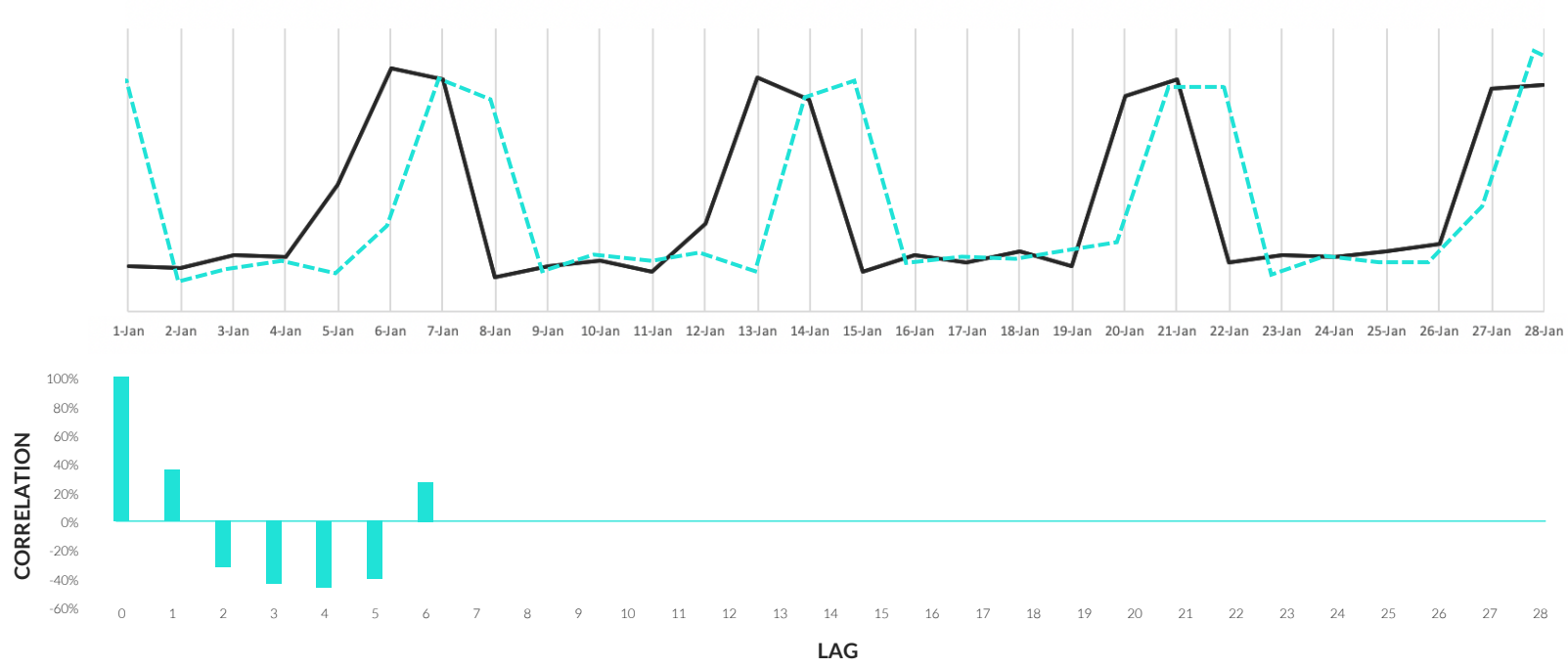
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

