CLASSIFICATION 101

CLASSIFICATION 101

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

TOPICS WE'LL COVER:

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

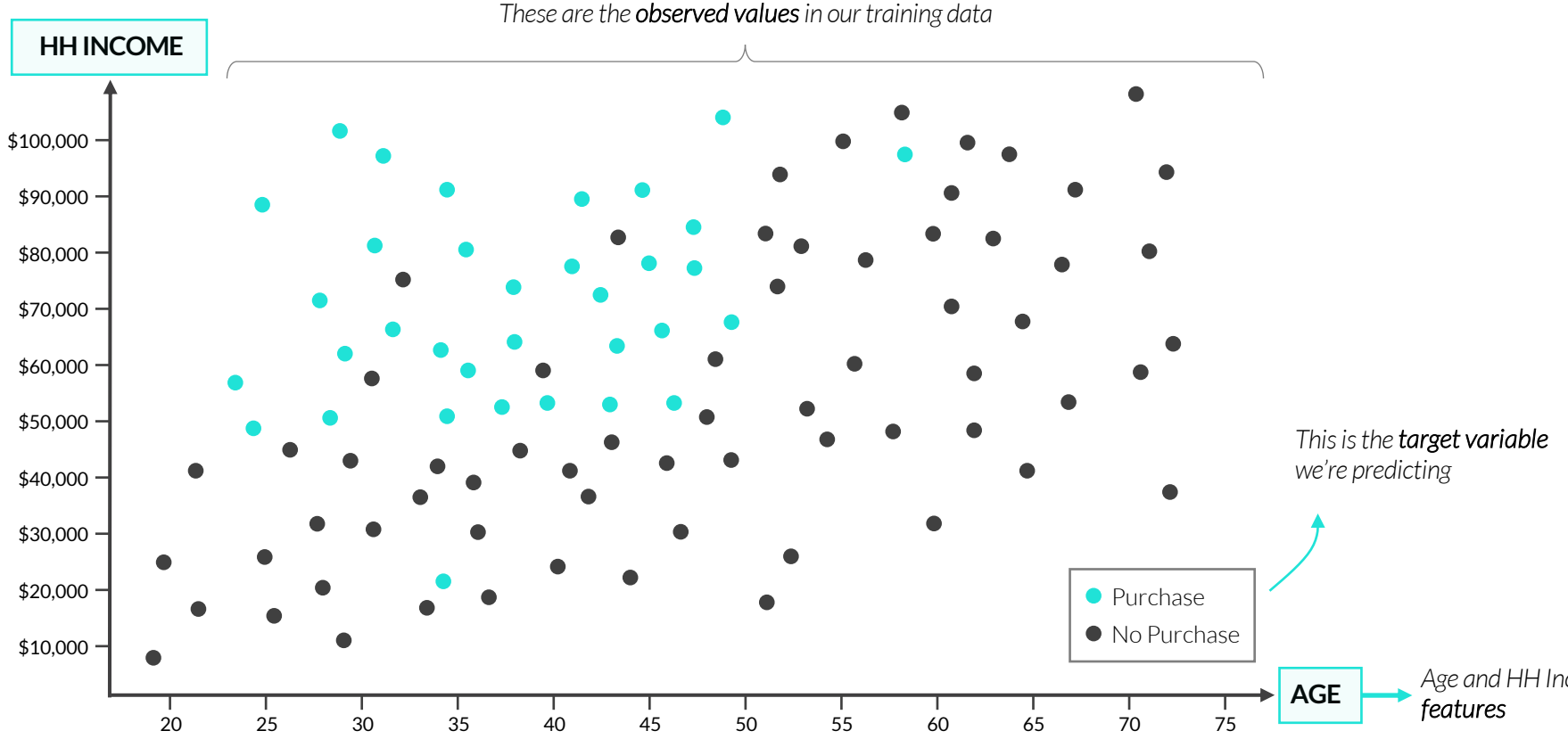
GOALS FOR THIS SECTION:

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

CLASSIFICATION 101

Classification modeling is a supervised learning technique used to predict a categorical variable *(target)* by modeling its relationship with other variables *(features)*

EXAMPLE *Predicting whether a customer will purchase based on demographic data*



CLASSIFICATION 101

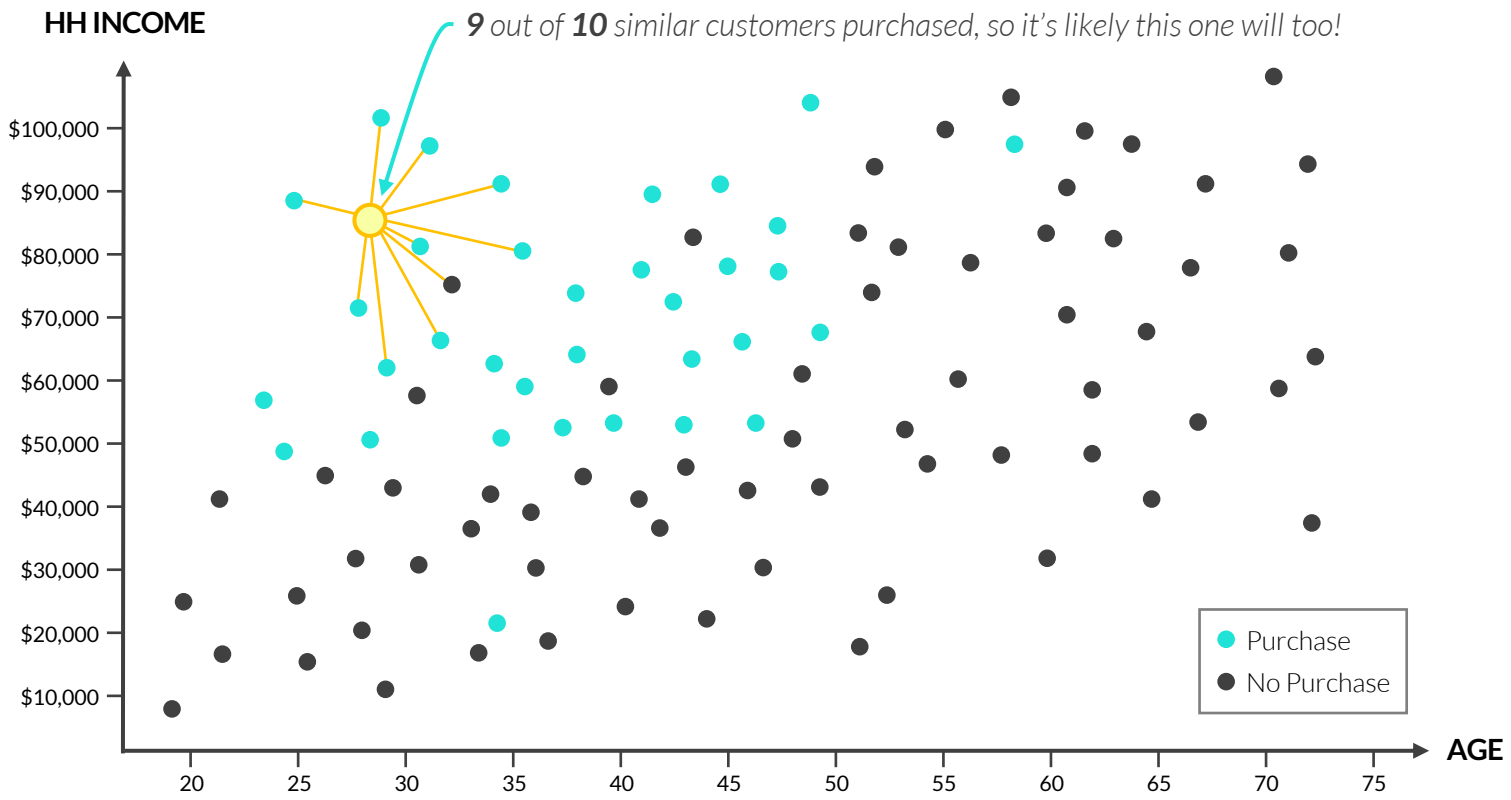
Classification modeling is a supervised learning technique used to predict a categorical variable *(target)* by modeling its relationship with other variables *(features)*

EXAMPLE *Predicting whether a customer will purchase based on demographic data*

CLASSIFICATION 101

Classification modeling is a supervised learning technique used to predict a categorical variable *(target)* by modeling its relationship with other variables *(features)*

EXAMPLE *Predicting whether a customer will purchase based on demographic data*



CLASSIFICATION 101

Classification analysis is a supervised learning technique used to predict a categorical variable *(target)* by modeling its relationship with 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
- Classification 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
- Classification helps understand how the features impact, or *predict*, the target

CLASSIFICATION 101

EXAMPLE

Predicting whether someone who has seen a website ad will purchase based on demographic data

User ID	Gender	Age	EstimatedSalary	Purchased
15589449	Male	39	106000	1
15694829	Female	32	150000	1
15668575	Female	26	43000	0
15602373	Male	29	75000	0
15635893	Male	60	42000	1
15593014	Male	28	59000	0

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

*Since this is **categorical**, we'll use classification to predict it*

***Gender**, **age**, and **estimated salary** are all **features**, since they can help us explain, or predict, whether a customer will purchase*

CLASSIFICATION 101

EXAMPLE

Predicting whether someone who has seen a website ad will purchase based on demographic data

User ID	Gender	Age	EstimatedSalary	Purchased
15589449	Male	39	106000	1
15694829	Female	32	150000	1
15668575	Female	26	43000	0
15602373	Male	29	75000	0
15635893	Male	60	42000	1
15593014	Male	28	59000	???

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

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

This is what our model will predict!

GOALS OF CLASSIFICATION

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

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

PREDICTION

- Used to **predict** the target as accurately as possible
- *"What is the probability a customer will purchase based on what we know about them?"*

INFERENCE

- Used to **understand the relationships** between the features and target
- *"How much does a customer's income impact their likelihood to purchase?"*

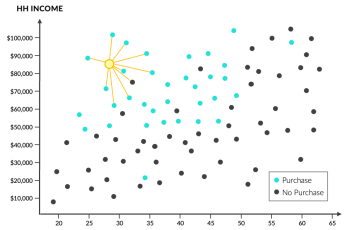
You often need to **strike a balance** between these goals – a model that is very inaccurate won't be too trustworthy for inference, and understanding the impact that variables have on predictions can help make them more accurate

TYPES OF CLASSIFICATION

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

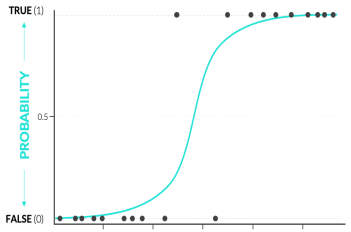
These are some of the most common **types of classification** modeling techniques:

K-Nearest Neighbors



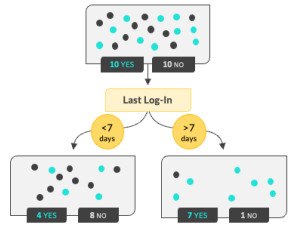
Classifies observations based on characteristics of nearby points

Logistic Regression



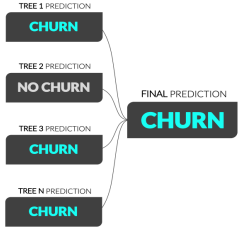
Predicts the probability that a datapoint is a given class

Decision Trees



Splits data by maximizing the difference between groups

Ensemble Methods



Uses the outputs of multiple models to improve accuracy

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

CLASSIFICATION MODELING WORKFLOW

- Classification 101
- Goals of Classification
- Types of Classification
- 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 for input into an ML model

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

Applying Algorithms

Build classification models from training data

- KNN
- Logistic Regression
- Decision Trees
- Ensemble Methods

Model Evaluation

Evaluate model fit on training & validation data

- Accuracy, Precision, F1
- Confusion Matrix
- Validation Performance

Model Selection

Pick the best model to deploy and identify insights

- Test performance
- Interpretability
- Scoring Speed

KEY TAKEAWAYS

- Classification modeling is used to **predict categorical values**
  - *There are several types of classification models, and we'll cover the most used ones in this course*
- The **target** is the value we want to predict, and **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*
- Classification 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 PREP & EDA

PRE-MODELING PREP & EDA

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

TOPICS WE'LL COVER:

- EDA for Classification
- Exploring the Target
- Exploring the Features
- Correlation
- Exploring Relationships
- Feature Engineering
- Data Splitting
- Preparing for Modeling

GOALS FOR THIS SECTION:

- Visualize and explore the target and features
- Explore relationships between the target and features, as well as between features themselves
- Learn how to identify multicollinear variables
- Review the data prep steps required for modeling, including feature engineering and data splitting

EDA FOR CLASSIFICATION

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

- EDA lets you identify and 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 classification**, it's important 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

DEFINING A TARGET

Defining a target variable for classification often requires data manipulation

- You may want to convert a continuous variable into a binary variable
- You may also want to collapse multiple categories into fewer classes

EXAMPLE *Converting a continuous numeric variable to binary*

Age	Gender	Amount Sold
77	Male	150
22	Female	0
33	Male	0
42	Female	1800

Age	Gender	Purchased?
77	Male	1
22	Female	0
33	Male	0
42	Female	1

We used a threshold of 0 in this example, but other problems might need other cutoff points!

DEFINING A TARGET

Defining a target variable for classification often requires data manipulation

- You may want to convert a continuous variable into a binary variable
- You may also want to collapse multiple categories into fewer classes

EXAMPLE | *Converting many to few categories*

Age	Gender	Item Category
77	Male	Groceries
22	Female	Beverages
33	Male	Fishing
42	Female	Sports
22	Male	Televisions
37	Male	Video Games

→

Age	Gender	Item Category
77	Male	Food & Bev
22	Female	Food & Bev
33	Male	Sports
42	Female	Sports
22	Male	Electronics
37	Male	Electronics

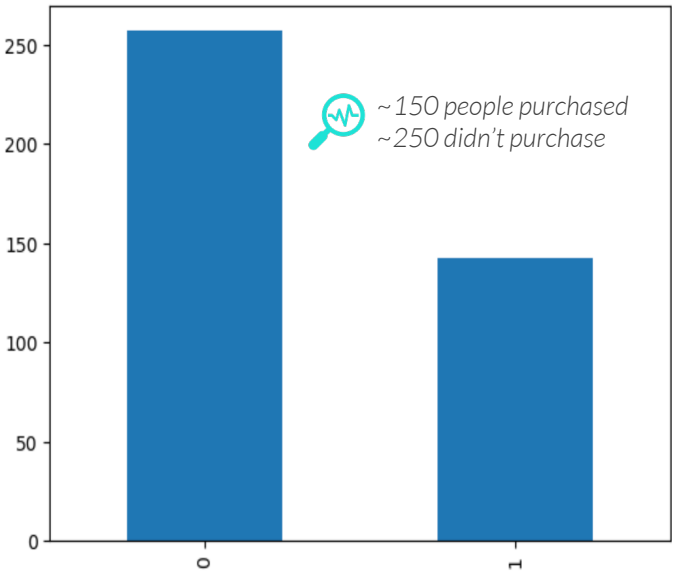
Most classification algorithms can be applied to more than two classes, but **multi-class problems are more challenging to model accurately and evaluate**

EXPLORING THE TARGET

Exploring the target variable lets you understand how often each category occurs, and whether you might have imbalanced data

- **Bar charts** are great tools for exploring classification targets

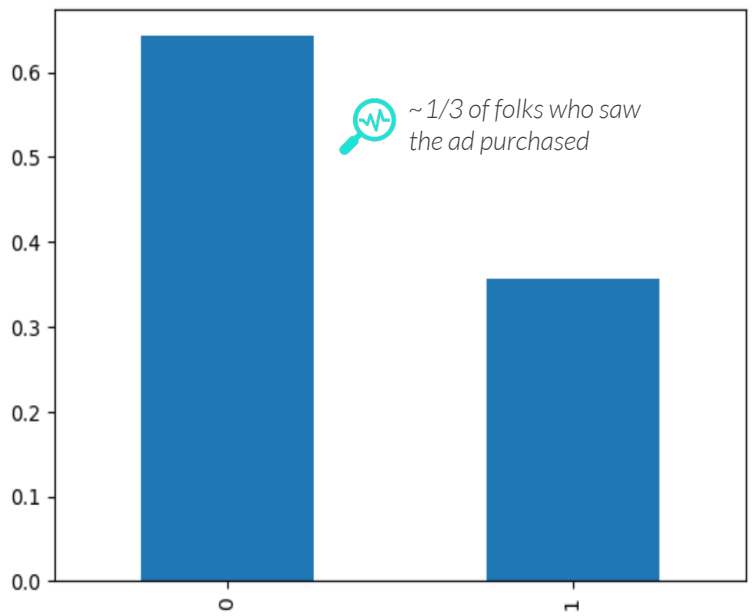
```
ads["Purchased"].value_counts().plot.bar();
```



*Our column is binary, which is ideal for modeling (**0**=Not Purchased, **1**=Purchased)*

***normalize=True** returns the value as percentages*

```
ads["Purchased"].value_counts(normalize=True).plot.bar();
```

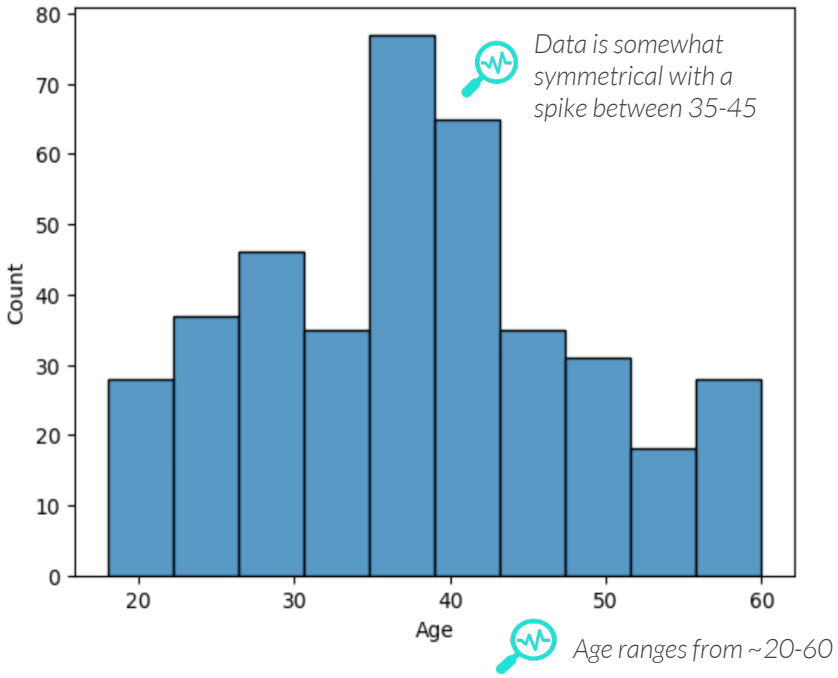


EXPLORING THE FEATURES

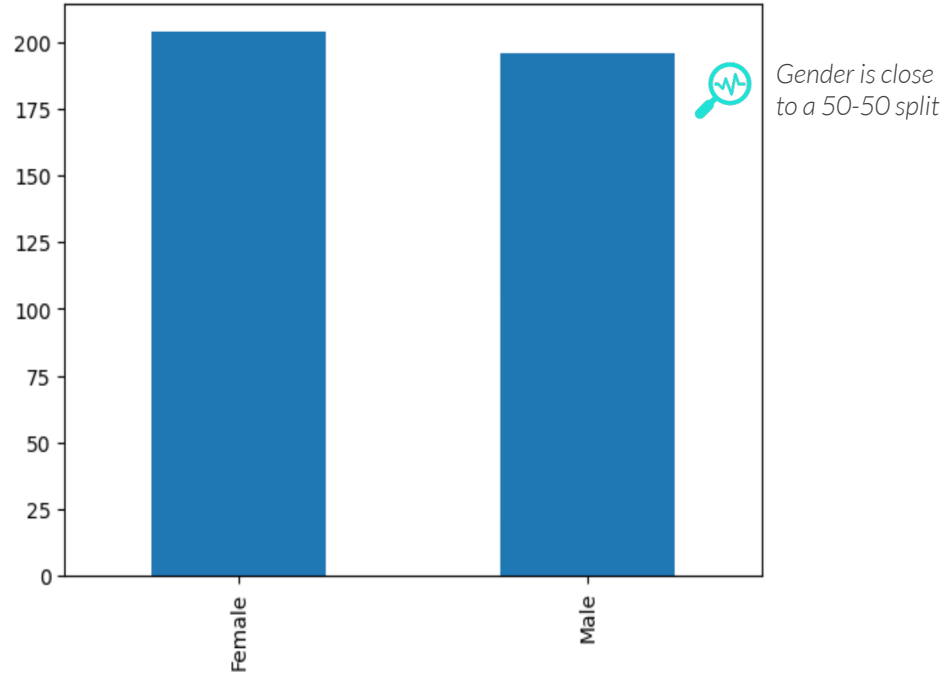
Exploring the features helps you understand them and start to get a sense of the transformations you may need to apply to each one

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

```
sns.histplot(ads["Age"]);
```




```
ads["Gender"].value_counts().plot.bar();
```

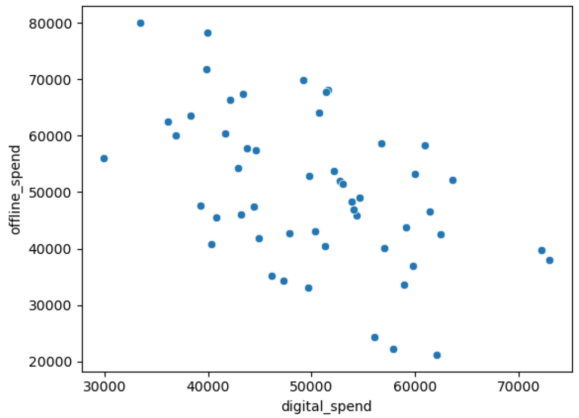


CORRELATION

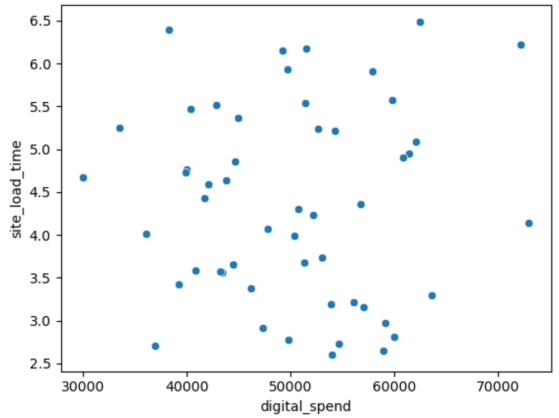
Correlation (r) measures the strength and direction of a relationship (-1 to 1)