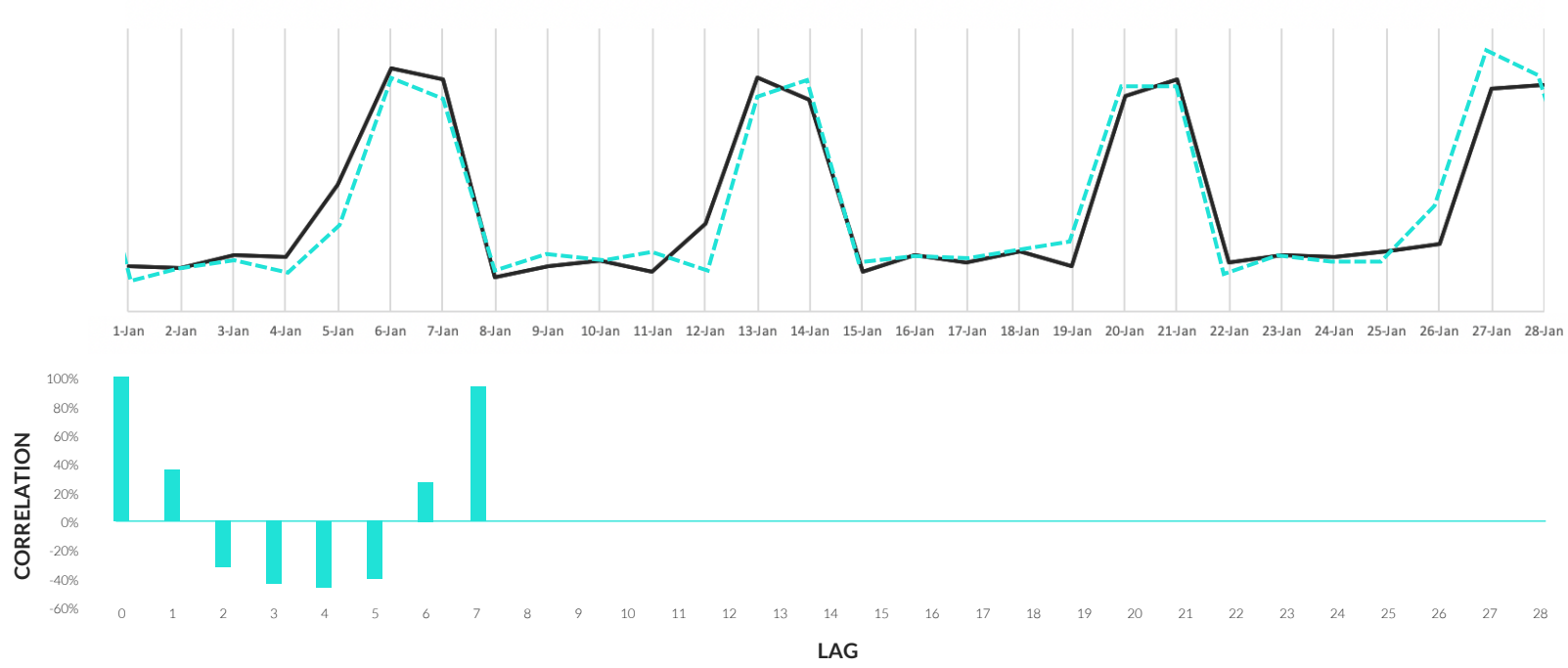
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