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

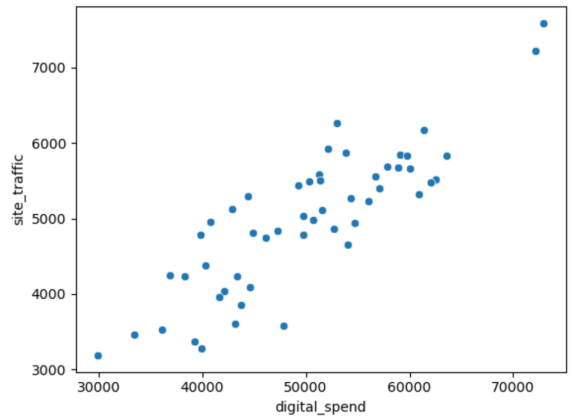
r = -0.499

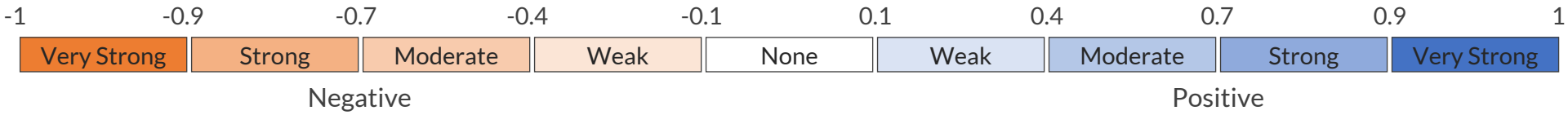


r = 0.008



r = 0.858



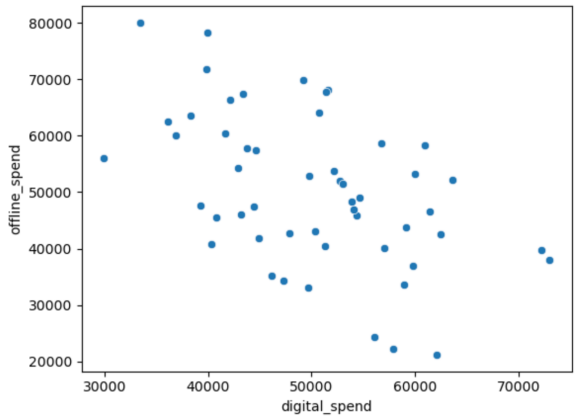


CORRELATION

Correlation (r) measures the strength and direction of a relationship (-1 to 1)

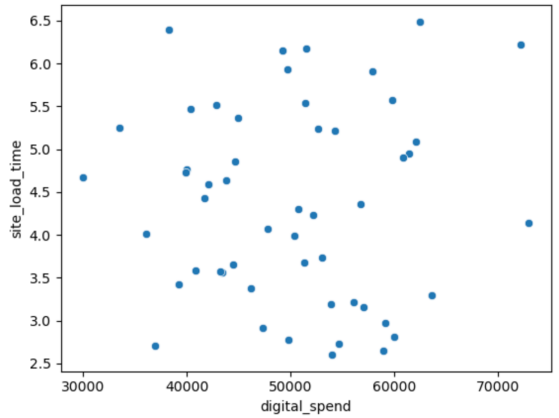
- You can use the **.corr()** method to calculate correlations in Pandas – df["col1"].corr(df["col2"])
- Multicollinearity refers to two highly correlated features being used to predict the same target

r = -0.499



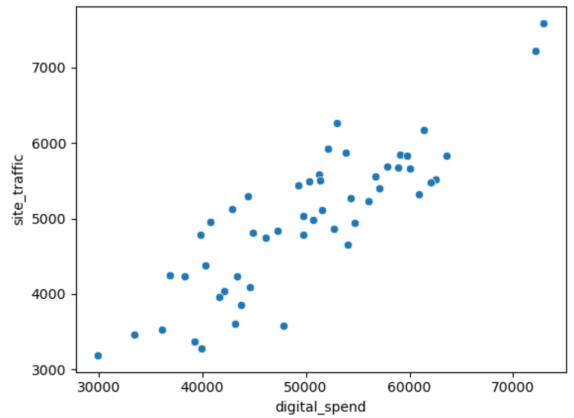
Moderate negative correlation

r = 0.008



No correlation

r = 0.858



Strong positive correlation

PRO TIP: Correlation can help screen for promising numeric features to use in your model, and identify potentially multi-collinear features that might lead to instability in your models

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 the correlation matrix in **sns.heatmap()** to make it easier to interpret

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

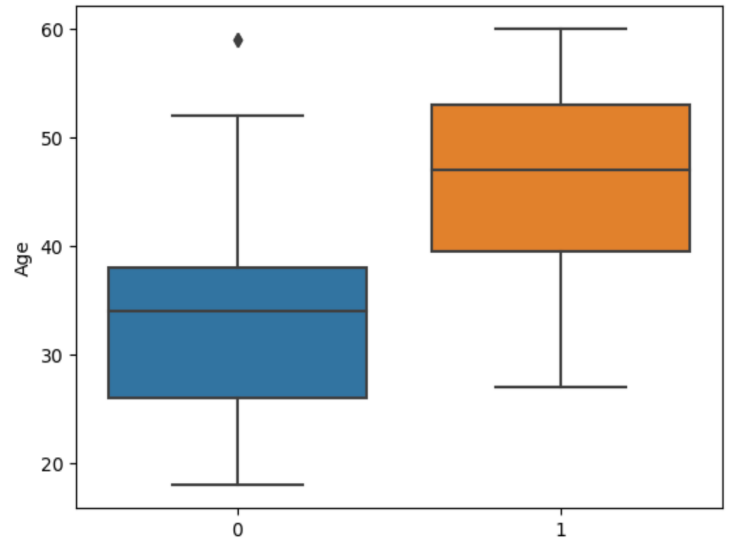
Age & Estimated Salary have promising correlations with Purchased. UserID, not surprisingly, does not.

FEATURE-TARGET RELATIONSHIPS

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

- Use **boxplots** for numeric features and **bar charts** for categorical features

```
sns.boxplot(ads, x="Purchased", y="Age");
```

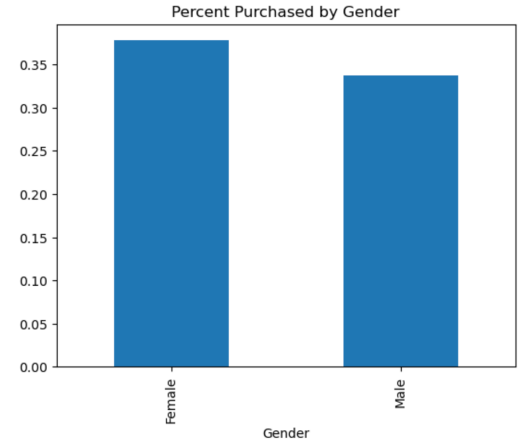



```
ads.groupby("Purchased")["Age"].mean()
```

```
Purchased
0    32.793774
1    46.391608
Name: Age, dtype: float64
```

Age for purchasers is significantly higher than non-purchasers

```
(ads
 .groupby("Gender")
 .agg({"Purchased": "mean"})
 .plot
 .bar(title="Percent Purchased by Gender", legend=False)
);
```



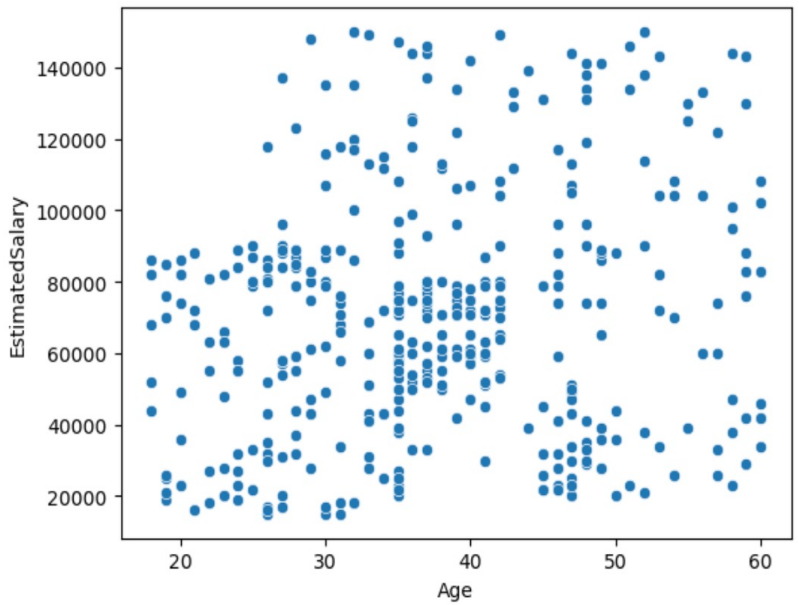
Purchase rate differs by gender:

- *This is a modest gap, so it may or may not be a useful predictor*
- *This gap may also be due to other features like age and income*

FEATURE-FEATURE RELATIONSHIPS

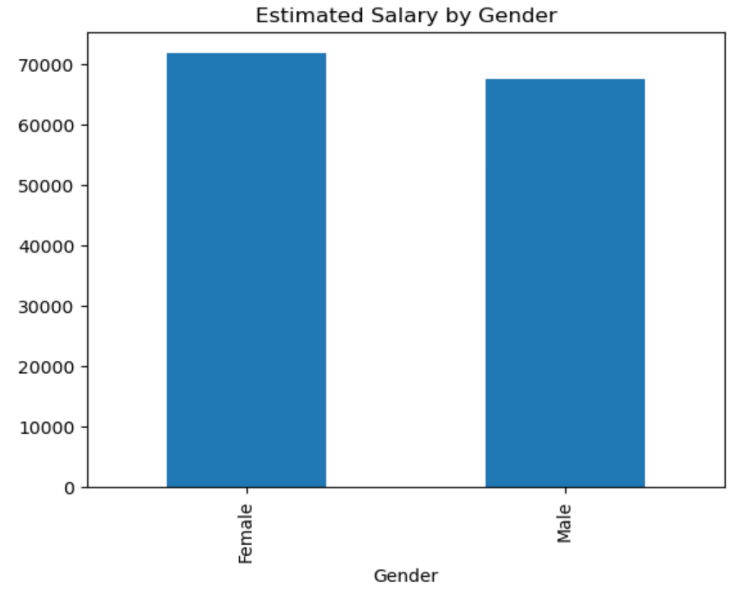
Exploring **feature-feature relationships** helps identify highly correlated features, which can cause problems with classification models *(more on this later!)*

```
sns.scatterplot(ads, x="Age", y="EstimatedSalary");
```



No clear relationship between age & salary, which means these features will provide unique information to our model

```
(ads
 .groupby("Gender")
 .agg({"EstimatedSalary": "mean"})
 .plot
 .bar(title="Estimated Salary by Gender", legend=False)
);
```



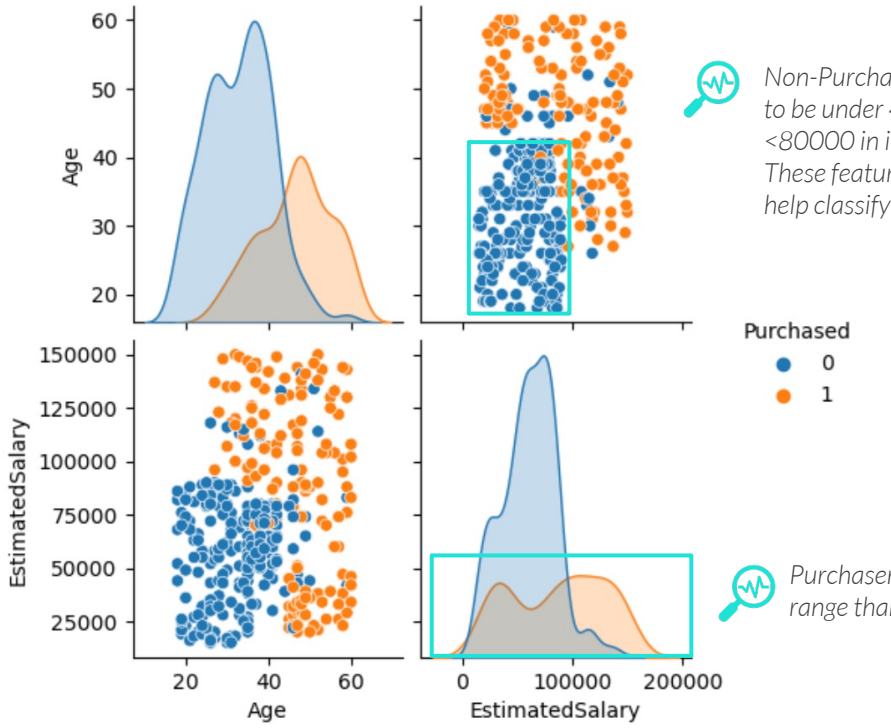
Females have higher salaries than males, which may indicate that salary, not gender, is driving purchase behavior!

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

- Specify **hue=target** to highlight differences between target categories

```
sns.pairplot(ads.drop("User ID", axis=1), hue="Purchased");
```



Non-Purchasers tend to be under 40 and <80000 in income. These features should help classify our data!

Purchasers have a higher salary range than non-purchasers

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

FEATURE ENGINEERING

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

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

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0

Because it's stored as text, "Gender" can't be input into a model. We may also want to engineer some other features to help our model fit the data (more later!)

```
ads = pd.get_dummies(
    ads.assign(
        age45 = np.where(ads["Age"] > 45, 1, 0),
        salary90 = np.where(ads["EstimatedSalary"] > 90000, 1, 0),
        agexsal = ads["Age"] * ads["EstimatedSalary"]
    ),
    drop_first=True
)

ads.head()
```

	User ID	Age	EstimatedSalary	Purchased	age45	salary90	agexsal	Gender_Male
0	15624510	19	19000	0	0	0	361000	True
1	15810944	35	20000	0	0	0	700000	True
2	15668575	26	43000	0	0	0	1118000	False
3	15603246	27	57000	0	0	0	1539000	False
4	15804002	19	76000	0	0	0	1444000	True

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

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

Math Calculations

- Combining features
- Interaction terms

Category Mappings

- Binary columns
- Dummy variables
- Binning

Scaling

- Standardization

We'll do an in-depth review of these

Group Calculations

- Aggregations
- Ranks within groups

DateTime Calculations

- Days from "today"
- Time between dates

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!

ARITHMETIC CALCULATIONS

Arithmetic calculations can be used to combine variables, especially for columns that don't mean much unless combined with others

- In other cases, combining features can help improve model stability or capture **interaction effects** – feature-target relationships that change based on the value of another feature

```
houses
```

	PRICE	BEDS	BATHS
0	399000	3	2
1	99900	2	1
2	539000	3	1
3	299000	2	1
4	320000	2	2
5	699900	4	3
6	295000	3	3
7	245900	1	1
8	480000	5	3
9	375000	5	2
10	770000	3	1

```
houses["total_rooms"] = houses["BEDS"] + houses["BATHS"]
houses["bed_bath_ratio"] = houses["BEDS"] / houses["BATHS"]
houses["bed_bath_interaction"] = houses["BEDS"] * houses["BATHS"]

houses
```

	PRICE	BEDS	BATHS	total_rooms	bed_bath_ratio	bed_bath_interaction
0	399000	3	2	5	1.500000	6
1	99900	2	1	3	2.000000	2
2	539000	3	1	4	3.000000	3
3	299000	2	1	3	2.000000	2
4	320000	2	2	4	1.000000	4
5	699900	4	3	7	1.333333	12
6	295000	3	3	6	1.000000	9
7	245900	1	1	2	1.000000	1
8	480000	5	3	8	1.666667	15
9	375000	5	2	7	2.500000	10
10	770000	3	1	4	3.000000	3

SCALING & STANDARDIZATION

Scaling requires setting all input features on a similar scale

- **Standardization** transforms all values to have a mean of 0 and standard deviation of 1
- Standardization is critical for k-nearest neighbors and regularized logistic regression

```
houses
```

	PRICE	BEDS
0	399000	3.0
1	99900	2.0
2	539000	3.0
3	299000	2.0
4	320000	2.0
5	699900	4.0
6	295000	3.0
7	245900	1.0
8	480000	5.0
9	375000	5.0
10	770000	3.0

```
(houses - houses.mean()) / houses.std()
```

	PRICE	BEDS
0	-0.061292	0.000000
1	-1.569570	-0.790569
2	0.644689	0.000000
3	-0.565564	-0.790569
4	-0.459667	-0.790569
5	1.456063	0.790569
6	-0.585735	0.000000
7	-0.833333	-1.581139
8	0.347168	1.581139
9	-0.182317	1.581139
10	1.809558	0.000000

*Later we'll use the **.StandardScaler()** function from the **sklearn** library for this purpose*

Standardization equation

$$\frac{x - x_{mean}}{x_{std}}$$

PRO TIP: Standardization is usually the last data prep step you should perform. (after all other feature engineering and data splitting). We'll walk through the proper workflow when we get to kNN!

BINARY COLUMNS

Categorical features must be converted to numeric before modeling

- In their simplest form, they can be represented with binary columns (0/1)

```
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

DUMMY VARIABLES

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

For some models (like logistic regression), 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"

Index	Carat	Clarity
0	0.52	VVS1
1	1.50	VVS1
2	0.91	SI1
3	1.04	SI2
4	0.30	SI1

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

Index	Carat	Clarity	VVS1	SI1	SI2
0	0.52	Blue	1	0	0
1	1.50	Blue	1	0	0
2	0.91	White	0	1	0
3	1.04	Yellow	0	0	1
4	0.30	White	0	1	0

It only takes 2 columns to distinguish 3 values, so you might sometimes see one less column

DUMMY VARIABLES

For some models (like logistic regression), 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

BINNING CATEGORICAL DATA

Adding dummy variables for each categorical column can lead to wide data sets and increase model variance; **binning** can solve this and improve interpretability

- After creating new categorical groups or "bins", you can create dummy variables for each

```
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
```

Rare categories like "I1" category can be lead to unstable or inaccurate estimates – like in many other areas of data analysis, not having a sufficiently large sample can lead to incorrect estimates!

```
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/bin the 8 clarity values into just 3 distinct buckets

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

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

BINNING CATEGORICAL DATA

Adding dummy variables for each categorical column can lead to wide data sets and increase model variance; **binning** can solve this and improve interpretability

- After creating new categorical groups or "bins", you can create dummy variables for each

EXAMPLE *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

EXAMPLE *Binning*

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

FEATURE ENGINEERING TIPS

- Anyone (including machines) can apply algorithms, but only those with **domain expertise** can engineer relevant features, which is what makes a great model
- In general, you want data to be long rather than wide (**many rows, few columns**)
- Once you start modeling, you'll likely find things you missed during data prep and will **continue to engineer features** and repeat the process of gathering, cleaning, and exploring the data

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

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

- First, create separate DataFrames for your features and target columns
- train_test_split(feature_df, target_column, test_pct, random_state)

```
from sklearn.model_selection import train_test_split

X = pd.get_dummies(ads[["Gender", "Age", "EstimatedSalary"]], drop_first=True)
y = ads["Purchased"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2023)

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

```
Training Set Rows: 320 Test Set Rows: 80
```

Perfect 80/20 split!

4 outputs:

- Training set for features (**X_train**)
- Test set for features (**X_test**)
- Training set for target (**y_train**)
- Test set for target (**y_test**)

4 inputs:

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

PREPARING FOR MODELING

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

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

Data prepared for **EDA**

Age	Gender	Purchased?
77	Male	Purchased
22	Female	No Purchase
33		No Purchase
42	Female	Purchased

Data prepared for **modeling**

Index	Age	Female	Missing	Purchased?
0	77	0	0	1
1	22	1	0	0
2	33	0	1	0
3	42	1	0	1

Features *Target*

**NOTE: Tree-based models can handle missing values without treatment, while KNN and Logistic Regression models cannot*

KEY TAKEAWAYS

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

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

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

- *Features that are highly correlated with your target or have different frequencies when split by category are likely to be strong predictors*
- *Features that are highly correlated with each other (multicollinearity) can cause problems in a model*

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

- *Use feature engineering techniques to create new features, and split the data for testing and training*
- *Data must be stored in a single table, each column must be numeric, and missing values must be addressed*

K-NEAREST NEIGHBORS

K-NEAREST NEIGHBORS

In this section we'll build our first classification models using the **K-Nearest Neighbors** algorithm. We'll start with theory, then dive into building KNN models using Python.

TOPICS WE'LL COVER:

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons

GOALS FOR THIS SECTION:

- Introduce the KNN algorithm and workflow, and build intuition for how the model works
- Learn how to assess model performance using accuracy metrics and the confusion matrix
- Find the optimal value of k through hyperparameter tuning and cross validation
- Compare "hard" vs. "soft" classification, and summarize general pros and cons for KNN

K-NEAREST NEIGHBORS (KNN)

K-Nearest Neighbors (KNN) is a classification technique designed to predict outcomes based on similar observed values

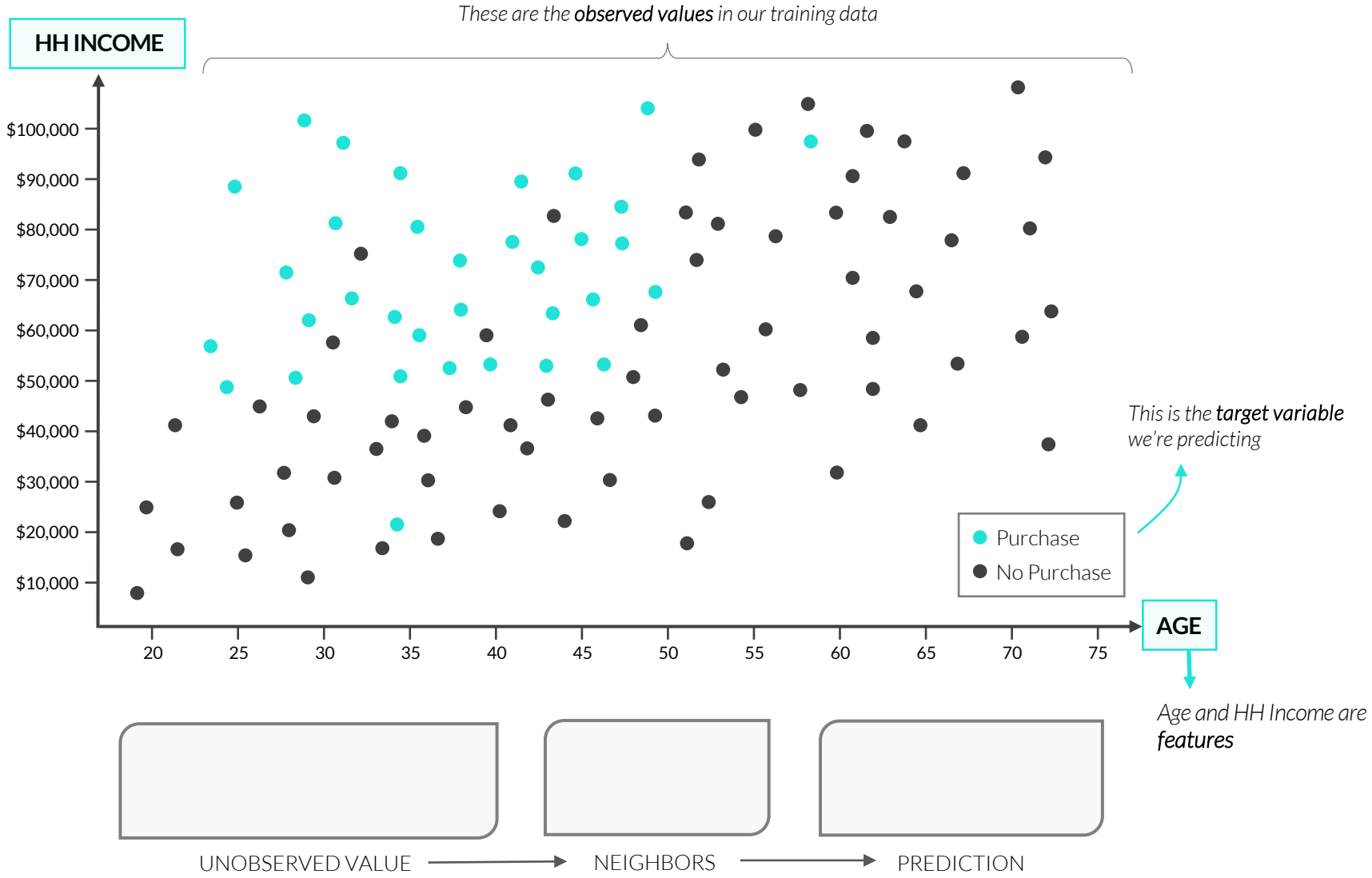
- In its simplest form, KNN creates a scatter plot with training data, plots a new unobserved value, and assigns a target class based on the classes of nearby points
- **K** represents the number of nearby points (or "neighbors") the model will consider when making a prediction

Example use cases:

- Predicting which customers will purchase a product based on demographics
- Classifying customers into pre-existing clusters or segments

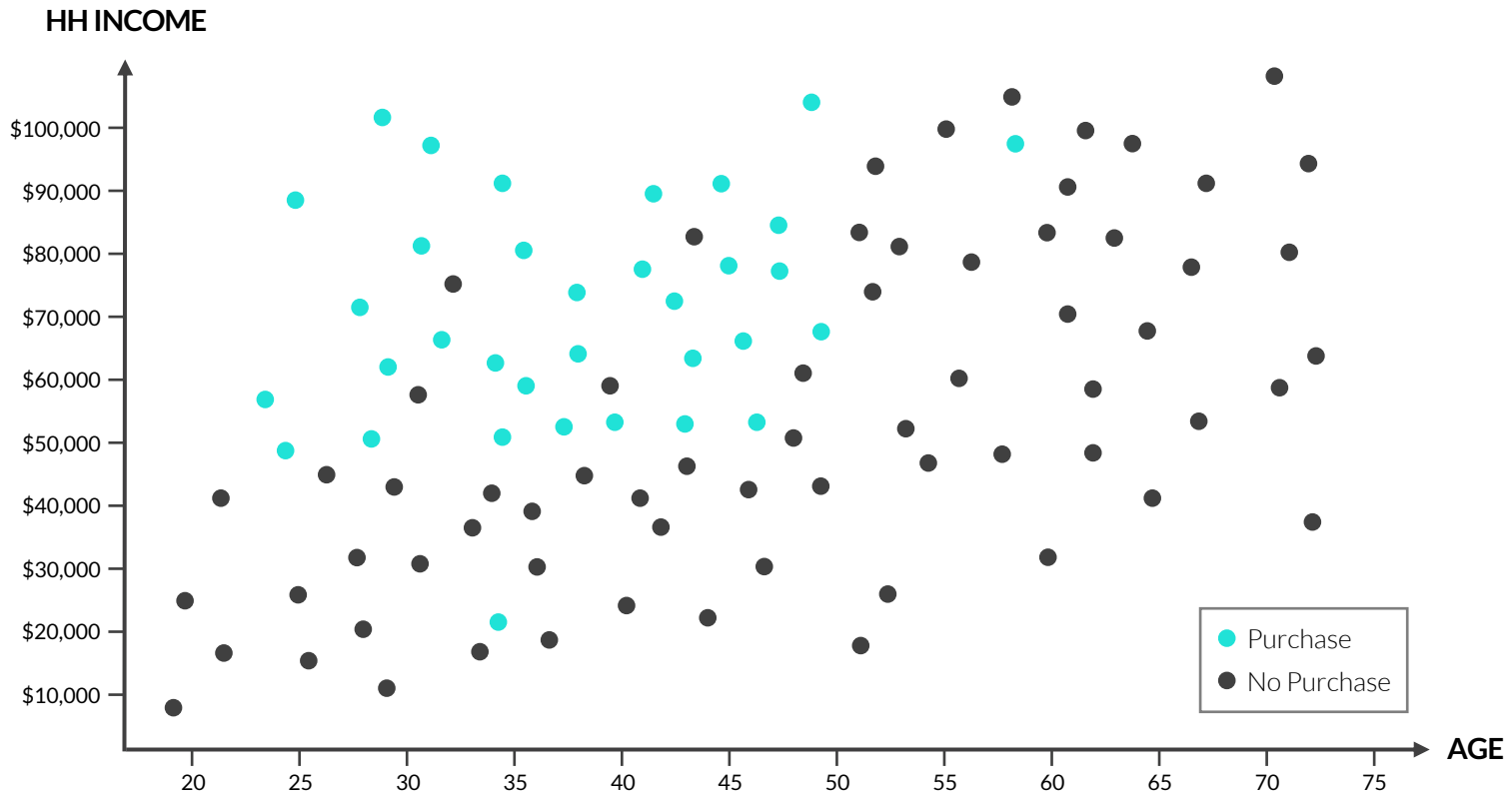
K-NEAREST NEIGHBORS (KNN)

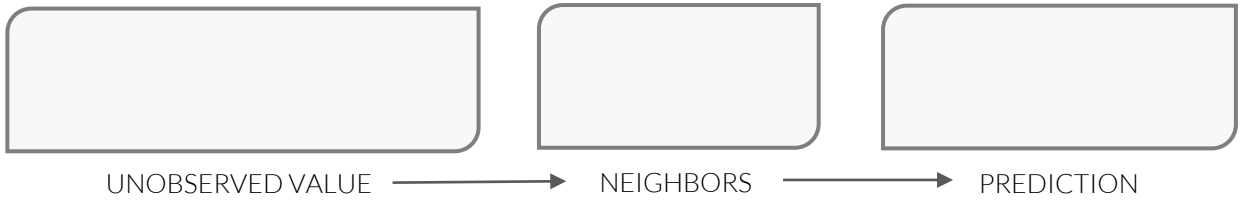
- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



K-NEAREST NEIGHBORS (KNN)

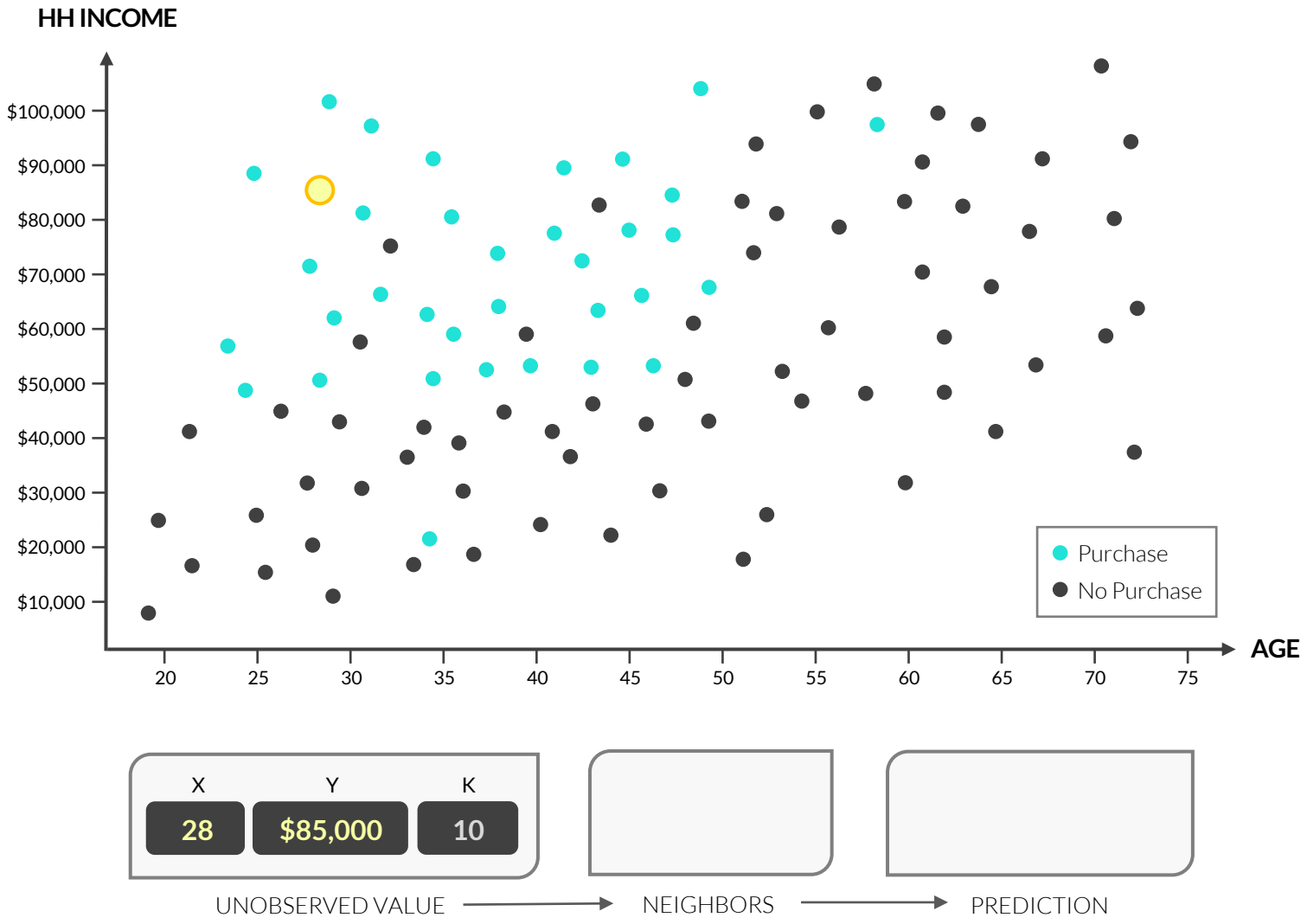
- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons





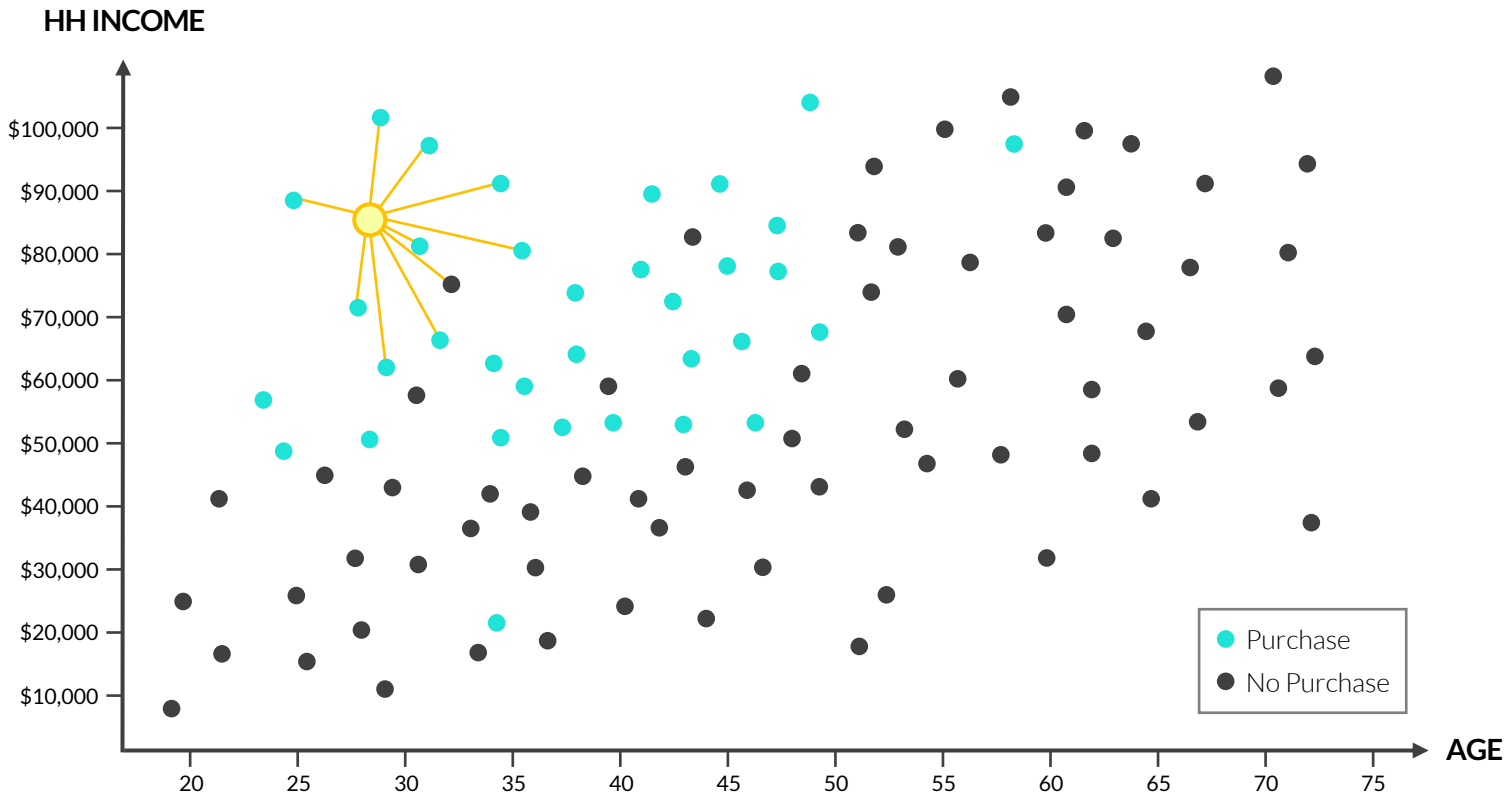
K-NEAREST NEIGHBORS (KNN)

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



K-NEAREST NEIGHBORS (KNN)

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



X	Y	K
28	$85,000	10

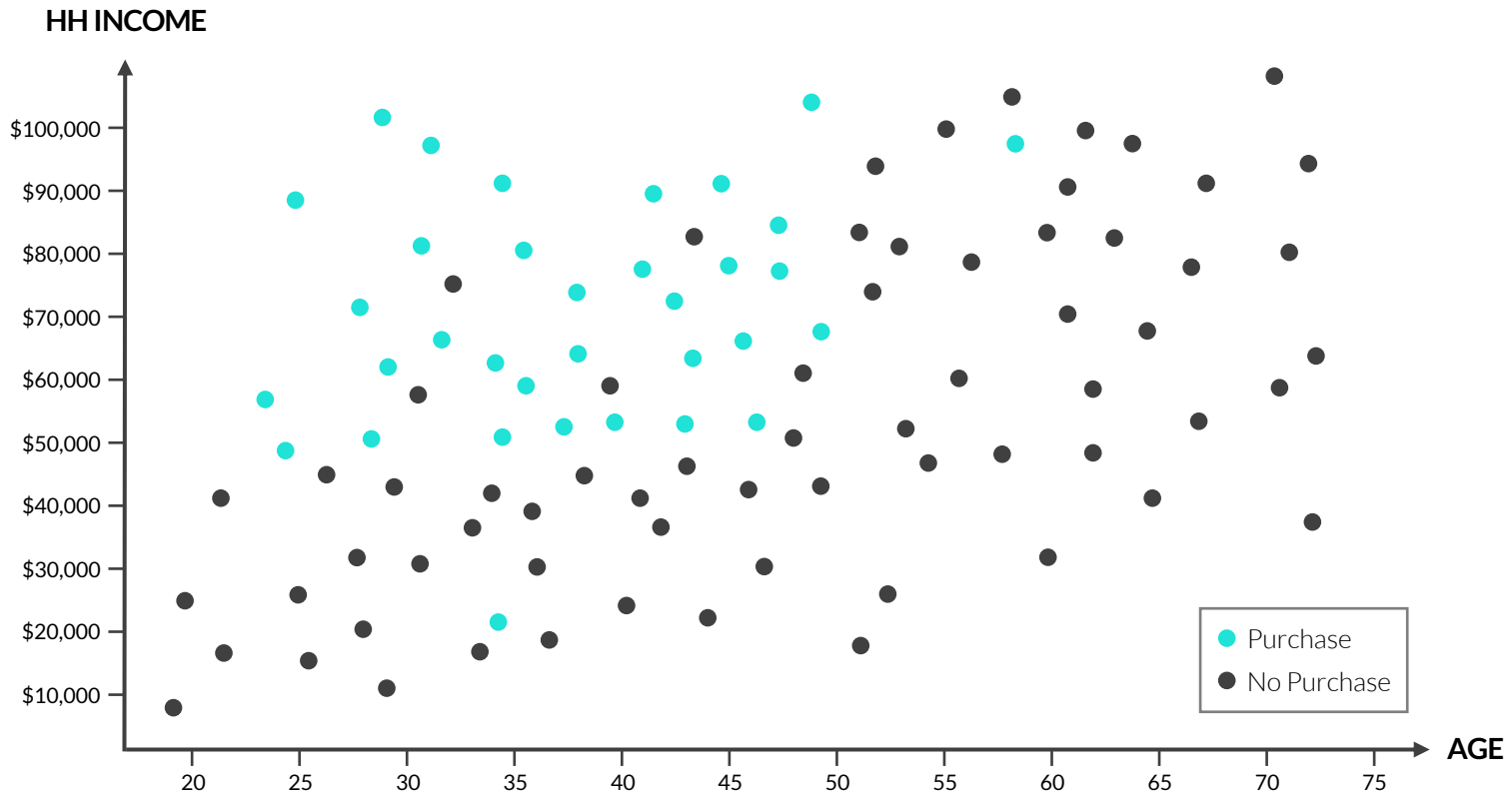
- 9 Purchase
- 1 No Purchase

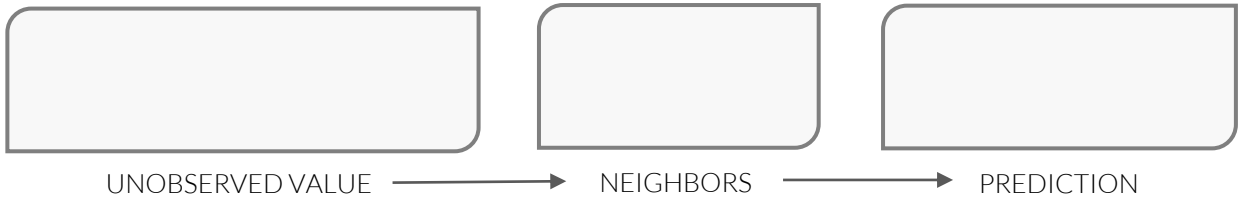
PURCHASE
(*90% Confidence*)

UNOBSERVED VALUE → NEIGHBORS → PREDICTION

K-NEAREST NEIGHBORS (KNN)

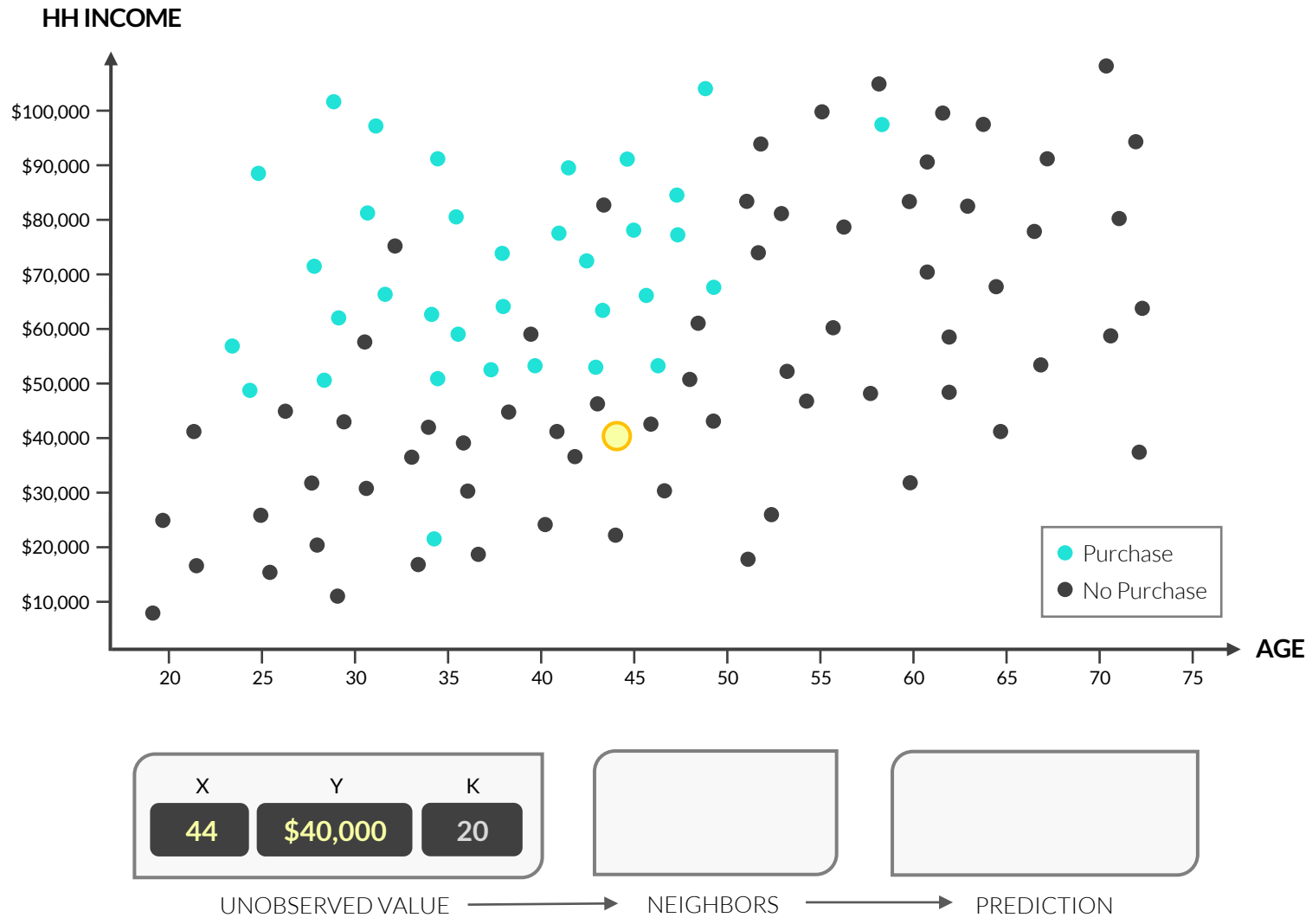
- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons





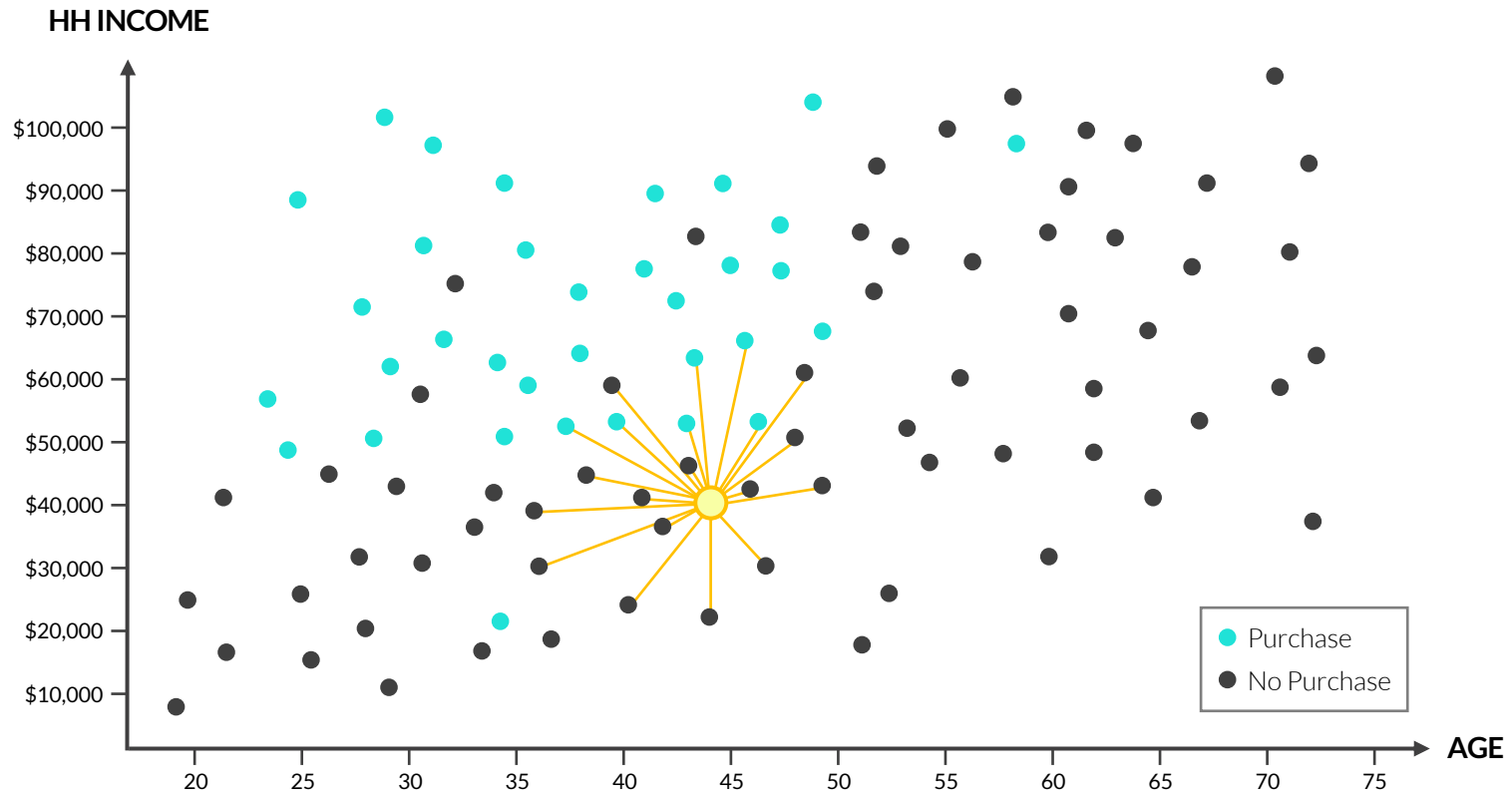
K-NEAREST NEIGHBORS (KNN)

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



K-NEAREST NEIGHBORS (KNN)

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



X	Y	K
44	$40,000	20

UNOBSERVED VALUE →

- 6 Purchase
- 14 No Purchase

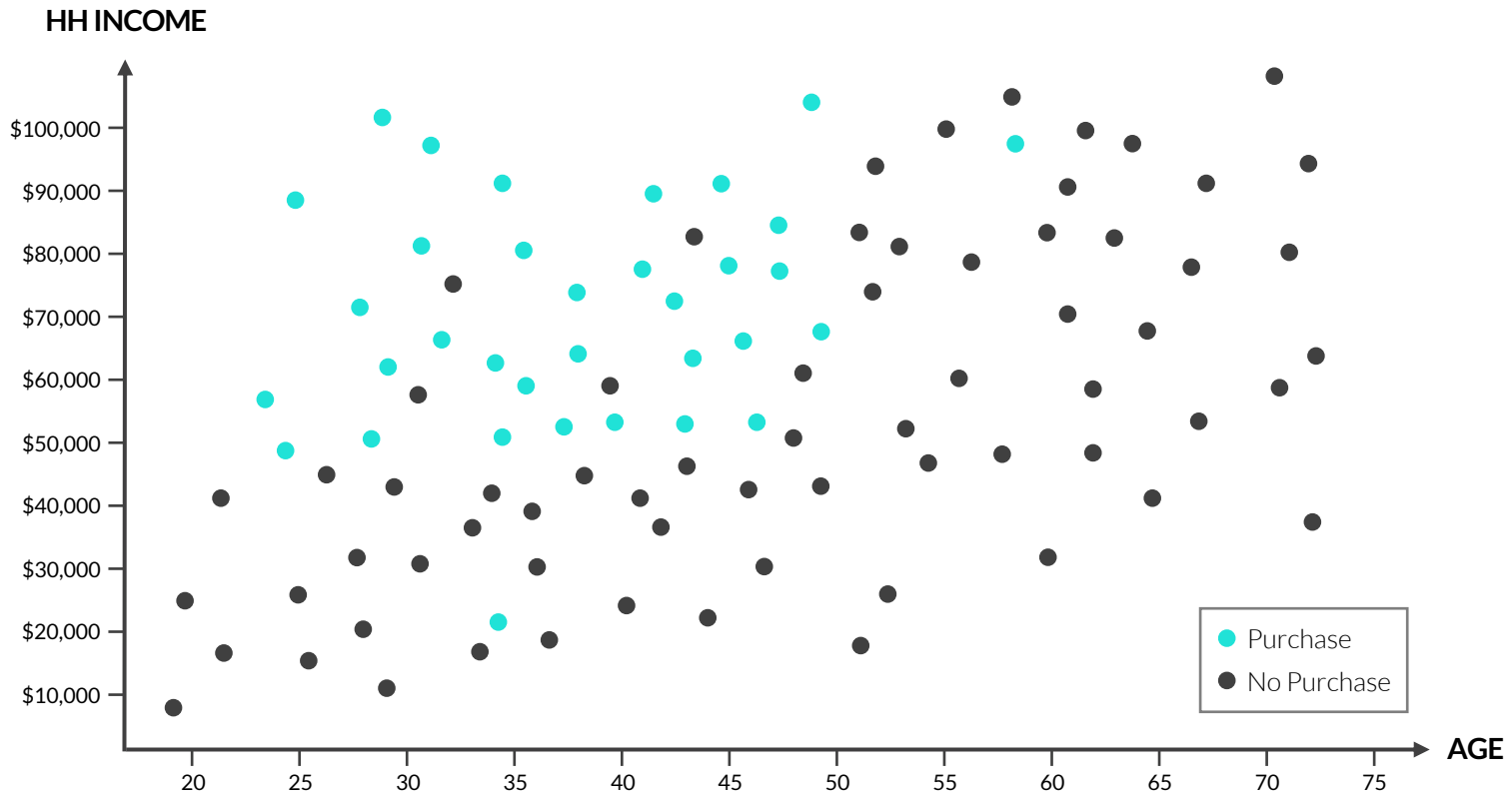
NEIGHBORS →

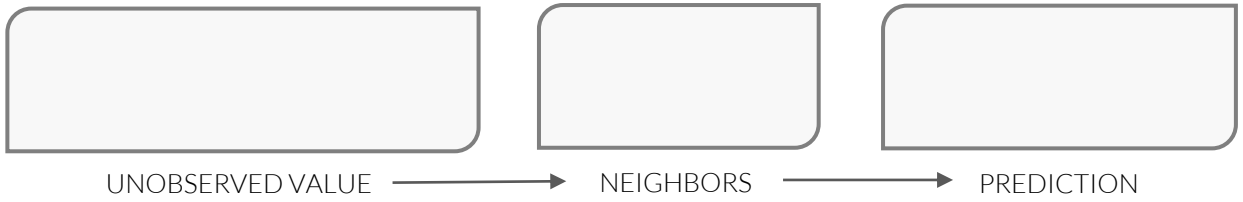
NO PURCHASE
(*70% Confidence*)

PREDICTION

K-NEAREST NEIGHBORS (KNN)

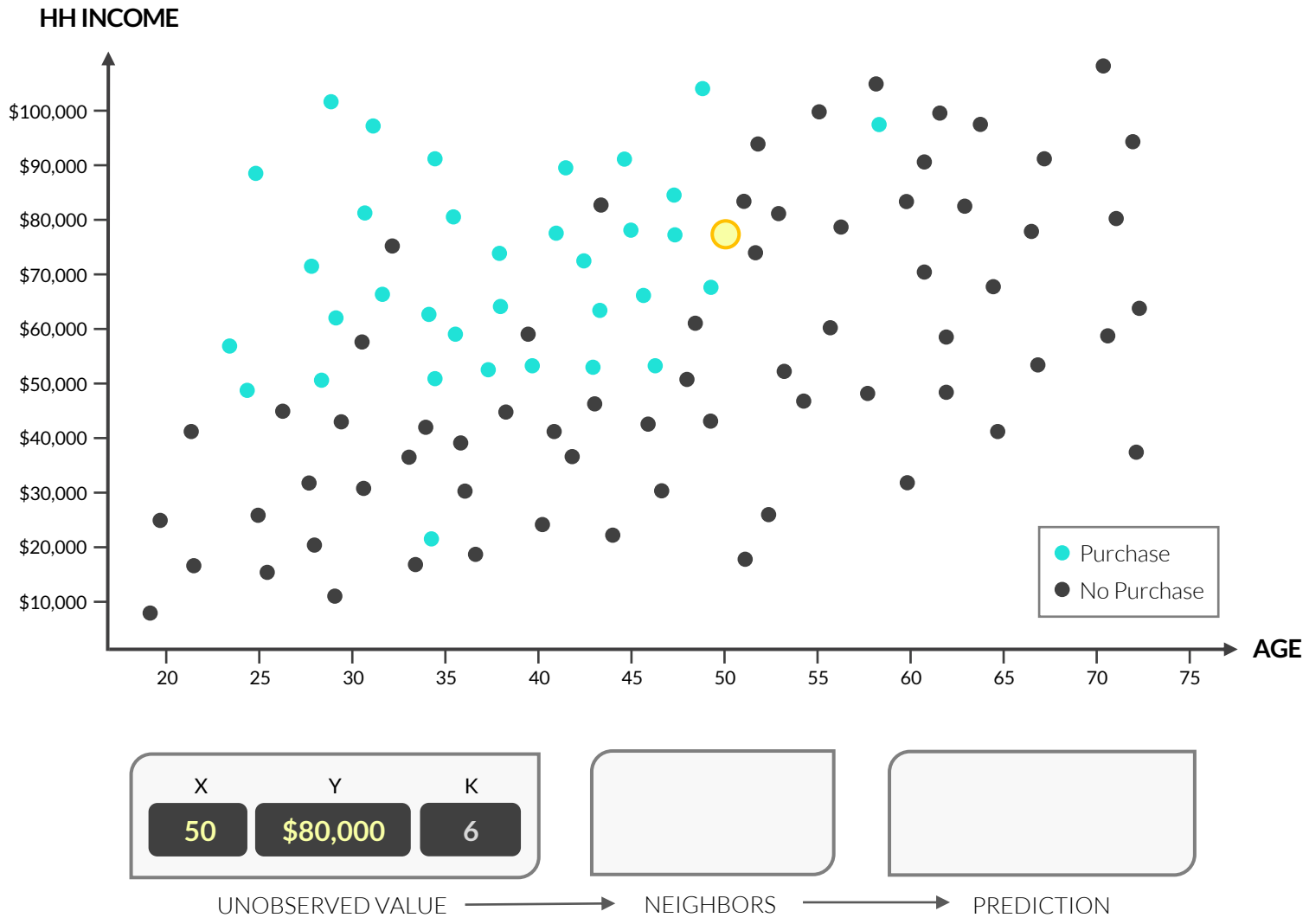
- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons





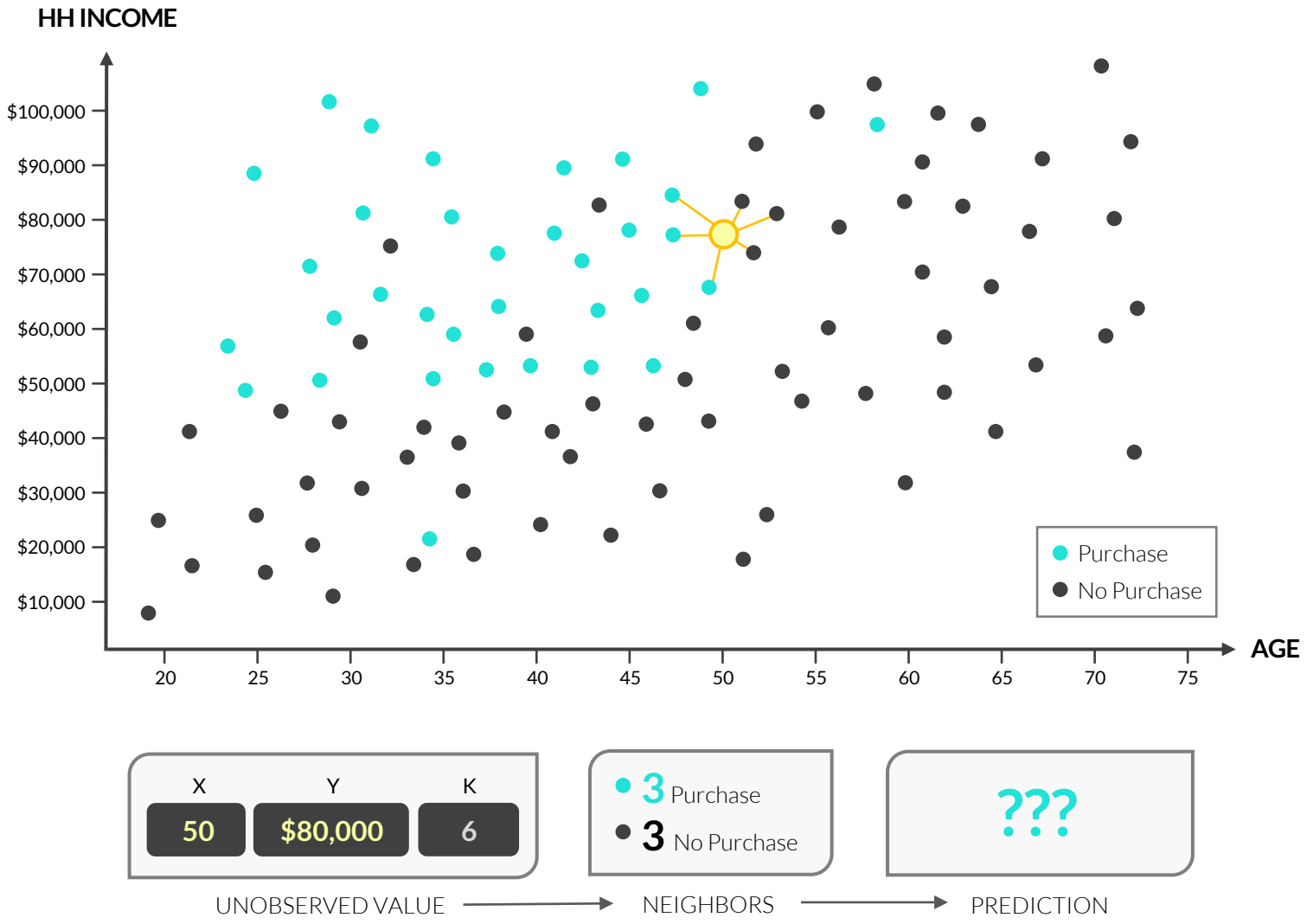
K-NEAREST NEIGHBORS (KNN)

- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



K-NEAREST NEIGHBORS (KNN)

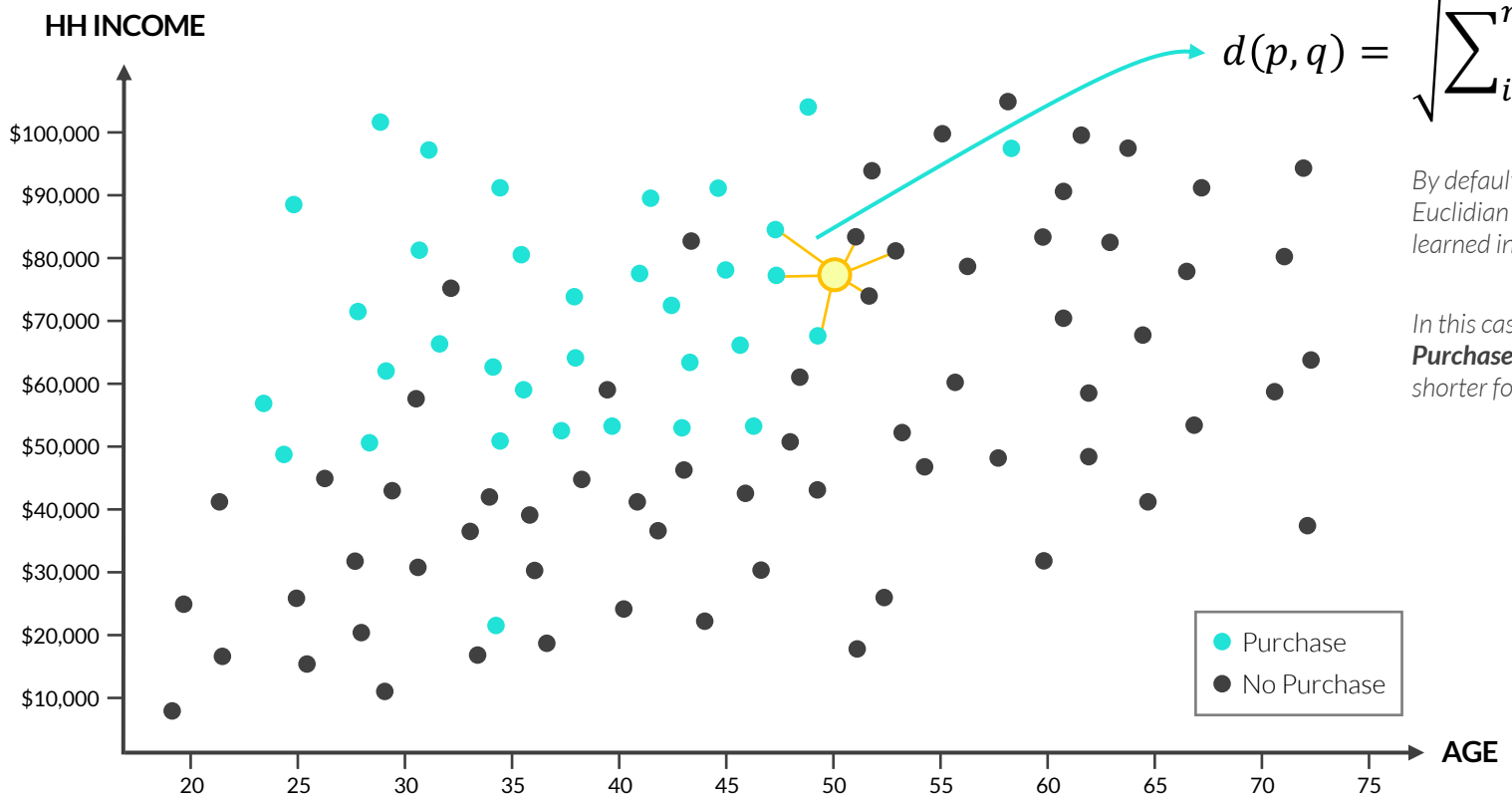
- KNN Overview
- KNN Workflow
- Model Accuracy
- Confusion Matrix
- Hyperparameter Tuning
- Cross Validation
- Hard vs. Soft Classification
- Pros & Cons



K-NEAREST NEIGHBORS (KNN)

What if there's a tie?

- In practice, KNN also factors *distance* between neighbors; if there are multiple modes, the class with the **shortest total distance** to the observation wins



$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

By default, the distance formula is Euclidian Distance, the same formula you learned in algebra!

*In this case the prediction would be **No Purchase** since the total distance is shorter for observations in that class*

K-NEAREST NEIGHBORS (KNN)

What if there's a tie?

- In practice, KNN also factors *distance* between neighbors; if there are multiple modes, the class with the **shortest total distance** to the observation wins

How do we figure out the "right" value for K?

- This is where **model validation** comes in! We will train our model on multiple K values to see which one is most accurate when applied to your validation data

What if we have more than 2 features?

- Scatter plots are great for demonstrating how KNN works for 2 features, but humans can't visualize beyond 3 or 4 dimensions
- ML algorithms can easily apply the same underlying logic and calculations to *many* features.

KNN WORKFLOW

Follow the **KNN workflow** below to ensure that you fit an accurate model:

1. **Split** training and test datasets to accurately assess model performance
2. **Scale** or standardize the data to make sure all features are weighted equally
3. **Fit & Tune** the model by adding or removing features and adjusting k
4. **Score** the model on the test data after fitting your final model on the combined training & validation sets

DATA SPLITTING

Before fitting any model, it's important to **split off a test data set**

- sklearn.model_selection.train_test_split(X, y, test_size, random_state)

Split the Data:

```
from sklearn.model_selection import train_test_split

X = pd.get_dummies(ads[["Gender", "Age", "EstimatedSalary"]], drop_first=True)
y = ads["Purchased"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2023)
```

DATA SCALING

Scaling or standardizing your data is necessary for kNN to ensure that features have an equal unit scale (**this is critical for distance-based models!**)

- sklearn.preprocessing.StandardScaler()

Scale the Data:

```
from sklearn.preprocessing import StandardScaler

std = StandardScaler()

X_train_std = std.fit_transform(X_train)
X_test_std = std.transform(X_test)
```

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

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

FITTING A KNN MODEL

In Python, you can **fit a kNN model** using the **scikit-learn** library

- Remember, data must be split and scaled prior to modeling!
- sklearn.neighbors.KNeighborsClassifier(n_neighbors=k).fit(X, y)

```
from sklearn.neighbors import KNeighborsClassifier

k = 5
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train_std, y_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

While we're manually specifying k here, in practice **you'll use cross-validation to determine the optimal value of k**

MODEL ACCURACY

Accuracy represents the number of observations predicted correctly divided the total number of observations

- Fitted sklearn models have a **.score()** method (model.score(X, y) which returns the accuracy score for any feature/target pair

Actual	Predicted	Correct?
1	1	Yes
0	1	No
0	0	Yes
1	0	No
1	1	Yes

Correct: **3**

Rows: **5**

Accuracy

60%

```
knn.score(X_train_std, y_train)
```

```
0.921875
```

```
knn.score(X_test_std, y_test)
```

```
0.8875
```

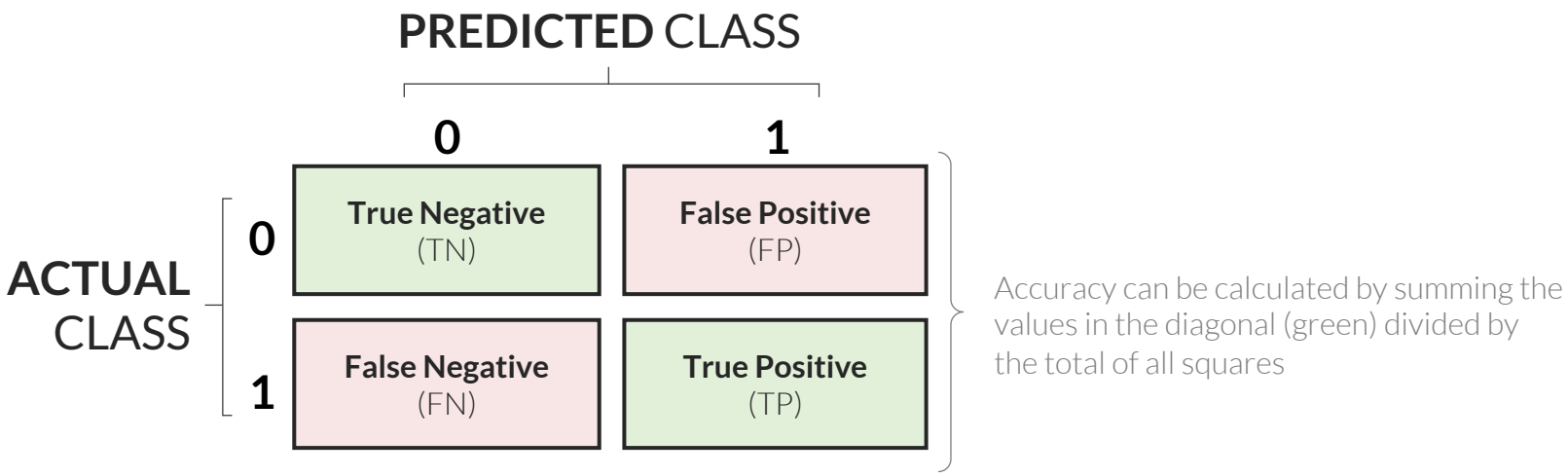
Our model performed a little bit worse on our test data, but this gap is reasonable given our test data only had 80 rows

Accuracy isn't always a useful metric, depending on your project goals. We'll dive into cases where it falls short in the model evaluation section!