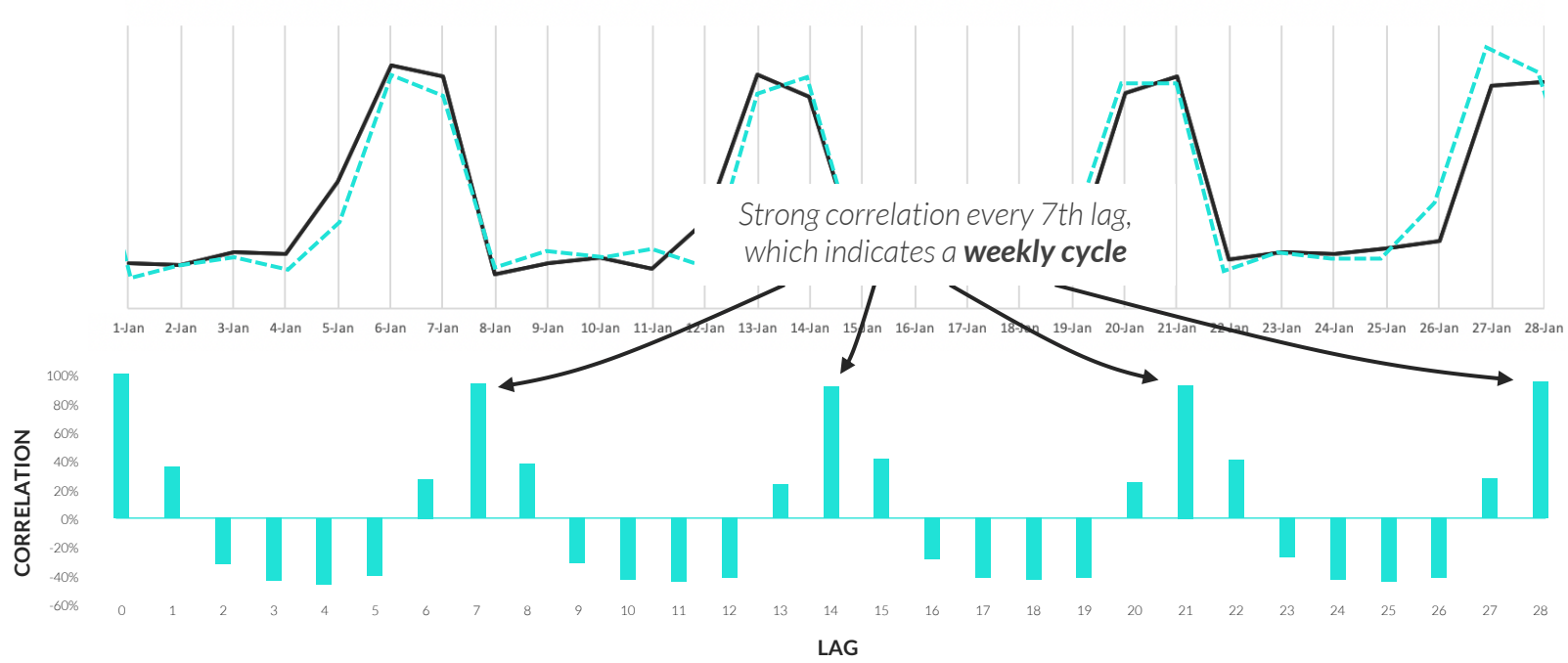
Smoothing

Decomposition

Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend



PRO TIP: AUTOCORRELATION CHART

Time Series Data

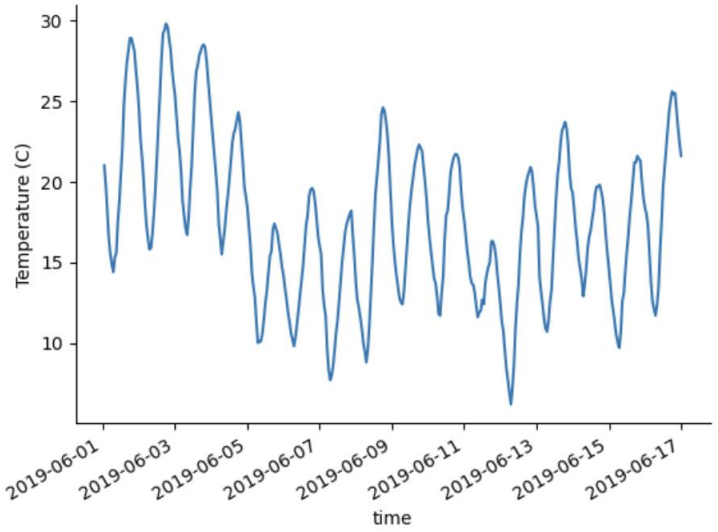
Smoothing

Decomposition

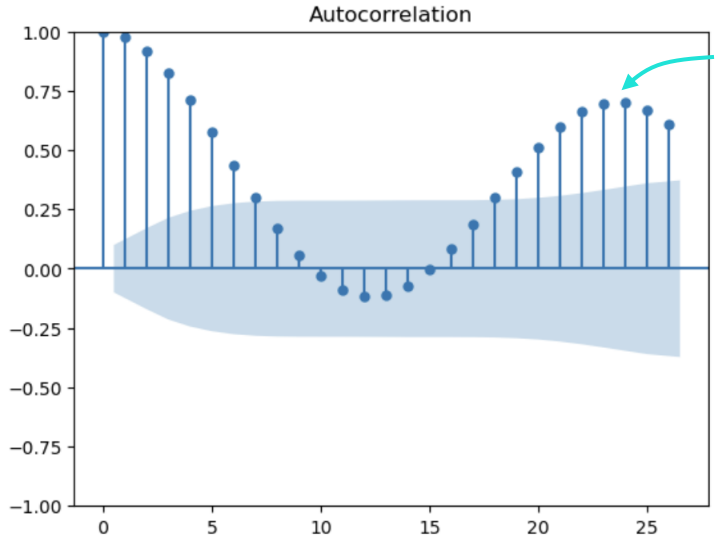
Forecasting

An **autocorrelation chart** calculates the correlation between time-series data and lagged versions of itself, then plots those correlations