THE CONFUSION MATRIX

The **confusion matrix** summarizes predicted vs. actual classes, and can be used for summarizing model performance and diagnosing inaccurate predictions

- Metrics like accuracy, precision and recall (more later!) can all be derived from this matrix



THE CONFUSION MATRIX

The **confusion matrix** summarizes predicted vs. actual classes, and can be used for summarizing model performance and diagnosing inaccurate predictions

- model.predict(X) returns class predictions from a given feature dataset
- Sklearn.metrics.confusion_matrix(actual_y, predicted_y)

```
from sklearn.metrics import confusion_matrix

y_pred = knn.predict(X_train_std)

print(confusion_matrix(y_train, y_pred))
```

```
[[196  13]
 [ 12  99]]
```

Accuracy: (196 + 99)/ (196 + 99 +13 + 12) = ***.9218***

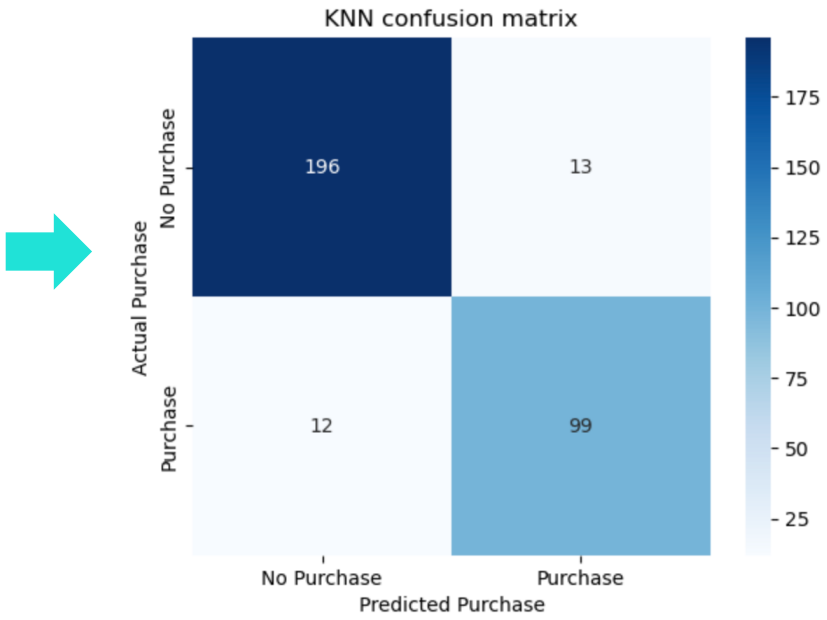
We had about the same number of false positives as false negatives, which means our model isn't significantly worse at predicting one class vs. the other

PRO TIP: CONFUSION MATRIX HEATMAP

Seaborn's **heatmap** function can be used to make a visual confusion matrix

- sns.heatmap(confusion_matrix(actual, predicted))

```
import seaborn as sns
from sklearn.metrics import confusion_matrix

knn_confusion = confusion_matrix(y_train, knn.predict(X_train_std))
sns.heatmap(
    knn_confusion,
    cmap="Blues",
    annot=True,
    fmt="g",
    square=True,
    xticklabels=["No Purchase", "Purchase"],
    yticklabels=["No Purchase", "Purchase"]
).set(
    xlabel='Actual Purchase',
    ylabel='Predicted Purchase',
    title='KNN confusion matrix'
);
```



ASSIGNMENT: FITTING A SIMPLE KNN MODEL

NEW MESSAGE

October 5, 2023

From: **Betty Binary** *(Data Scientist)*

Subject: **KNN**

Hi again,

We've gotten the green light to proceed with the modeling phase of the project.

Let's start simple but move quickly. Please fit a KNN model with k=5 using the features 'age' and 'hoursperweek'. Make sure to scale your data, and please generate a test accuracy score and confusion matrix.

Also, please build a scatterplot with x as 'age' and y as 'hoursperweek' with datapoints colored by their predicted values from your model.

02_KNN_Assignments.ipynb

Reply Forward

Key Objectives

1. Build a KNN Model
2. Use your model to generate an accuracy score and confusion matrix
3. Build a scatterplot with your features in the X and Y axes and the points colored by their predicted values

HYPERPARAMETER TUNING

Hyperparameters are settings that affect how well a model fits the data

- Unlike model parameters (like regression coefficients) these are not learned from the data
- Parameters are set before the model is fit and optimized based on validation performance

K-Nearest Neighbors

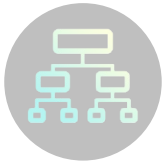
Examples:

- *K*
- *Distance metric*
- *weights*

Logistic Regression

Examples:

- *Regularization Strength*
- *Penalty Type*
- *Solver*



Decision Trees

Examples:

- *Criterion (entropy, gini)*
- *Tree depth*
- *Leaf size*

Random Forests

Examples:

- *# Trees*
- *# Samples*
- *# Features*
- *Re-Sampling*

Gradient Boosted Trees

Examples:

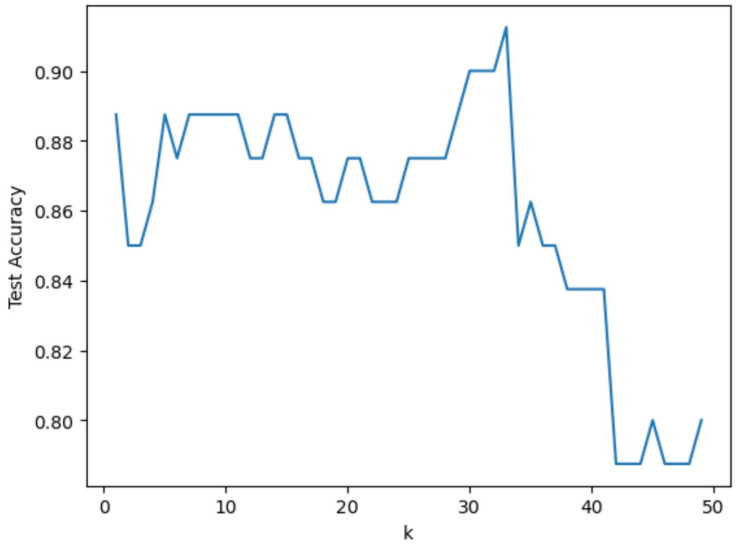
- *# Trees*
- *# Learning Rate*
- *# Sample Size*

OPTIMIZING K

The **optimal value of k** can be determined via model validation

- k is bounded between **1** and **n**, where n is the number of rows in a dataset
- High values of k will tend to underfit, and low values of k tend to overfit

```
accuracy_scores = []

for k in range (1, 50):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_std, y_train)
    accuracy = accuracy_score(knn.predict(X_test_std), y_test)
    accuracy_scores.append(accuracy)

(sns.lineplot(
    x=range(1, 50),
    y=accuracy_scores)
 .set(
     xlabel="k",
     ylabel="Test Accuracy")
);
```

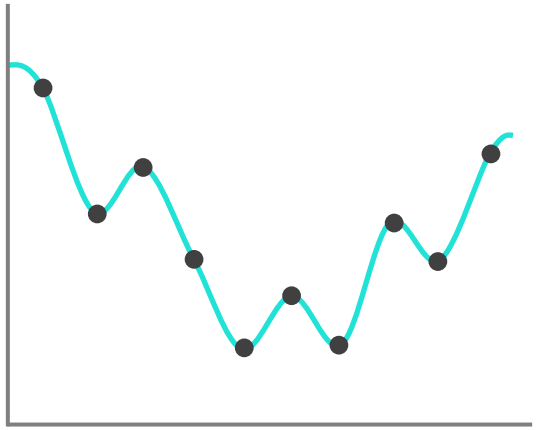


Out of sample accuracy increases as we increase k, as we reduce variance, but eventually k grows so large we underfit

OVERFITTING & UNDERFITTING

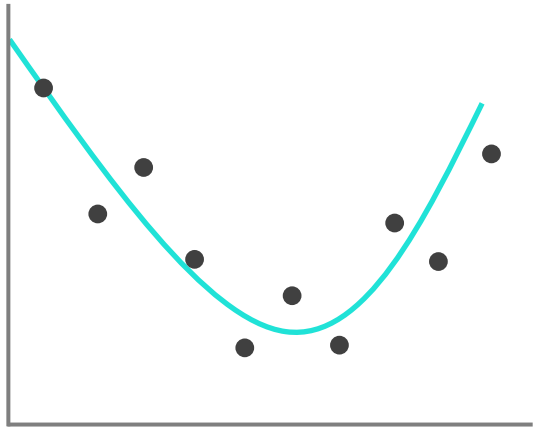
Data splitting is primarily used to avoid **overfitting**, which is when a model predicts known *(Training)* data very well but unknown *(Test)* data poorly

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



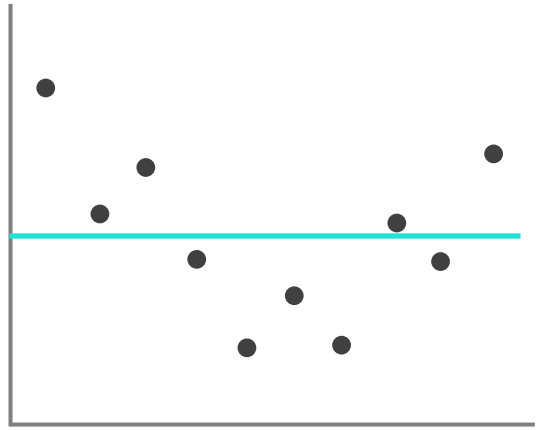
OVERFIT model

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



WELL-FIT model

- *Models the training data just right*
- *Generalizes well to test data*
- *Balances bias and variance*



UNDERFIT model

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

OVERFITTING & UNDERFITTING

You can diagnose overfit & underfit models by **comparing evaluation metrics** like Accuracy, Precision, and Recall between the train and test data sets

- Large gaps between train & test scores indicate that a model is **overfitting** the data
- Poor results across both train & 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
- Adding new features
- Feature engineering

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!

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 or outliers on out-of-sample data*

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
- This is the preferred method for tuning hyperparameters in scikit-learn

5-Fold Cross Validation

Fold 1	*Fold 2*	*Fold 3*	*Fold 4*	*Fold 5*	*Validation Score*
Training	Training	Training	Training	Validation	0.75
Training	Training	Training	Validation	Training	0.67
Training	Training	Validation	Training	Training	0.79
Training	Validation	Training	Training	Training	0.72
Validation	Training	Training	Training	Training	0.70

Cross validation score:

0.73

CROSS VALIDATION

Cross validation tends to yield the best out of sample performance

- GridSearchCV(model, parameters) can be used for tuning model hyperparameters

```
from sklearn.model_selection import GridSearchCV

parameters = {"n_neighbors": range(1, 50)}

gridsearch = GridSearchCV(KNeighborsClassifier(), parameters)

gridsearch.fit(X_train_std, y_train)

gridsearch.best_params_
```

Creates a parameter dictionary with a range of values to test

Sets up a cross validation loop to score all combinations of hyperparameters in our dictionary using the specified model

```
{'n_neighbors': 15}
```

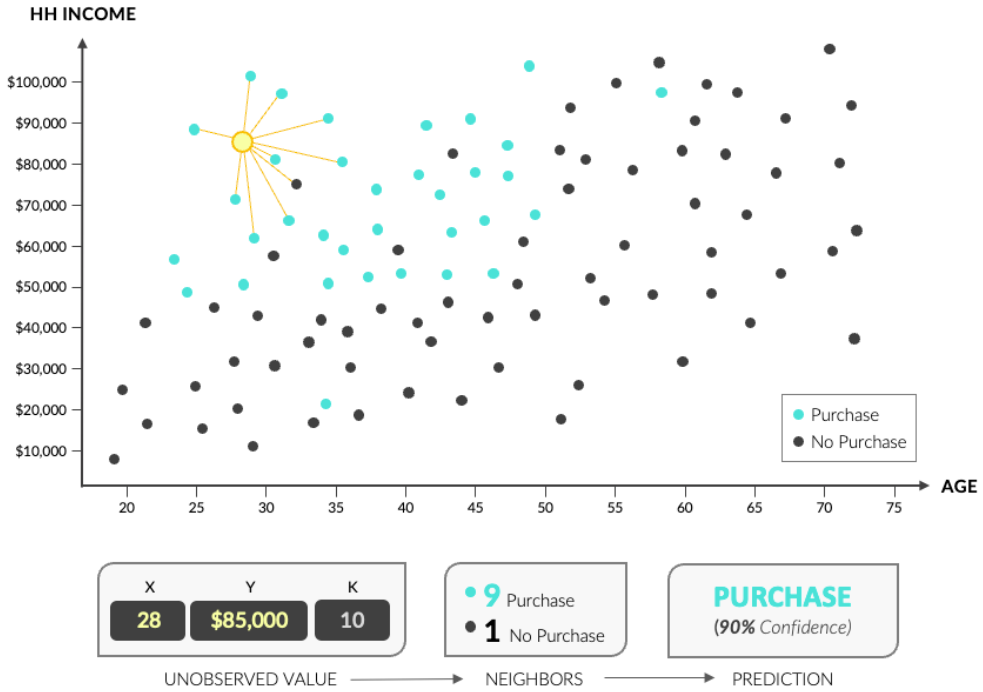
```
k = 15
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train_std, y_train)

accuracy_score(y_test, knn.predict(X_test_std))
```

```
0.8875
```

“HARD” VS. “SOFT” CLASSIFICATION

Classification models can produce both “**hard” and “soft” classification scores**

- “**Hard**” classification produces the *predicted* class (e.g. Class A or Class B)
- “**Soft**” classification produces *probabilities* of being a given class



By default, a probability of .5 is the cutoff for hard classification. Later in the course, we will look at how **you can change the hard classification threshold to better achieve your modeling goals.**

Hard Classification:

Because 9/10 neighbors purchased, our model predicts this customer **will purchase**

Soft Classification:

Because 9/10 neighbors purchased, our model predicts this has a **90% chance to purchase**

"HARD" VS. "SOFT" CLASSIFICATION

Classification models can produce both "**hard" and "soft" classification scores**

- model.predict(data) produces hard classifications for a feature dataset
- model.predict_proba(data) produces soft classification scores

Hard Classification:

```
knn.predict(X_test_std)[:5]
```

```
array([1, 1, 0, 0, 1])
```

*The model predicts rows **1**,**2**, and **5** in our test data set will purchase our products*

Soft Classification:

```
knn.predict_proba(X_test_std)[:5]
```

```
array([[0.06666667, 0.93333333],
       [0.46666667, 0.53333333],
       [0.73333333, 0.26666667],
       [1.        , 0.        ],
       [0.06666667, 0.93333333]])
```

Class 0 (Not Purchased) ← → **Class 1** (Purchased)

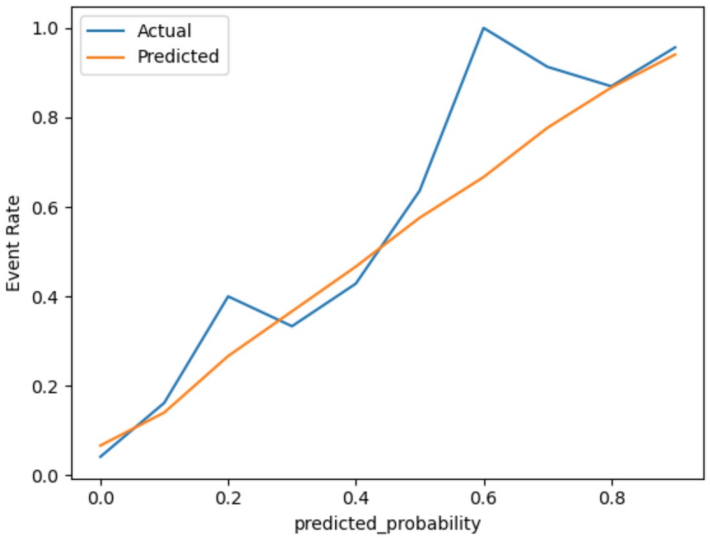
*Looking at soft probabilities, our model is much more confident that rows 1 and 5 will purchase, with a probability of **.93**, than row 3, which only has a purchase probability of **.53***

PROBABILITY VS. EVENT RATE

Plotting **predicted probabilities vs. observed event rates** give us a visual estimation of "soft" classification performance

- We plot the predicted probability vs. average the positive class occurs in each bin
- Ideally our predicted probabilities align closely with our event rate

```python
import numpy as np

bins = np.linspace(0, 1, 11)

# Create dataframe with Actual Classes & Predicted Probabilities
actual_vs_prob = (
    pd.DataFrame({
        "Actual": y_train,
        "Predicted": knn.predict_proba(X_train_std)[:, 1]})
)

# Create bins at increments of .1
actual_vs_prob["predicted_probability"] = (
    pd.cut(actual_vs_prob["Predicted"],
    bins=bins,
    labels = bins[:-1].round(2))
)

# Group by probability bins and calculate actual even rate & plot
(actual_vs_prob
 .groupby("predicted_probability")
 .agg({"Actual": "mean", "Predicted": "mean"})
 .plot
 .line(ylabel="Event Rate")
)
```



It isn't perfect, but overall, our event rate rises with predicted probability

ASSIGNMENT: TUNING A KNN MODEL

NEW MESSAGE

October 5, 2023

From: **Betty Binary** *(Data Scientist)*

Subject: **Optimized KNN**

Hi again,

The prototype model looks promising, but now it's time to think bigger. Please include all available features in your model, and use cross validation to tune the value of k.

Please report back with the test accuracy and generate a confusion matrix for your test dataset.

Thanks!

02_KNN_Assignments.ipynb

Reply Forward

Key Objectives

1. Tune the k hyperparameter in KNN
2. Report test accuracy and generate a confusion matrix for your test dataset

PROS & CONS OF KNN

KNN has some pros and cons that data scientists should be aware of when deciding whether it is the best model for the task at hand

- In general, KNN is a great choice for smaller datasets, but faces severe performance drawbacks when working with larger, more complex data

Simplicity

KNN only has one hyperparameter (k) to tune. It's also easy to understand and implement, making it great for beginners.

Great for Small Datasets

KNN doesn't require a lot of training data to build a (relatively) stable model

Interpretability

We can view the "nearest neighbors" of a prediction to understand why the model made specific decisions

Computational Costs

When scoring new observations, the distance between that point must be calculated for all points in the data to classify it. This can take a long time with large datasets!

Memory Usage

To make predictions, the entire dataset must be stored in memory, which adds a lot of overhead in production

The Curse of Dimensionality

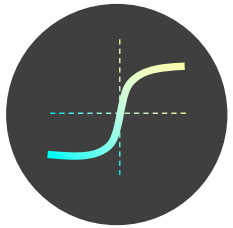
Distance metrics break down when working with large numbers of features (or categories). Multidimensional spaces become sparse, and distance is less meaningful.

KEY TAKEAWAYS

- **KNN** is a distance-based algorithm that classifies points based on the classes of nearby data points
  - *Euclidean distance is used to calculate the distance between points*
- The **hyperparameter "k"** specifies how many neighbors get a vote
  - *The value of k can be tuned via cross-validation techniques*
- **Accuracy** measures how many points were classified correctly
  - *The confusion matrix gives us more detail about what our model got right, and where it misclassified data*
- KNN (like all models) has **pros and cons** that should be considered
  - *KNN is great for prototypes and small datasets, but struggles with larger, more complex data*

LOGISTIC REGRESSION

LOGISTIC REGRESSION



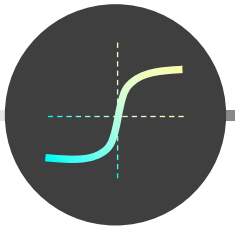
In this section we'll learn **logistic regression** and build models using Python. We'll also explore multi-class classification algorithms, model tuning and regularization, and more.

TOPICS WE'LL COVER:

- Model Overview
- Logistic Equation
- Likelihood
- Multiple Logistic Regression
- Fitting & Scoring
- Regularization & Tuning
- Multi-Class Models
- Pros & Cons

GOALS FOR THIS SECTION:

- Build an intuition for how logistic equations and classification models work
- Learn how to fit, score, and interpret binary and multi-class models
- Apply feature engineering techniques to improve model accuracy
- Fit and tune regularized logistic models using hyperparameters like strength and penalty type
- Explore the pros and cons of logistic regression



LOGISTIC REGRESSION

Logistic Regression is a classification technique used to predict the probability of a binary (true/false) outcome

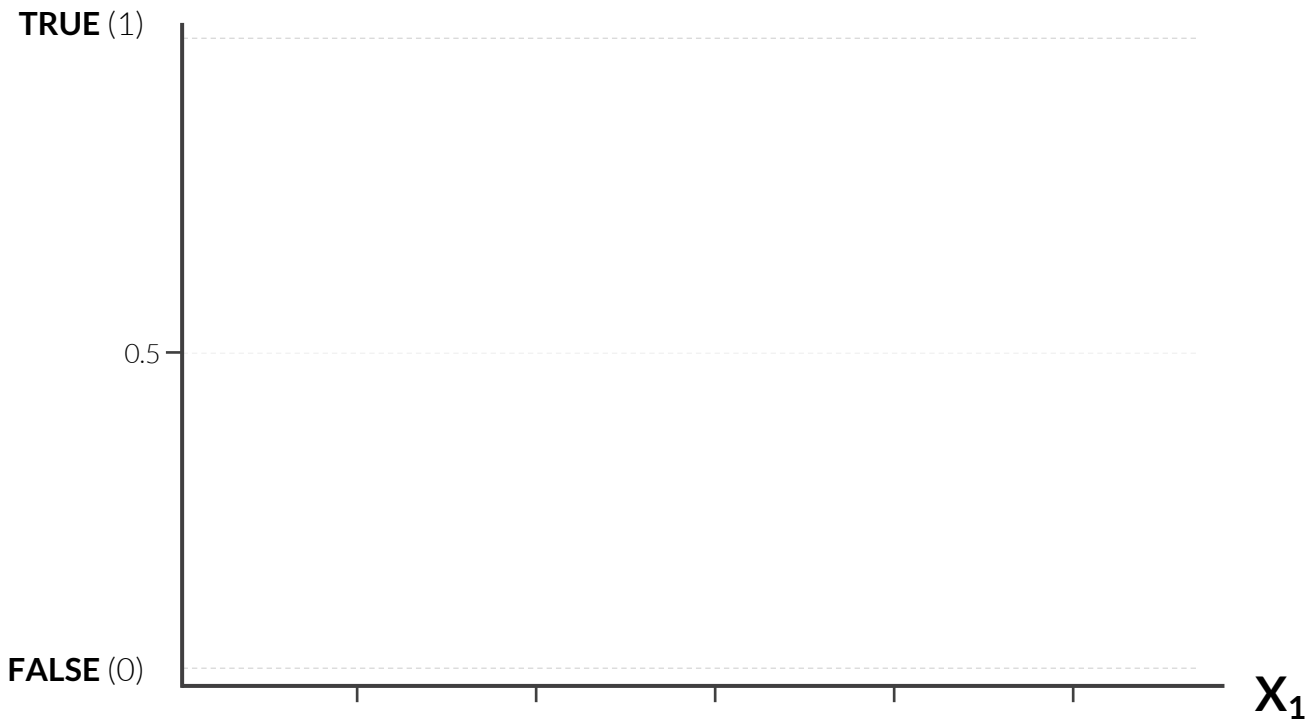
- In its simplest form, logistic regression forms an **S-shaped (sigmoid) curve between 0 and 1**, which represents the probability of a TRUE outcome for any given value of X
- The **likelihood function** measures how accurately a model predicts outcomes, and is used to optimize the "shape" of the curve
- Although it has the word "regression" in its name, logistic regression is not used for predicting numeric variables

Example use cases:

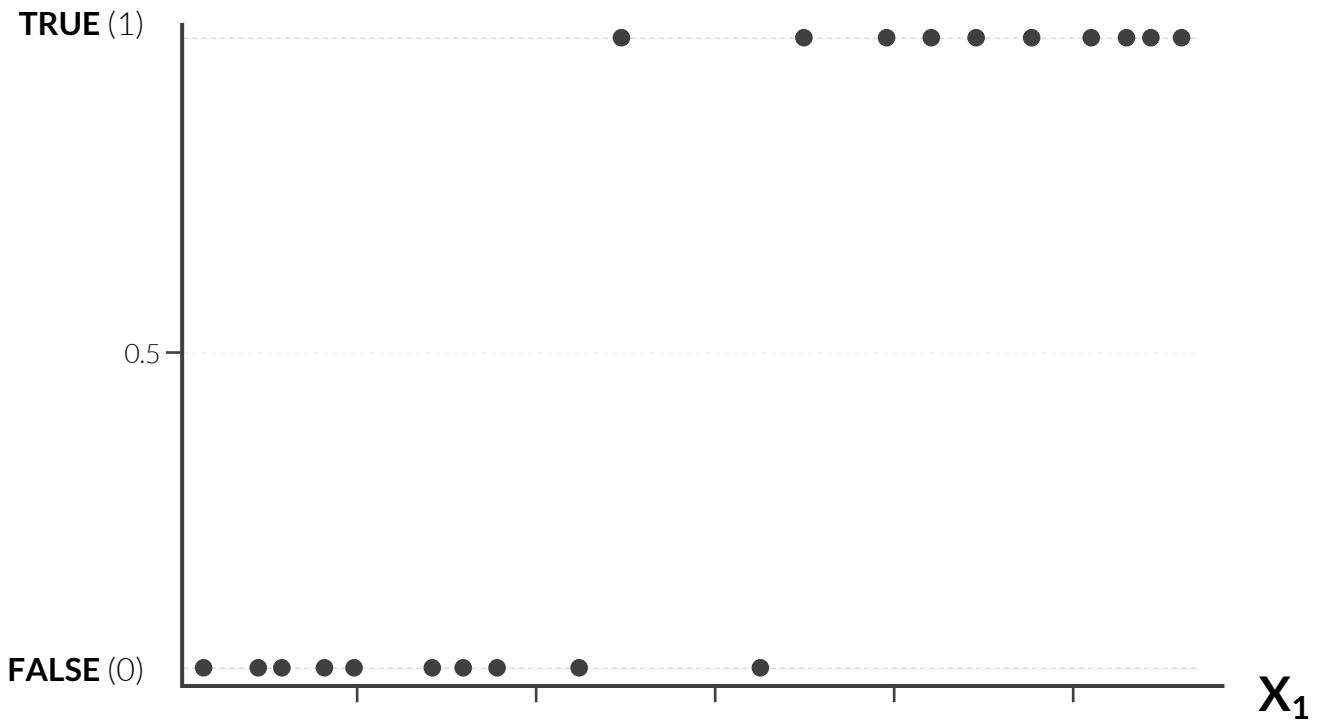
- Flagging spam emails or fraudulent credit card transactions
- Determining whether to serve a particular ad to a website visitor

LOGISTIC REGRESSION

- Model Overview
- Logistic Equation
- Likelihood
- Multiple Logistic Regression
- Fitting & Scoring
- Regularization & Tuning
- Multi-Class Models
- Pros & Cons

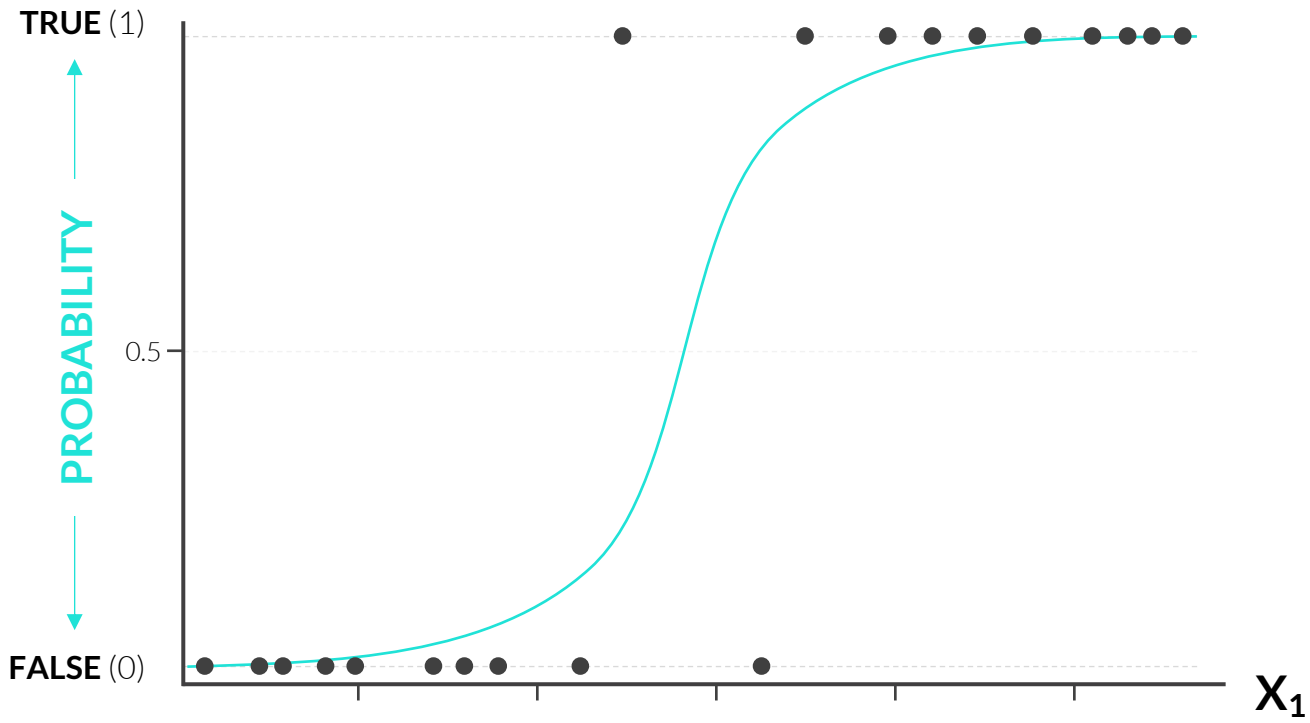


LOGISTIC REGRESSION

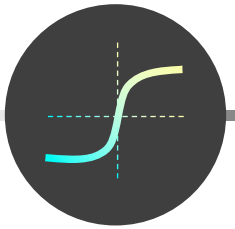


- Each dot represents an observed value, where **X is a numerical feature** and **Y is the binary outcome** *(true/false)* that we want to predict

LOGISTIC REGRESSION



- Logistic regression plots the **best-fitting curve between 0 and 1**, which tells us the probability of Y being TRUE for any given value of X_1



LOGISTIC VS LINEAR REGRESSION

Logistic regression and linear regression are both linear models, but **logistic regression outputs probability**, which is bounded by 0 and 1

- The logit transformation helps convert linear model outputs to probabilities

Name	Equation	Limits
Linear Equation	$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$	$(-\infty, +\infty)$
Probability	p	$[0, 1]$
Odds	$\frac{p}{1-p}$	$[0, +\infty)$
Log Odds (Logit)	$log(\frac{p}{1-p})$	$(-\infty, +\infty)$

Solving for p gives us the equation for the logistic regression line!

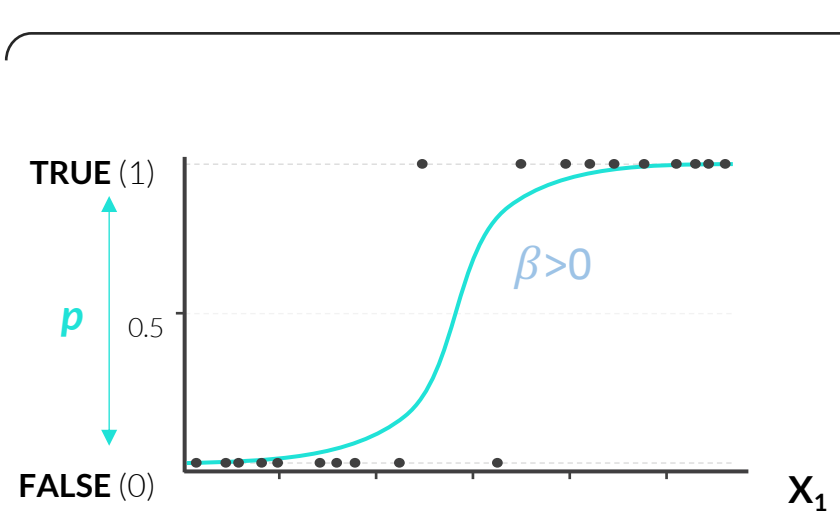
Log Odds (The Logit Function) **links probability to a linear combination of features** which has an output between 0 and 1

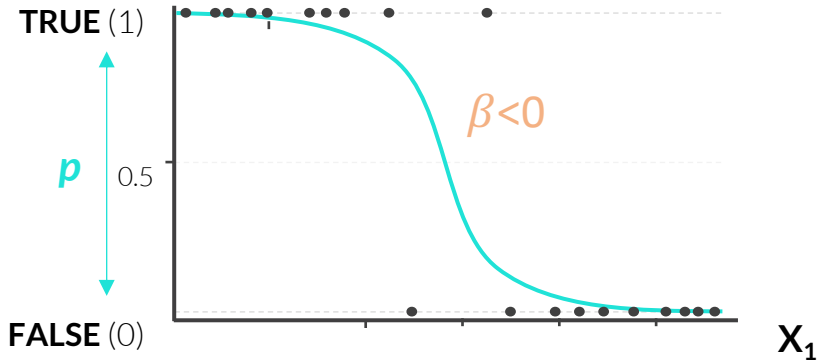
LOGISTIC EQUATION

The **logistic equation** solves for probability by setting log odds to a linear combination of features and solving for p

- $p = \frac{1}{1+e^{-\vec{x}}}$ = where $\vec{x} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$
- This is also known as a **sigmoid function**

$$p = \frac{1}{1+e^{-\beta x}}$$



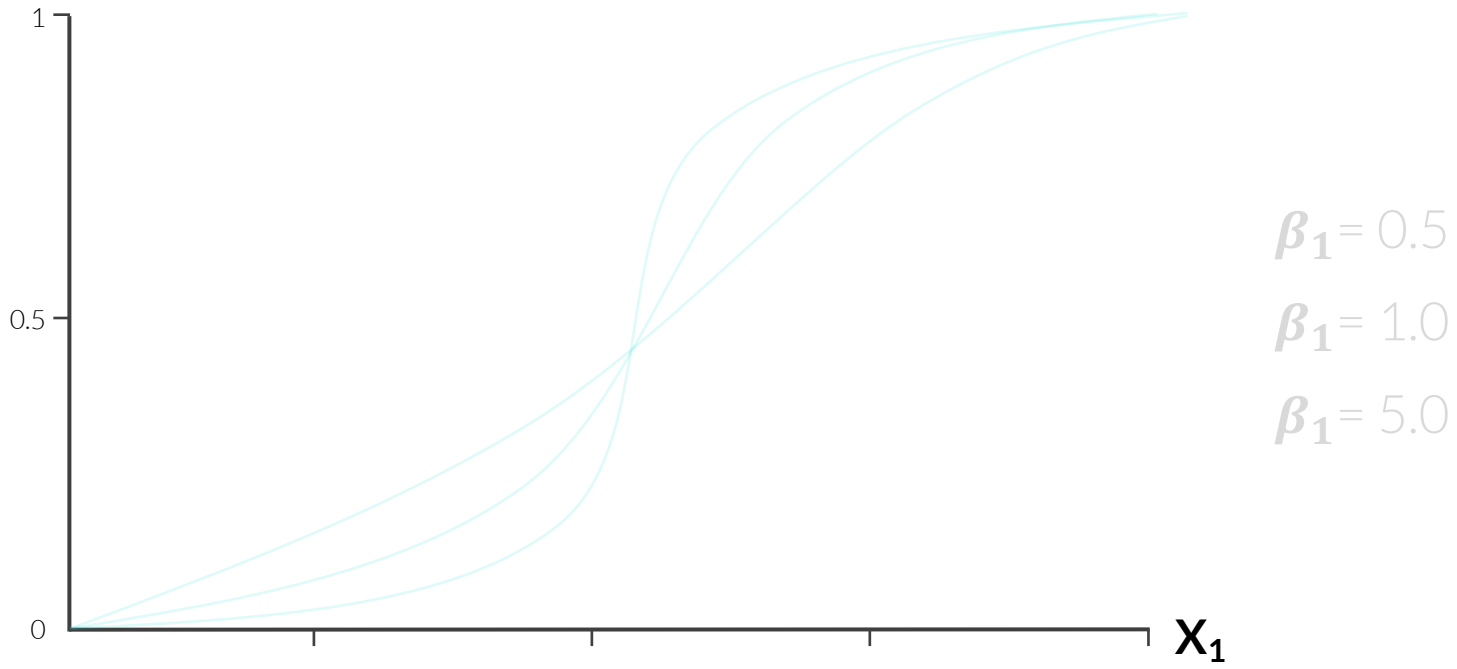


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where $\boldsymbol{\beta_0}$ is the intercept, **X** is the feature value and $\boldsymbol{\beta_1}$ is the weight *(or slope)*

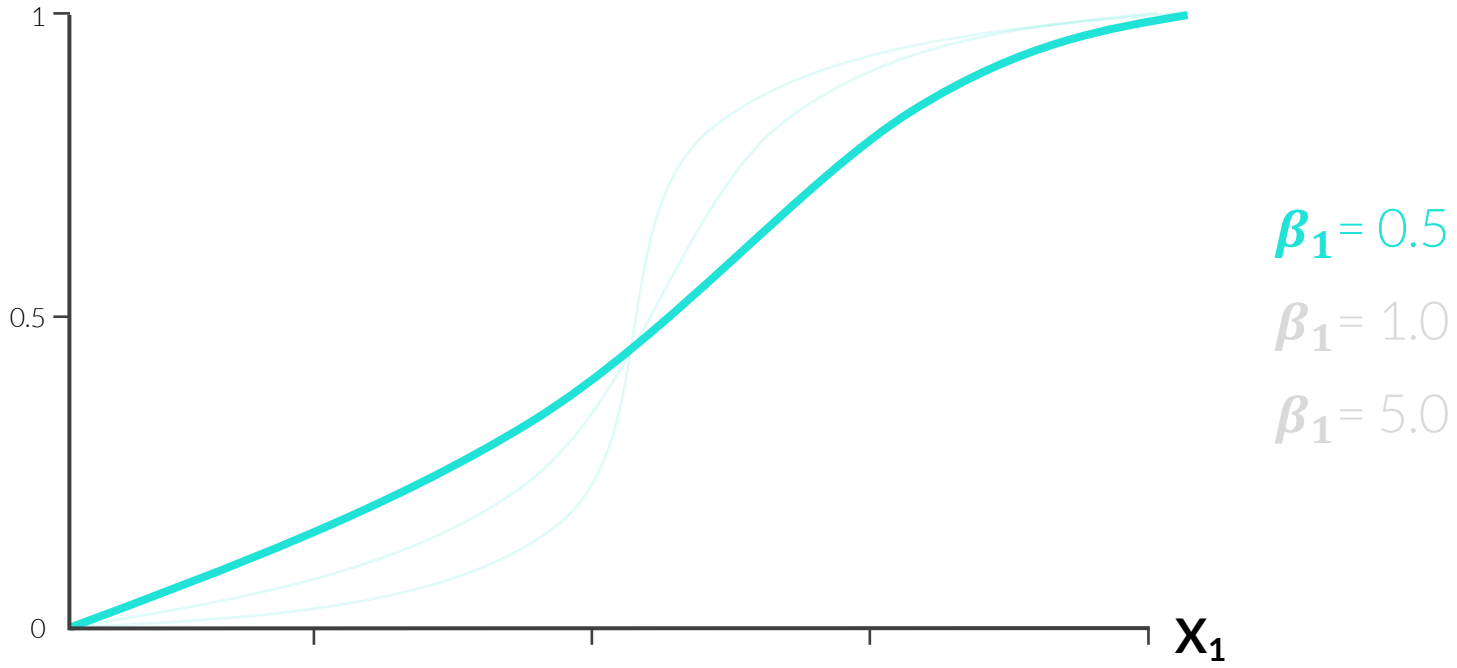


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where $\boldsymbol{\beta_0}$ is the intercept, **X** is the feature value and $\boldsymbol{\beta_1}$ is the weight *(or slope)*

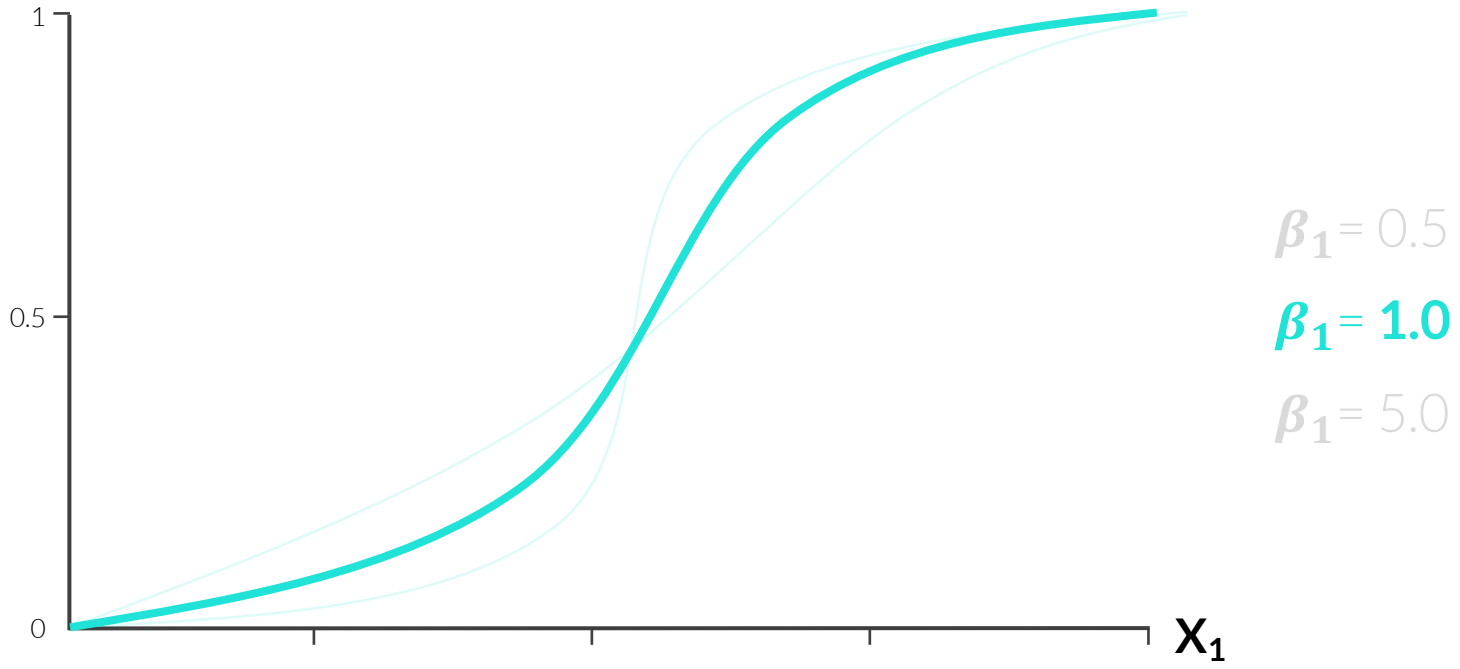


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where β_0 is the intercept, **X** is the feature value and β_1 is the weight *(or slope)*

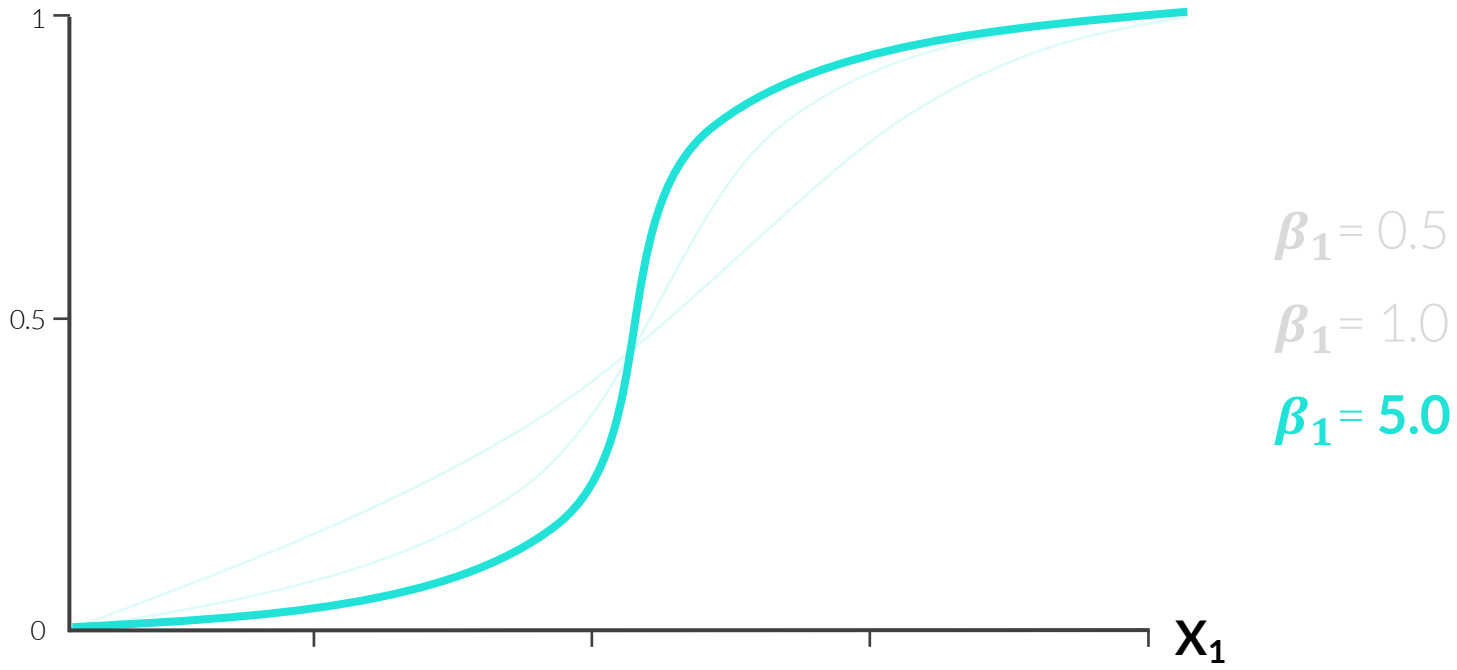


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where β_0 is the intercept, **X** is the feature value and β_1 is the weight *(or slope)*

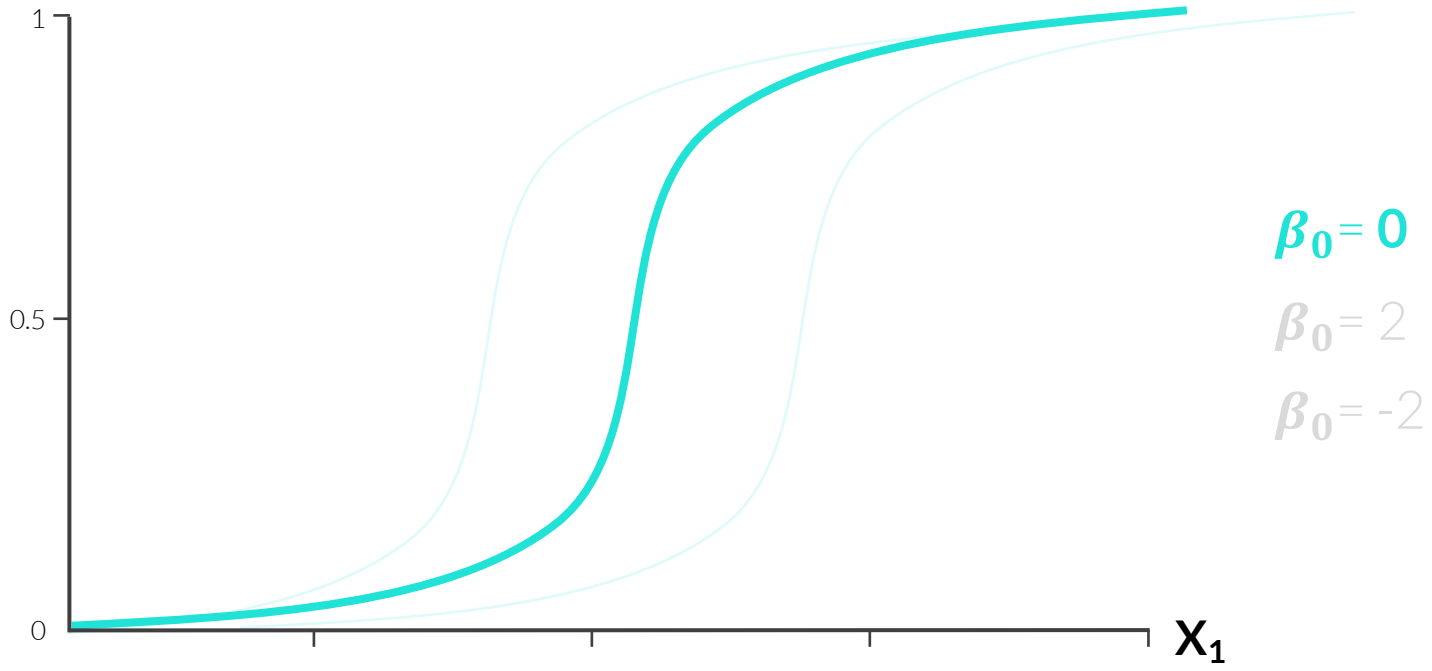


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where β_0 is the intercept, **X** is the feature value and β_1 is the weight *(or slope)*

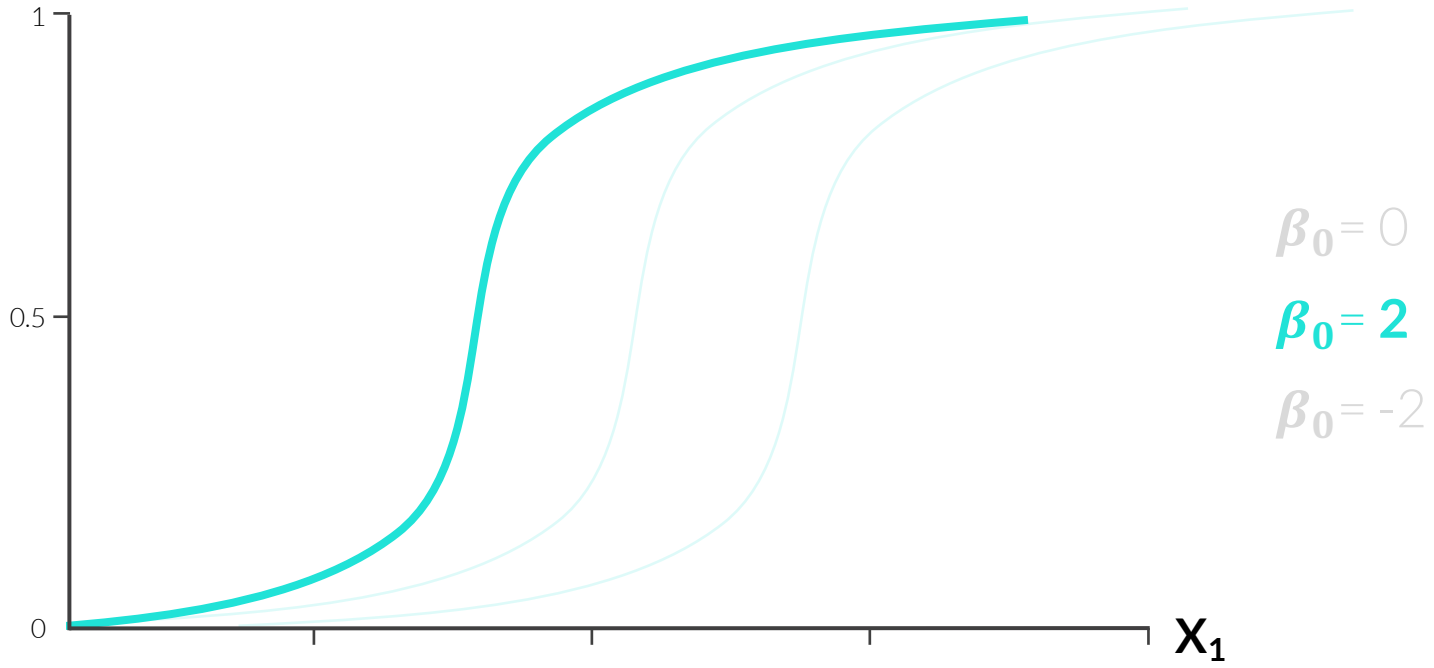


LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where β_0 is the intercept, **X** is the feature value and β_1 is the weight *(or slope)*

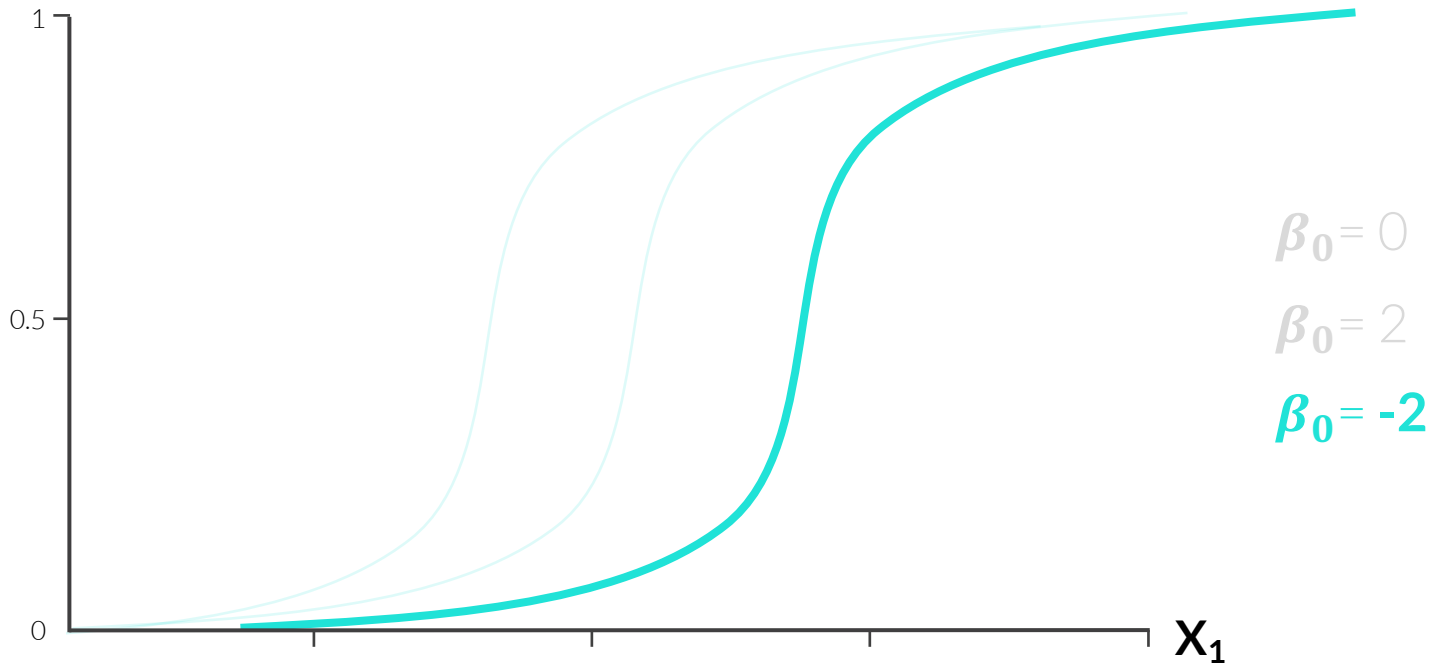


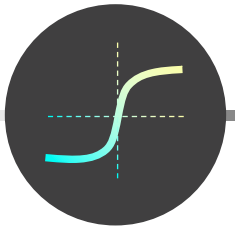
LOGISTIC EQUATION

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

Linear equation, where β_0 is the intercept, **X** is the feature value and β_1 is the weight *(or slope)*





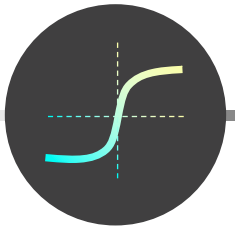
LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values

Model Output	Actual Observation: Y = **0**	Actual Observation: Y = **1**
~**0**	HIGH	LOW
~**1**	LOW	HIGH

- When our model output is close to the actual Y, we want likelihood to be **HIGH** (near **1**)
- When our model output is far from the actual Y, we want likelihood to be **LOW** (near **0**)



LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values

Model Output	Actual Observation: Y = **0**	Actual Observation: Y = **1**
~**0**	HIGH	LOW
~**1**	LOW	HIGH

LIKELIHOOD FUNCTION:

$$\mathcal{L} = (\boldsymbol{output})^{\boldsymbol{y}} * (\mathbf{1} - \boldsymbol{output})^{\mathbf{1}-\boldsymbol{y}}$$

LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values

Model Output	Actual Observation: Y = **0**	Actual Observation: Y = **1**
~**0**	HIGH	LOW
~**1**	LOW	HIGH

$$(output)^{y} * (1 - output)^{1-y}$$

$$(.99)^{1} * (1 - .99)^{1-1}$$

$$(.99)^{1} * (.01)^{0}$$

$$.99 * 1$$

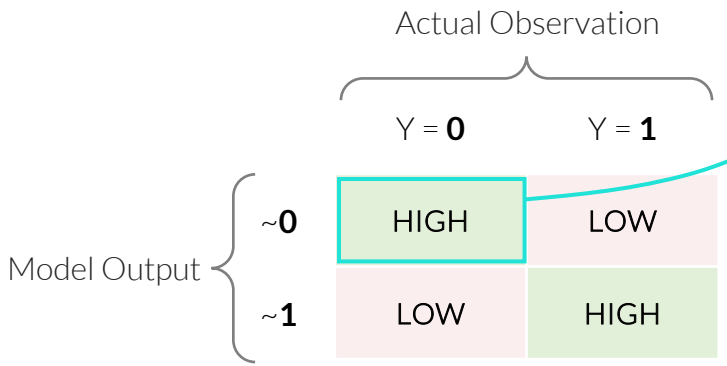
$$=$$

.99

LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values



$$(output)^{y} * (1 - output)^{1-y}$$

$$(.01)^{0} * (1 - .01)^{1-0}$$

$$1 * (.99)^{1}$$

$$1 * .99$$

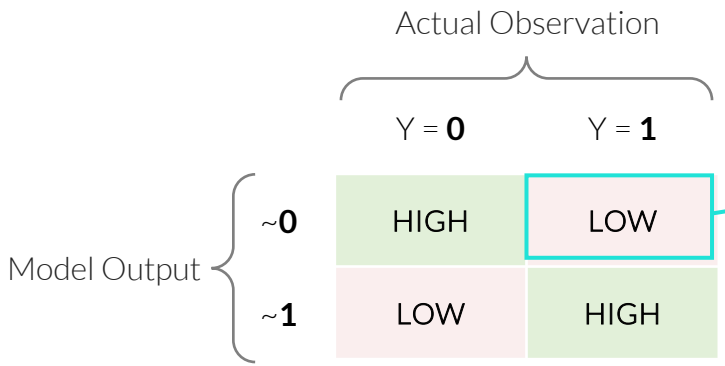
$$=$$

$$.99$$

LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values



$$(output)^{y} * (1 - output)^{1-y}$$

$$(.01)^{1} * (1 - .01)^{1-1}$$

$$.01 * (.99)^{0}$$

$$.01 * 1$$

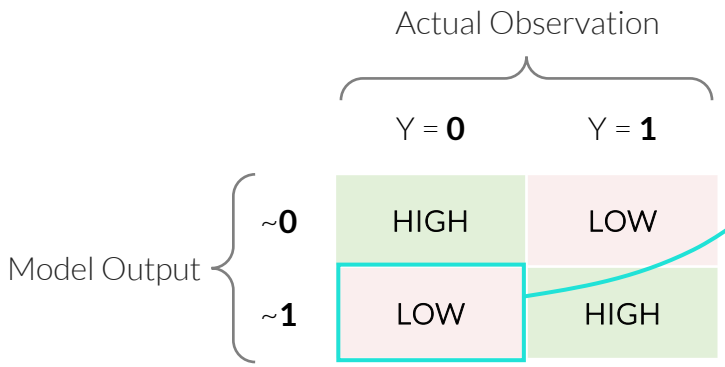
=

.01

LIKELIHOOD

How do we determine the "right" values for β?

- **Likelihood** is a metric that tells us how good our model is at correctly predicting Y, based on the shape of the curve
- This is where **machine learning** comes in; instead of human trial-and-error, an algorithm determines the best weights to maximize likelihood using observed values



$$(output)^{y} * (1 - output)^{1-y}$$

$$(.99)^{0} * (1 - .99)^{1-0}$$

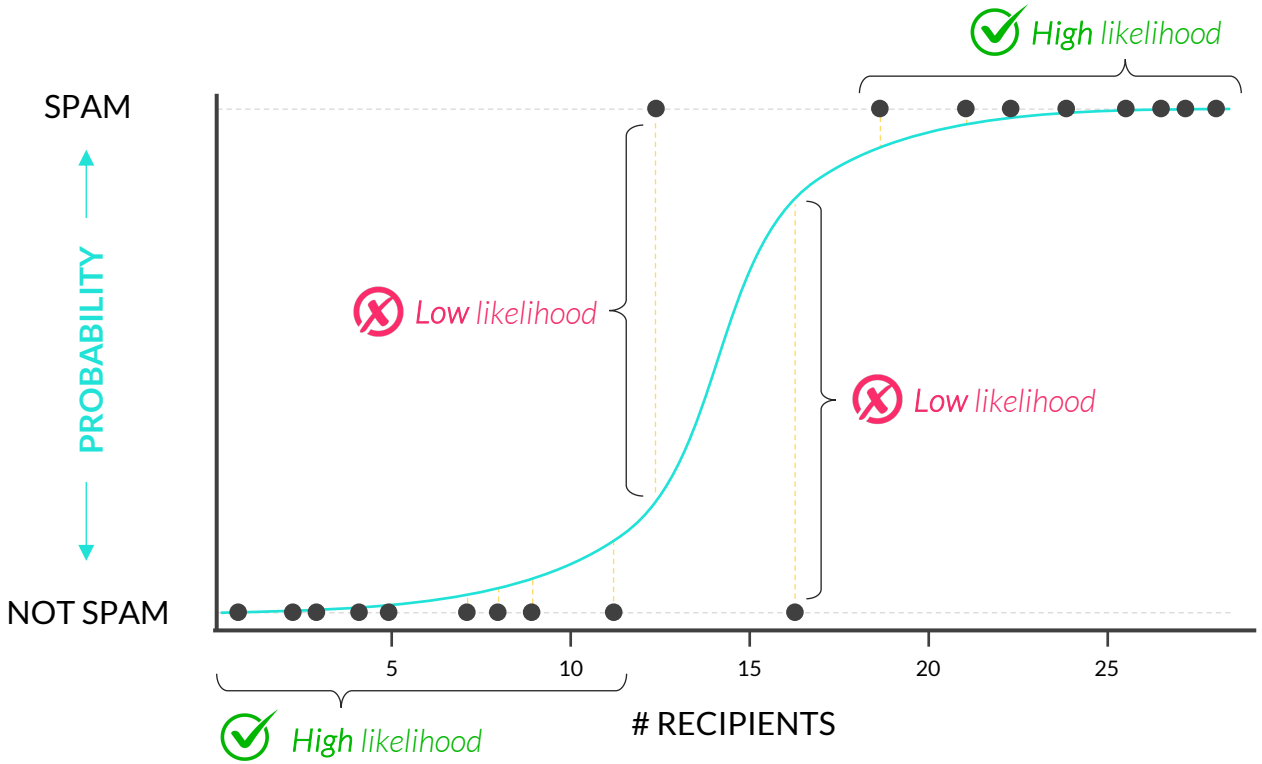
$$1 * (.01)^{1}$$

$$1 * .01$$

$$=$$

$$.01$$

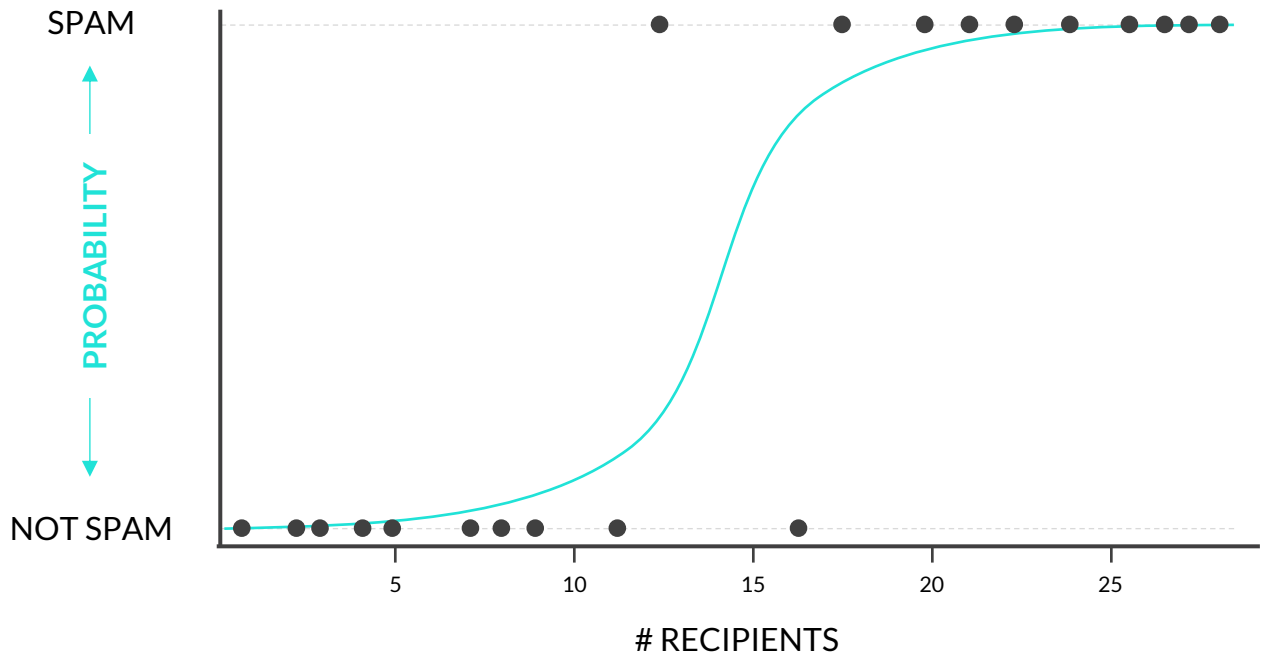
LIKELIHOOD



- Observations closest to the curve have the **highest likelihood values** *(and vice versa)*, so maximizing total likelihood allows us to find the curve that fits our data best

MULTIPLE LOGISTIC REGRESSION

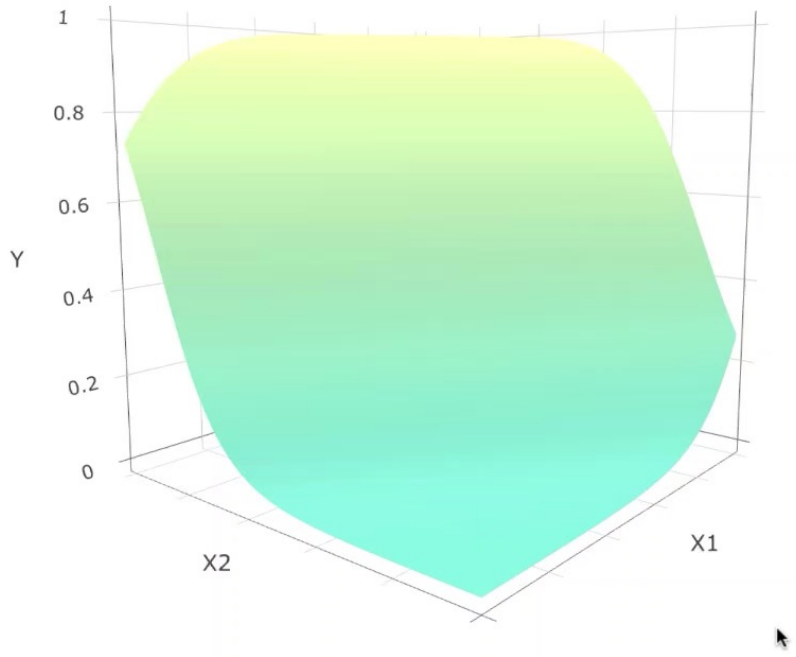
What if *multiple* X variables could help predict Y?



- **# of Recipients** can help us detect spam, but so can other variables like the **number of typos**, **count of words like "free" or "bonus"**, **sender reputation score**, etc.

MULTIPLE LOGISTIC REGRESSION

What if *multiple* X variables could help predict Y?



- Logistic regression can handle **multiple features**, but visual interpretation breaks down at >2 features *(this is why we need machine learning!)*

MULTIPLE LOGISTIC REGRESSION

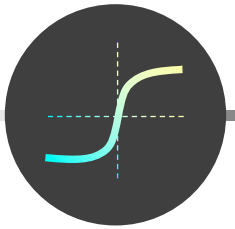
What if *multiple* X variables could help predict Y?