- This allows you to visualize which lags are highly correlated with the original data, and reveals the length (or period) of the seasonal trend




```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(weather);
```



This peak indicates a seasonal pattern every 24 periods (hours)

If your season has trend, **you must difference your time series first,** which can be be done with the .diff() method in Pandas

ASSIGNMENT: DECOMPOSITION

NEW MESSAGE

August 5, 2023

From: **Tammy Tiempo** *(Financial Analyst)*

Subject: **Decomposition**

Hi there!

Thanks for your help with smoothing.

In a related project, we're looking at weather in neighboring Spain and we want to confirm the seasonality of weather on both a daily and annual basis.

Can you decompose hourly and monthly weather and plot ACF charts for them?

Thanks!

10_time_series_assignments.ipynb

Reply Forward

Key Objectives

1. Use time series decomposition to understand the trend, seasonality and noise of time series data
2. Use an ACF chart to estimate the seasonal window for time series data

FORECASTING

Time Series Data

Smoothing

Decomposition

Forecasting

Time series **forecasting** lets you predict *future values* using historical data

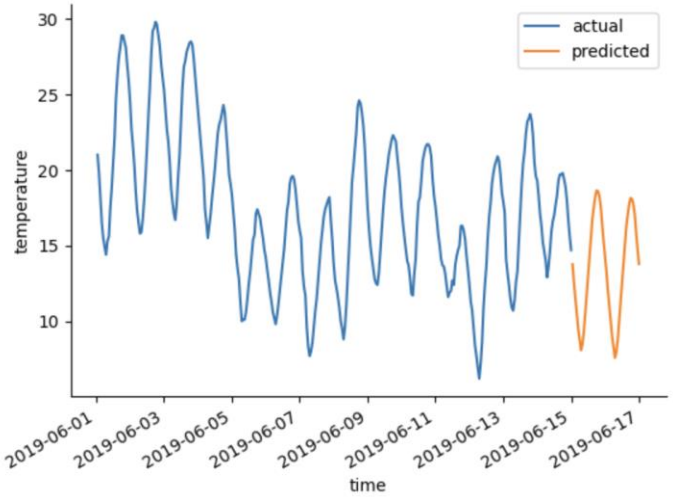
- Forecasting models use existing trends & seasonality to make accurate future predictions

Common Forecasting Techniques:

- Linear Regression
- Facebook Prophet

We'll cover these in the course

- Holt-Winters
- ARIMA Modeling
- LSTM (deep learning approach)



Forecasts get less accurate the further out we are trying to predict, so think carefully about how far in advance you really need to forecast in the context of the problem

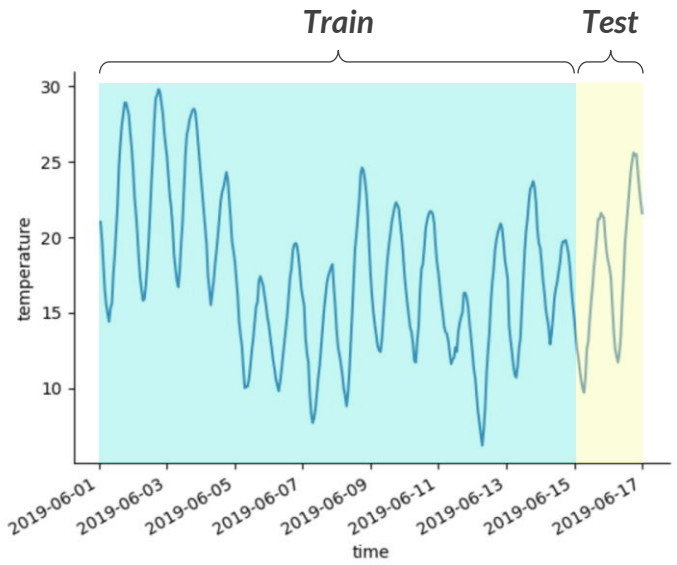
DATA SPLITTING

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

Time series **data splitting** does not follow traditional train/test splits:

- Instead of random splits, we'll need to split by points in time to mimic forecasting the future
- The training data set should be at least as long as your desired forecast window (test data)
- You may need to change the period to get enough holdout data

EXAMPLE *Forecasting the next 48 hours of weather*




```
weather_train = weather["2019-06-01": "2019-06-14"]
weather_test = weather["2019-06-15": "2019-06-16"]

weather_train.head(2)
```

```
time
2019-06-01 00:59:59    21.0
2019-06-01 01:59:59    19.8
Name: temperature, dtype: float64
```

```
weather_test.tail(2)
```

```
time
2019-06-16 22:59:59    22.4
2019-06-16 23:59:59    21.6
Name: temperature, dtype: float64
```

LINEAR REGRESSION WITH TREND & SEASON

Time Series Data

Smoothing

Decomposition

Forecasting

It common to build forecasting models using **linear regression** by creating "trend" and "seasonal" dummy variables

- The **trend** variable is a sequence of numbers that increments by 1 unit every period
- The **seasonal** variables are dummies for every period in the season

```
weather = pd.get_dummies(
    weather.assign(
    trend = weather.index + 1,
    hour = weather["time"].dt.hour.astype("string")
    ),
    drop_first=True
)

weather.head()
```

This creates a "trend" variable and 24 "seasonal" dummy variables (1 per hour in a day)

	time	temperature	trend	hour_1	hour_10	...	hour_8	hour_9
0	2019-01-01 00:59:59	3.1	1	0	0		0	0
1	2019-01-01 01:59:59	2.9	2	1	0		0	0
2	2019-01-01 02:59:59	3.1	3	0	0		0	0
3	2019-01-01 03:59:59	1.9	4	0	0		0	0
4	2019-01-01 04:59:59	0.7	5	0	0		0	0

FITTING THE MODEL

Time Series Data

Smoothing

Decomposition

Forecasting

You can **fit the model** on the training data like a regular linear regression model

- It violates the assumptions of linear regression, but it's often very effective

```
import statsmodels.api as sm

weather_train = weather["2019-05-17": "2019-05-31"]
weather_test = weather["2019-06-01": "2019-06-03"]

y = weather_train["temperature"]
X = sm.add_constant(weather_train.iloc[:, 1:])

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

model.summary()
```

OLS Regression Results

Dep. Variable:	temperature	**R-squared:**	0.764
Model:	OLS	**Adj. R-squared:**	0.748
Method:	Least Squares	**F-statistic:**	45.31
Date:	Mon, 21 Aug 2023	**Prob (F-statistic):**	3.71e-90
Time:	19:13:56	**Log-Likelihood:**	-847.76
No. Observations:	360	**AIC:**	1746.
Df Residuals:	335	**BIC:**	1843.
Df Model:	24		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-58.8630	4.662	-12.626	0.000	-68.034	-49.692
trend	0.0212	0.001	15.793	0.000	0.019	0.024
hour_1	-1.1612	0.965	-1.203	0.230	-3.060	0.737
hour_10	-0.3855	0.965	-0.399	0.690	-2.284	1.513
hour_11	0.9933	0.965	1.029	0.304	-0.905	2.892
hour_12	2.4787	0.965	2.568	0.011	0.580	4.377

The temperature increases by .02 degrees every hour

The temperature is 2.47 degrees higher than the reference level at which is hour_0, or midnight

SCORING THE MODEL

Time Series Data

Smoothing

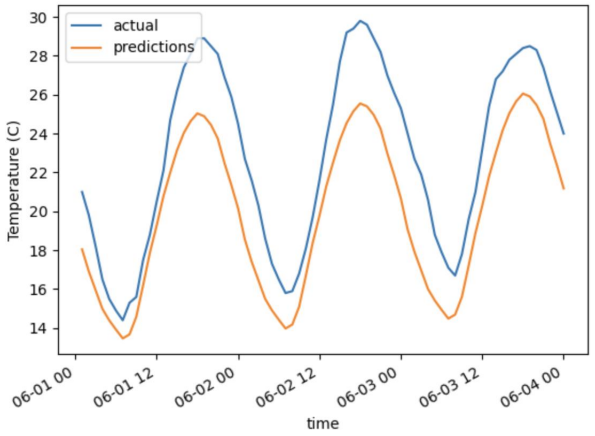
Decomposition

Forecasting

To **score the model**, you can plot the predictions against the actual values for the test data and calculate error metrics like MAE and MAPE

- The Mean Absolute Percentage Error (MAPE) is is calculated by finding the average percent error of all the data points *(it's essentially MAE converted to a percentage)*

```
weather_test = (
    weather_test
     .assign(predictions = model.predict(sm.add_constant(weather_test.iloc[:, 1:])))
     .rename({"temperature": "actual"}, axis=1)
     .loc[:, ["actual", "predictions"]]
)

weather_test.plot(ylabel="Temperature (C)")
```




```
from sklearn.metrics import mean_absolute_percentage_error as mape
from sklearn.metrics import mean_absolute_error as  mae

print(
    f"MAE: {mae(weather_test['actual'], weather_test['predictions'])}",
    f"MAPE: {mape(weather_test['actual'], weather_test['predictions'])}",
)
```

```
MAE: 2.9616765872741584 MAPE: 0.12747465248026735
```

The model is wrong by 2.96 degrees on average, or by 12.7%

FACEBOOK PROPHET

Time Series Data

Smoothing

Decomposition

Forecasting

Facebook Prophet is a forecasting package available in Python and R that was developed by Meta's data science research team

Instead of OLS, it uses an additive regression model with three main components:

1. Growth curve trend *(created by automatically detecting changepoints in the data)*
2. Yearly, weekly, and daily seasonal components
3. User-provided list of important holidays

PRO TIP: Prophet is a relatively sophisticated and easy to use package, making it ideal for quick forecasting activities

FACEBOOK PROPHET

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

EXAMPLE *Forecasting the next 48 hours of weather*

```
# !conda install prophet --y

from prophet import Prophet

weather_train = weather["2019-05-17": "2019-05-31"].reset_index()
weather_test = weather["2019-06-01": "2019-06-03"].reset_index()

weather_train = weather_train.rename({"time": "ds", "temperature": "y"}, axis=1)

model = Prophet()
model.fit(weather_train)
```

Install Facebook Prophet

Import the package

Split the data

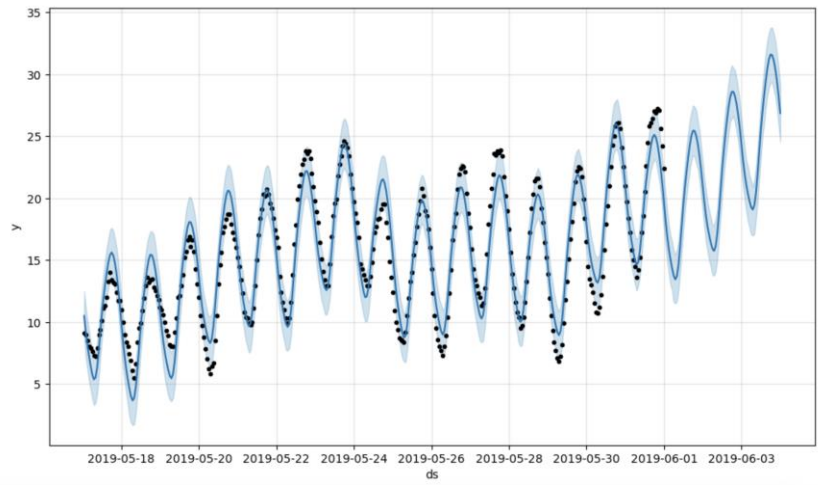
Rename the x and y variables to "ds" and "y"

Fit the model

```
20:16:53 - cmdstanpy - INFO - Chain [1] start processing
20:16:53 - cmdstanpy - INFO - Chain [1] done processing
<prophet.forecaster.Prophet at 0x174e33490>
```

```
future = model.make_future_dataframe(periods=72, freq="H")

forecast = model.predict(future)

model.plot(forecast)
```

Specify the number of periods and period frequency to predict, and plot the predictions