Makes the output fall between **0** and **1**

$$\frac{1}{1 + e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_n x_n)}}$$

Weighted independent variables (x_1, x_2...x_n)

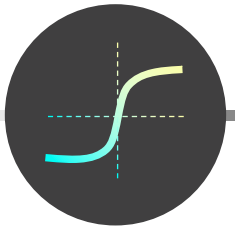
- Logistic regression is about finding the **best combination of weights** (β_1, β_2...β_n) for a given set features (x_1, x_2...x_n) to maximize the likelihood function



LOGISTIC REGRESSION WORKFLOW

Follow the **logistic regression workflow** to ensure that you fit an accurate model:

1. **Split** training and test datasets to accurately assess model performance
2. **Scale** or standardize the data if you are going to use regularization
3. **Fit & Tune** the model by adding or removing features and adjusting the regularization hyperparameters
4. **Score** the model on the test data after fitting your final model on the combined training & validation sets



FITTING A LOGISTIC REGRESSION MODEL

In Python, you can **fit a logistic regression model** using the **scikit-learn** library

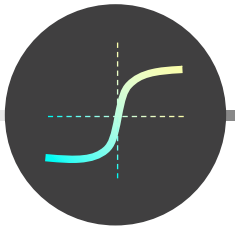
- Sklearn.linear_model.LogisticRegression()
- **model.coef_** and **model.intercept_** return the fitted coefficients and intercept

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()

lr = logreg.fit(X_train, y_train)
```

```
print(f"Coefficients: {lr.coef_}")
print(f"Intercept: {lr.intercept_}")
```

```
Coefficients: [[-5.82819478e-03 -5.92319067e-07 -7.15206560e-04]]
Intercept: [-0.00127207]
```

All coefficients are negative, which doesn't make sense given the positive correlations between Purchase, Age, and Estimated Salary



SCORING MODEL ACCURACY

In Python, you can **fit a logistic regression model** using the **scikit-learn** library

- **model.score(X, y)** returns an accuracy score for a given set of features and target values
- **confusion_matrix(y_actual, y_pred)** returns a confusion matrix

```
print(f"Train Accuracy: {lr.score(X_train, y_train)}")
print(f"Test Accuracy: {lr.score(X_test, y_test)}")
```

```
Train Accuracy: 0.653125
Test Accuracy: 0.6
```

Our accuracy was very poor compared to KNN. We should look at a confusion matrix to diagnose…

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, lr.predict(X_test))
```

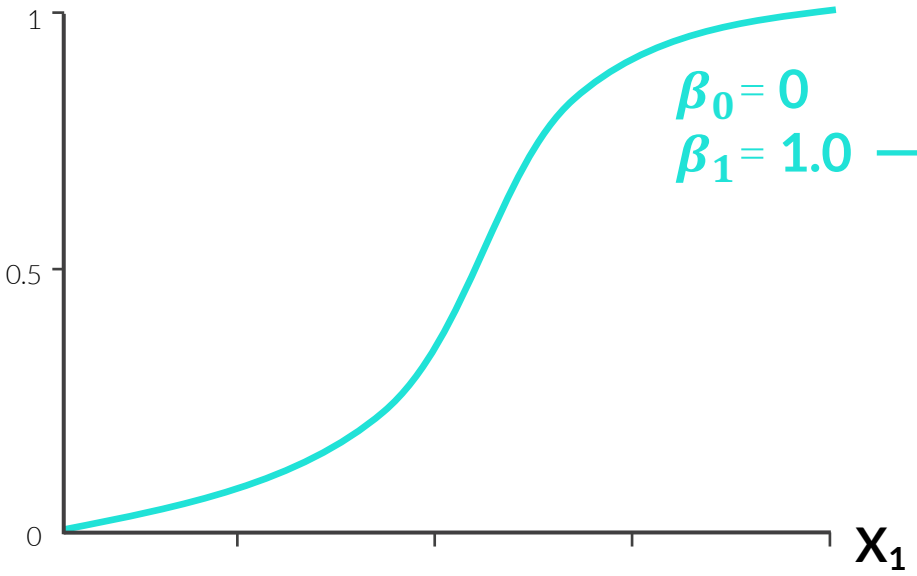
```
array([[48,  0],
       [32,  0]])
```

Our accuracy is even worse than it looks – we're not predicting a single customer will purchase! Hopefully feature selection or engineering can fix this problem…

INTERPRETING COEFFICIENTS

The **coefficients** of a logistic regression model represent the **change in log odds**

- **Positive** coefficients mean the predicted probability *increases* as the feature increases
- **Negative** coefficients means the predicted probability *decreases* as the feature increases
- The **odds ratio**, calculated by exponentiating the coefficient, represents the multiplicative change in odds caused by a one unit increase in the feature

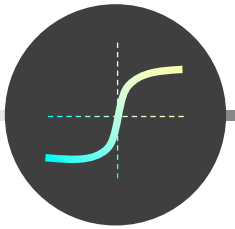


$\beta_0 = 0$

$\beta_1 = 1.0 \longrightarrow e^1 = 2.72$

*Increasing X_1 by 1 unit leads to a **2.72x** increase in the odds of our target being 1*

While odds have a constant increase, **the change in probability depends on where you are in the range of X,** as demonstrated by the shape of the curve on the left.



INTERPRETING COEFFICIENTS

The **coefficients** of a logistic regression model represent the **change in log odds**

- **Positive** coefficients mean the predicted probability *increases* as the feature increases
- **Negative** coefficients means the predicted probability *decreases* as the feature increases
- The **odds ratio**, calculated by exponentiating the coefficient, represents the multiplicative change in odds caused by a one unit increase in the feature

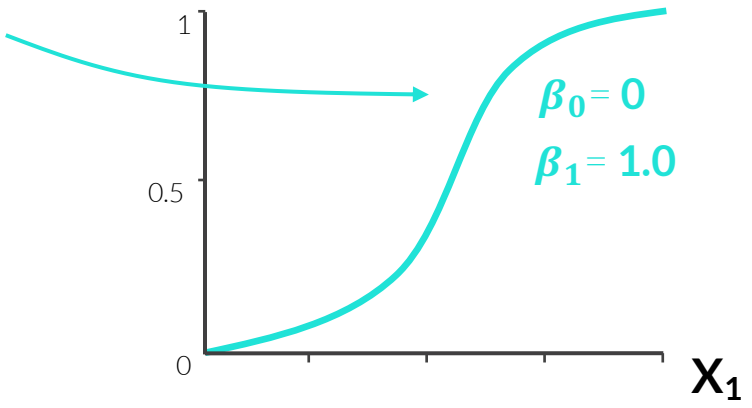
Linear coefficient predict log odds:

$$\log(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1$$

Solving for p yields a probability curve:

$$p = \frac{1}{1+e^{-\beta x}}$$

X_1 Value	Log Odds ($\beta_0 + \beta_1 x_1$)	Predicted Probability
-1	0 + 1(-1) = -1	.269
0	0 + 1(0) = 0	.500
1	0 + 1(1) = 1	.731
2	0 + 1(2) = 2	.881
5	0 + 1(5) = 5	.993



ASSIGNMENT: LOGISTIC REGRESSION

NEW MESSAGE

October 8, 2023

From: **Larry Logit** *(Sr. Statistician)*

Subject: **Logistic Regression**

Hi there!

Your KNN model looked pretty good! But you should try a logistic regression model as well – they usually outperform KNN and are better for production!

Can you fit a logistic regression model on the income data? Start with the variables 'age' and 'hoursperweek' to establish a baseline.

Thanks!

03_logistic_regression_assignments.ipynb

Reply Forward

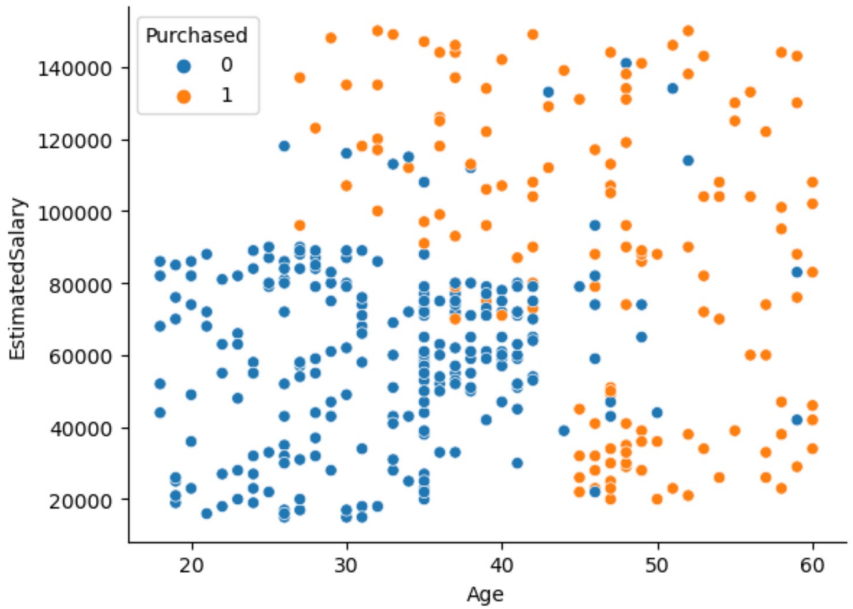
Key Objectives

1. Fit a Logistic Regression Model in Python
2. Generate a confusion matrix and report accuracy for the test data

FEATURE ENGINEERING & SELECTION

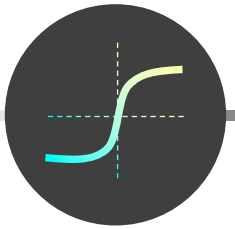
Feature engineering is a critical part of the modeling process

- Thorough EDA, an understanding of ML algorithms, and subject matter expertise can all help with feature engineering and selection



Purchasers don't always have high income or age – there is an interaction between these variables.

KNN works very well for this type of data, but logistic regression coefficients estimate a constant increase or decrease across continuous features. Feature engineering may help here!



FEATURE ENGINEERING & SELECTION

Feature engineering is a critical part of the modeling process

- Thorough EDA, an understanding of ML algorithms, and subject matter expertise can all help with feature engineering and selection

```
ads = ads.assign(
    age45 = np.where(ads["Age"] > 45, 1, 0),
    salary90 = np.where(ads["EstimatedSalary"] > 90000, 1, 0),
    agexsal = ads["Age"] * ads["EstimatedSalary"]
)

ads.head()
```

	User ID	Gender	Age	EstimatedSalary	Purchased	age45	salary90	agexsal
0	15624510	Male	19	19000	0	0	0	361000
1	15810944	Male	35	20000	0	0	0	700000
2	15668575	Female	26	43000	0	0	0	1118000
3	15603246	Female	27	57000	0	0	0	1539000
4	15804002	Male	19	76000	0	0	0	1444000

Features	Test Accuracy
Gender + Age + Salary	.6
Gender + Age	.763
Gender + Salary	.725
Gender + Salary + Age + agexsal	.763
Gender + age45 + salary90	.875

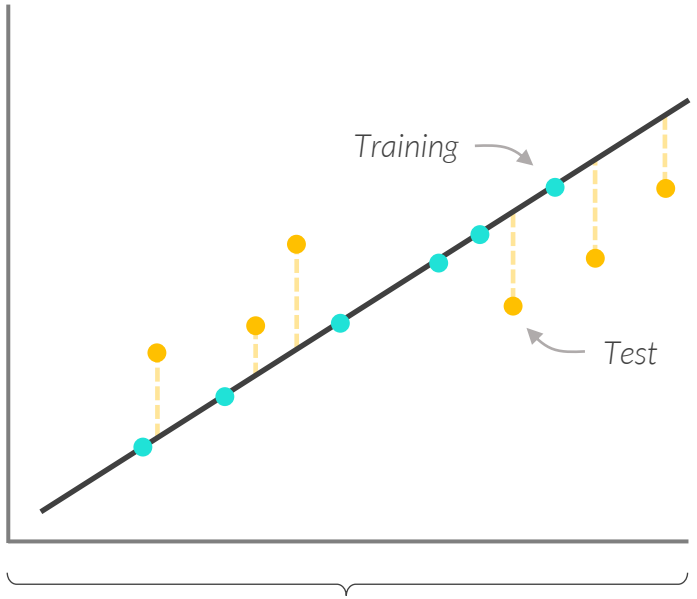
Our model that used hard cut points as features performed much better than ones using continuous features

REGULARIZATION

Regularized models intentionally underfit training data by NOT choosing the model that best fits training data, with the purpose of reducing the variance in the test data

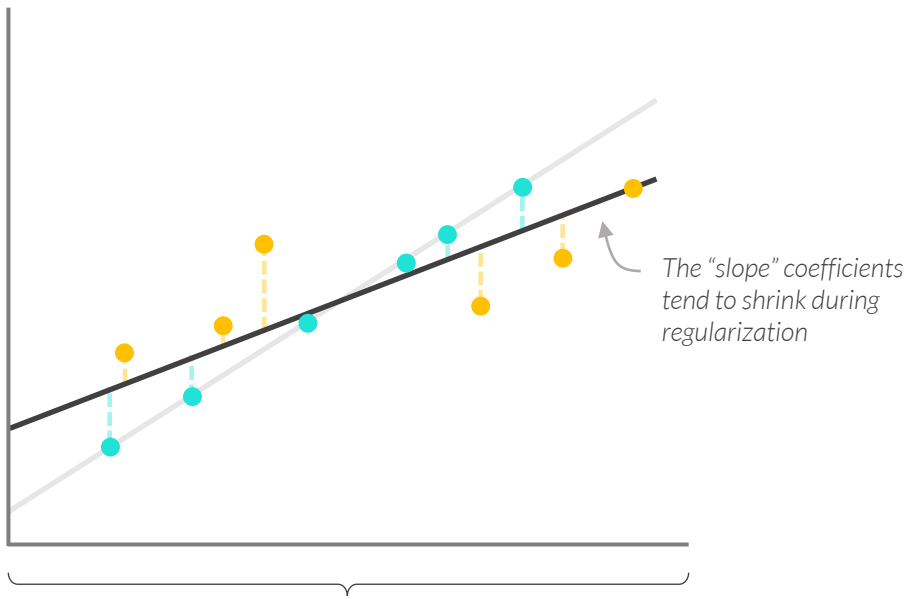
- This helps reduce overfitting and leads to better predictions (especially in smaller data sets)

Linear Equation:

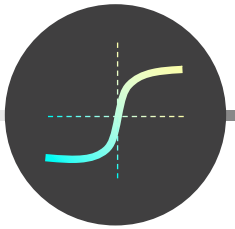


Low bias, high variance (overfit model)

Regularized Linear Equation:



Balance of bias and variance



THE COST FUNCTION

In logistic regression, the **cost function** is the negative log of likelihood (Log Loss):

$$J = -\log(likelihood)$$

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

*The **negative log of likelihood (Log Loss)***

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

$$J = -\log(likelihood) + \alpha R$$

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

*The **regularization** term*

TYPES OF REGULARIZED REGRESSION

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

$$J = -\log(likelihood) + \alpha R$$

Lasso (L1) Penalty

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

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

Ridge (L2) Penalty

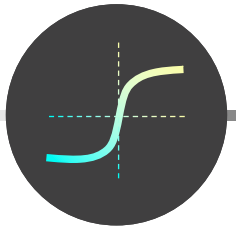
*Sum of the **squared** coefficient values*

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

Elastic Net Penalty

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

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



TUNING A REGULARIZED MODEL

- Model Overview
- Logistic Equation
- Likelihood
- Multiple Logistic Regression
- Fitting & Scoring
- Regularization & Tuning
- Multi-Class Models
- Pros & Cons

Hyperparameters are settings that affect how well a model fits the data

- Unlike model parameters (like regression coefficients) these are not learned from the data
- Parameters are set before the model is fit and optimized based on validation performance

K-Nearest Neighbors

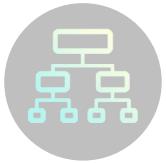
Examples:

- *K*
- *Distance metric*
- *weights*

Logistic Regression

Examples:

- *Regularization Strength*
- *Penalty Type*
- *Solver*



Decision Trees

Examples:

- *Criterion (entropy, gini)*
- *Tree depth*
- *Leaf size*

Random Forests

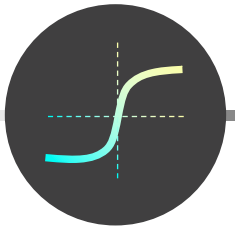
Examples:

- *# Trees*
- *# Samples*
- *# Features*
- *Re-Sampling*

Gradient Boosted Trees

Examples:

- *# Trees*
- *# Learning Rate*
- *# Sample Size*

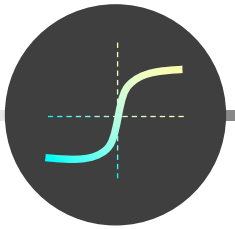


TUNING A REGULARIZED MODEL

Tuning a regularized logistic regression model can help prevent overfitting and lead to better out of sample performance

- **Feature scaling** is required prior to fitting your model because regularization impacts coefficient magnitudes, which are related to feature unit scales

```
from sklearn.preprocessing import StandardScaler

std = StandardScaler()

X_train_std = std.fit_transform(X_train)
X_test_std = std.transform(X_test)
```



TUNING A REGULARIZED MODEL

Tuning a regularized logistic regression model can help prevent overfitting and lead to better out of sample performance

- After scaling, we'll use **GridSearchCV** to test combos of regularization strength and type
- LogisticRegression(C=1.0, penalty='l2', solver="saga")

```
from sklearn.model_selection import GridSearchCV

parameters = {
    "C": np.linspace(.1, 2, 20),
    "penalty": ["l1", "l2"]
}

gridsearch = GridSearchCV(LogisticRegression(solver="saga"), parameters)

gridsearch.fit(X_train_std, y_train)

gridsearch.best_params_
```

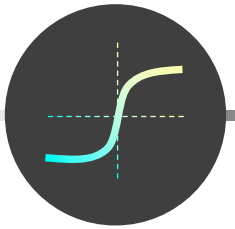
__C__ is the regularization strength (smaller = less complexity)

__Penalty__ selects the type of regularization ['__l1__', '__l2__', '__elasticnet__']

Set up cross validation loop to select best parameters (__solver="saga"__ is required for some types of regularization)

```
{'C': 0.3, 'penalty': 'l1'}
```

Best regularization strength was __0.3__ using '__l1__' (__lasso__) penalization



TUNING A REGULARIZED MODEL

Tuning a regularized logistic regression model can help prevent overfitting and lead to better out of sample performance

- Once we've determined the optimal hyperparameters, we'll **fit and score** our final model
- LogisticRegression(C=1.0, penalty='l2', solver="saga")

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(C=.3, penalty="l1", solver="saga")

lr = logreg.fit(X_train_std, y_train)

print(f"Train Accuracy: {lr.score(X_train_std, y_train)}")
print(f"Test Accuracy: {lr.score(X_test_std, y_test)}")
```

```
Train Accuracy: 0.853125
Test Accuracy: 0.875
```

With regularization our test performance was much better than our original logistic model, even with the same features!

ASSIGNMENT: TUNED LOGISTIC REGRESSION

NEW MESSAGE

October 10, 2023

From: **Larry Logit** *(Sr. Statistician)*

Subject: **RE: Logistic Regression**

Hi there!

The baseline model looks great! Now let's try to fit a fully fledged logistic regression model by leveraging all the features available to us and tuning regularization parameters.

If we see significant improvements this very well could be our MVP production model!

03_logistic_regression_assignments.ipynb

Reply Forward

Key Objectives

1. Perform feature selection and feature engineering as part of the modeling process
2. Tune logistic regression regularization hyperparameters

MULTI-CLASS LOGISTIC REGRESSION

Can I use logistic regression to predict *multiple* classes?

- Logistic regression can also be used for **multi-class classification problems**
- These are less common in business settings and tend to be more challenging

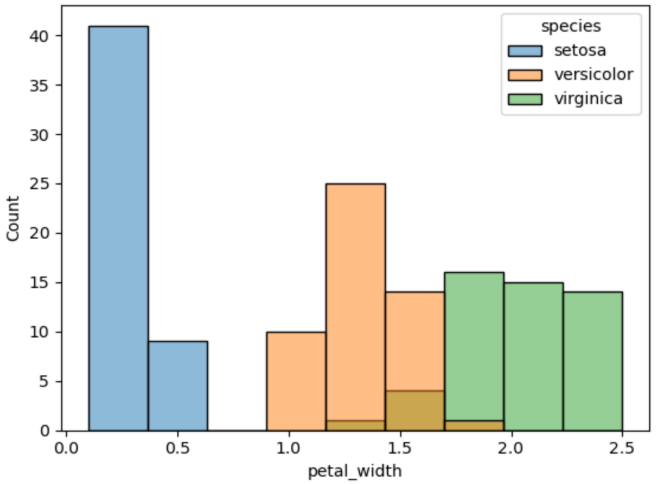
EXAMPLE | *Predicting a species of iris flower*

```
import seaborn as sns

iris = sns.load_dataset("iris")
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

*The famous **iris dataset** has three different species of iris flowers and measurements for each flower*

```
sns.histplot(data=iris, x="petal_width", hue="species");
```



***Setosa** has a much smaller petal width than the other species. **Versicolor** and **virginica** are similar, but virginica has wider petals on average.*

Petal width should be a great feature to separate species!

MULTI-CLASS LOGISTIC REGRESSION

Can I use logistic regression to predict *multiple* classes?

- LogisticRegression(multi_class="auto")
- Logistic regression is only capable of predicting *binary* outcomes, so one model will be fit for each class with the others grouped as the negative class

```
lr = LogisticRegression(max_iter=1000, multi_class="auto")

lr.fit(X, y)
```

```
lr.classes_
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object
```

The classes_ attribute helps tie class labels to their model outputs – the first set of outputs will belong to 'setosa', then 'versicolor', and so on

```
lr.coef_
```

```
array([[-0.42333655,  0.96718172, -2.51725526, -1.0793699 ],
       [ 0.53430021, -0.3215421 , -0.20640337, -0.94418398],
       [-0.11096366, -0.64563963,  2.72365863,  2.02355388]])
```

One set of parameters is returned for each class.

*At the far right, we see that setosa and versicolor have **negative** slopes for petal width, while virginica (which had the largest petal width, has a **positive** coefficient*

```
lr.intercept_
```

```
array([  9.84952339,   2.23795871, -12.0874821 ])
```

MULTI-CLASS LOGISTIC REGRESSION

Can I use logistic regression to predict *multiple* classes?

- LogisticRegression(multi_class="auto")
- Logistic regression is only capable of predicting *binary* outcomes, so one model will be fit for each class with the others grouped as the negative class

```
lr = LogisticRegression(max_iter=1000, multi_class="auto")

lr.fit(X, y)
```

```
lr.score(X, y)
```

```
0.9733333333333334
```

```
confusion_matrix(y, y_pred)
```

```
array([[50,  0,  0],
       [ 0, 47,  3],
       [ 0,  1, 49]])
```

Our model performed very well in separating species

```
lr.predict_proba(X).round(2)
```

```
[0.15, 0.85, 0.  ],
[0.  , 0.9 , 0.1 ],
[0.04, 0.91, 0.05],
[0.06, 0.94, 0.01],
[0.02, 0.9 , 0.09],
[0.01, 0.98, 0.01],
[0.  , 0.78, 0.22],
[0.07, 0.92, 0.01],
[0.01, 0.93, 0.07],
[0.01, 0.77, 0.22],
[0.02, 0.97, 0.02],
```

One probability for each class is also generated.

In most cases our model is quite confident about which class each row belongs to.

ASSIGNMENT: MULTI-CLASS LOGISTIC REGRESSION

NEW MESSAGE

October 12, 2023

From: **Larry Logit** *(Sr. Statistician)*

Subject: **Multiclass Logistic Regression**

Hi there!

I need to you to look at a dataset for a project we have coming up. I'm debating whether to make this a multi-class problem or convert it to a binary classification problem.

Can you fit a prototype multi-class logistic regression and report test performance? I've put more details in the notebook.

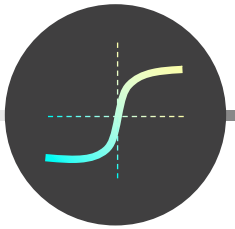
Thanks!

03_logistic_regression_assignments.ipynb

Reply Forward

Key Objectives

1. Fit a Multiclass Logistic Regression
2. Generate an accuracy score and confusion matrix for test data



PROS & CONS OF LOGISTIC REGRESSION

Logistic regression has some pros and cons that data scientists should be aware of when deciding whether it is the best model for the task at hand

- In general, logistic regression is a great choice for classification and is among the most commonly used algorithms; however, it tends to be less accurate than ensembled trees

Interpretability

We can review logistic regression coefficients to understand and communicate their direct impact on classification scores

Simplicity

Logistic regression has few hyperparameters to tune, and tends to perform well on small datasets

Fast Scoring

Logistic Regression's linear structure makes it highly performant when deployed in production environments

Multicollinearity

Logistic regression assumes that the features are not highly correlated. Highly correlated features can cause problems with coefficient stability & interpretability

Feature Engineering

Unlike tree-based models which naturally capture non-linear relationships and interactions between features, you will often need to invest significant time feature engineering to achieve similar performance, which reduces the model's interpretability

PRO TIP: Logistic regression is a great model to start with when working on a new classification problem due to its combination of performance and interpretability

KEY TAKEAWAYS

- Logistic Regression is a **linear classification model**
  - *It is highly interpretable and simple in structure, although it tends to underperform ensembled trees*
- **The Logit Link Function** constrains model outputs between 0 and 1
  - *Understanding this link is the key to being able to interpret model coefficients*
- **Regularization hyperparameters** can be tuned to reduce overfitting
  - *Regularization penalizes model complexity and has the impact of shrinking model coefficients, which can lead to better out of sample model performance*
- **Multiclass logistic regression** allows us to model more complex problems
  - *Multiclass logistic regression models are actually a series of binary models, rather than a single model with three possible outcomes*