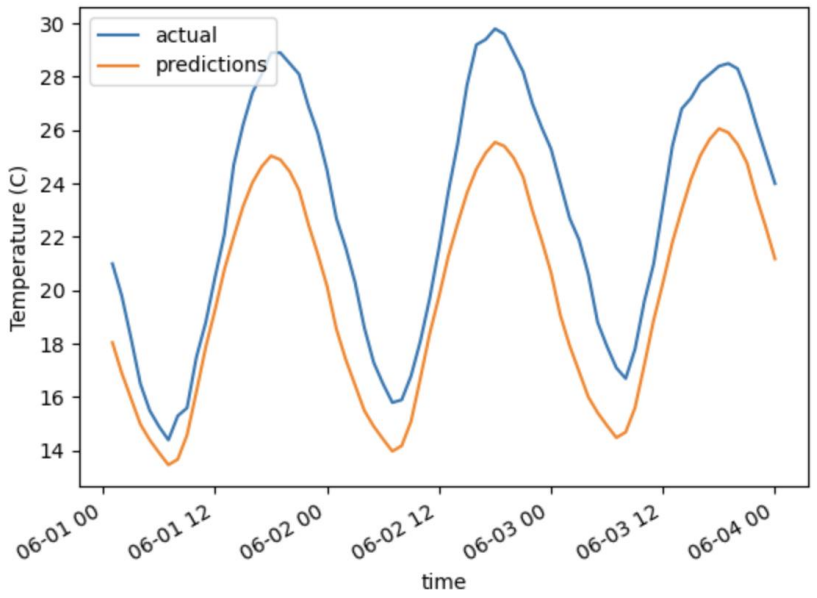
LINEAR REGRESSION VS. PROPHET

- Time Series Data
- Smoothing
- Decomposition
- Forecasting

Linear Regression

MAE: 2.96

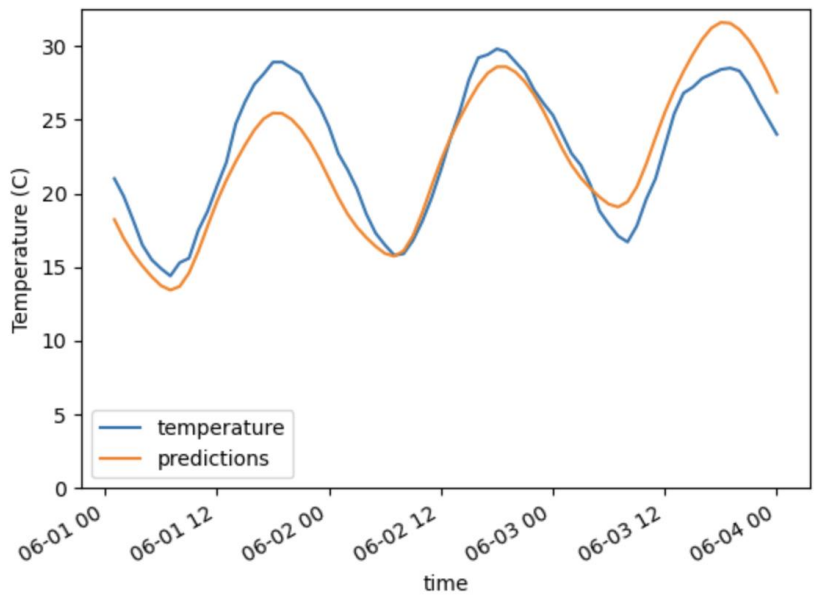
MAPE: 0.127



Facebook Prophet

MAE: 2.09

MAPE: .091



ASSIGNMENT: FORECASTING

NEW MESSAGE

August 10, 2023

From: **Tammy Tiempo** *(Financial Analyst)*

Subject: **Forecasting**

Hi there!

We're looking at evaluating two forecasting methods to propose to an airline client.

Can you take a look at historical airline passenger growth and compare the accuracy of forecasts for linear regression and Facebook Prophet?

I've left a few more details in the notebook.

Thanks!

10_time_series_assignments.ipynb

Reply | Forward

Key Objectives

1. Split time series data into train and test datasets
2. Fit time series models with linear regression and Facebook Prophet
3. Compare the accuracy of the two forecasting methods, calculating MAE and MAPE

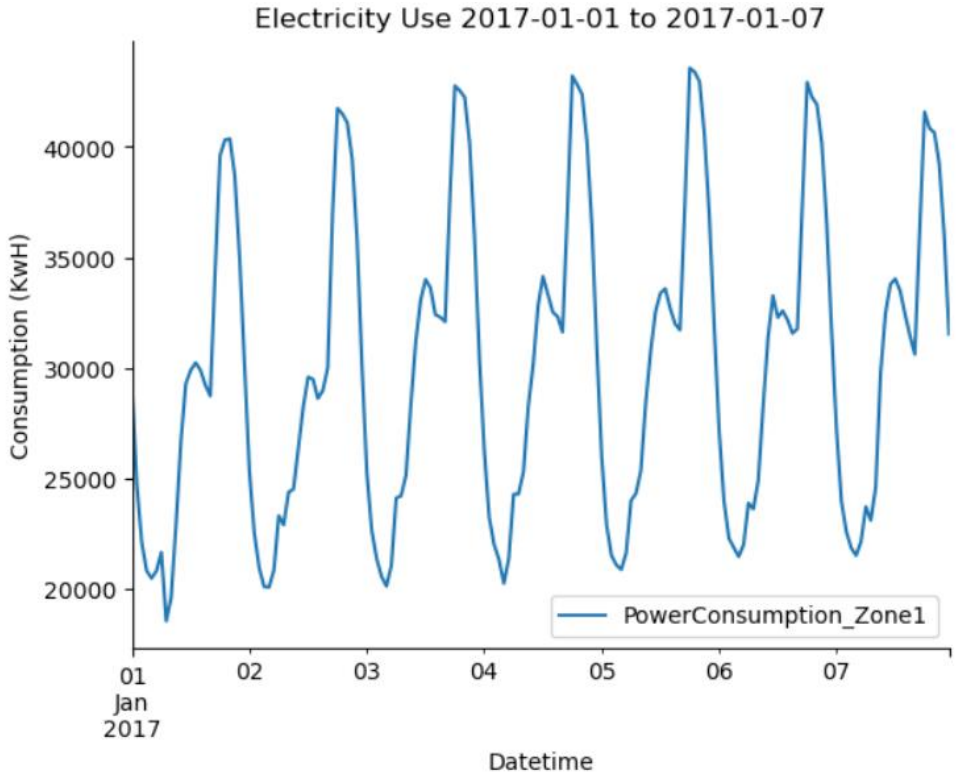
KEY TAKEAWAYS

- **Time series data** requires each row to represent a unique moment in time
  - *It's important to decide on the units, or row granularity, of your time series data (years, months, etc.)*
- Time series **smoothing** allows you to visualize trends in the time series data
  - *Common techniques include moving average and exponential smoothing*
- **Decomposition** breaks the data down into trend, season, and random noise
  - *Time series data can be decomposed in an additive or multiplicative fashion*
- Time series **forecasting** allows you to predict future values in time
  - *Common techniques include linear regression (even though the assumptions are violated) and Facebook Prophet*

PROJECT 2: FORECASTING

PROJECT DATA: ELECTRICITY CONSUMPTION

Datetime	PowerConsumption_Zone1
2017-01-01 00:00:00	29197.974683
2017-01-01 01:00:00	24657.215190
2017-01-01 02:00:00	22083.037973
2017-01-01 03:00:00	20811.139240
2017-01-01 04:00:00	20475.949367
2017-01-01 05:00:00	20807.088607
2017-01-01 06:00:00	21648.607595
2017-01-01 07:00:00	18540.759495
2017-01-01 08:00:00	19605.063290
2017-01-01 09:00:00	22905.316455



MAVEN CONSULTING GROUP

ASSIGNMENT: FINAL PROJECT

NEW MESSAGE

August 14, 2023

From: **Tammy Tiempo** *(Financial Analyst)*

Subject: **Forecasting Electricity Consumption**

Hello,

Can you build a simple forecast for our Moroccan Electricity consumption? I'd like to see a linear regression model compared to Facebook Prophet, now that we know they're both reasonable forecasting methods.

I'm hoping we can get a forecast accurate enough that allows us to properly estimate the demand for electricity, which will help us properly price it during peak demand.

-Tammy

11_time_series_project.ipynb

Reply Forward

Key Objectives

1. Explore and manipulate time series data
2. Perform time series data splitting
3. Build and compare the predictive accuracy of forecasting models

MAVEN CONSULTING GROUP