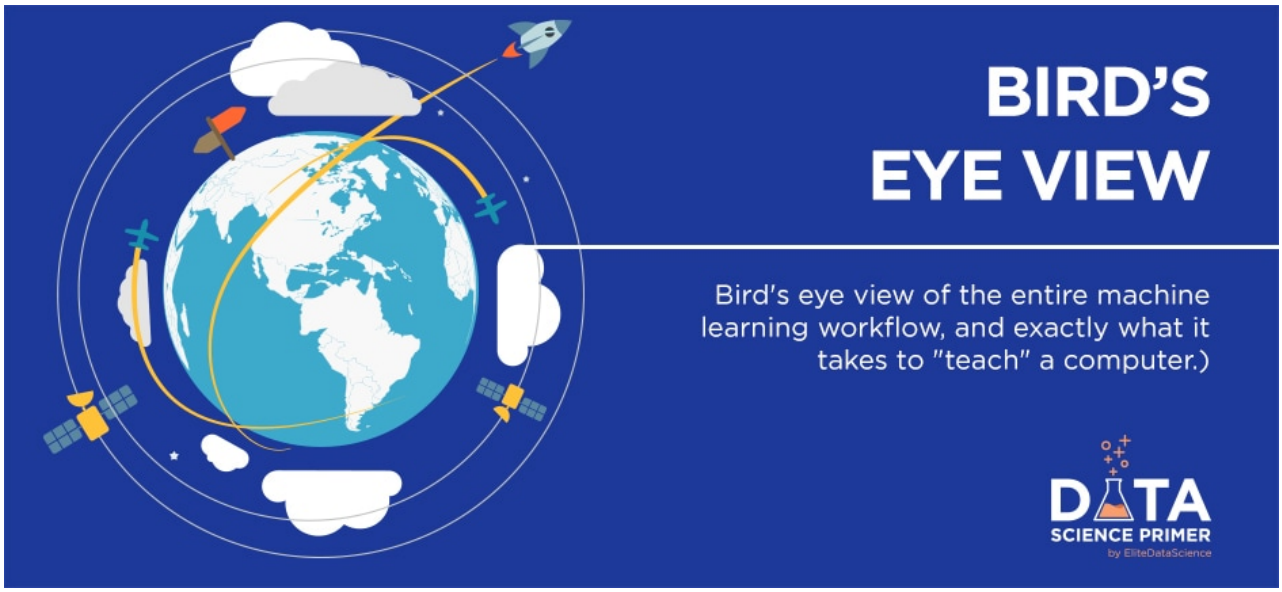
# Bird's Eye View

elitedatascience.com/birds-eye-view




Welcome to our 7-part mini-course on data science and applied machine learning!

Over these 7 chapters, our goal is to provide you with an end-to-end blueprint for applied machine learning, while keeping this as actionable and succinct as possible.

With that, let's get started with a bird's eye view of the **machine learning workflow**.

First things first. One really cool (optional) challenge you can do in the next hour is training your first machine learning model!

That's right, we've put together a complete step-by-step tutorial for training a model that can predict wine quality. Feel free to go check it out at any time.

Now, tutorials like that are excellent for getting your feet wet, but if you want to consistently get great results with machine learning, **you must develop a reliable, systematic approach to solving problems.**

And that's what we'll tackle throughout the rest of this mini-course.

## Machine Learning ≠ Algorithms

First, we must clear up one of the biggest misconceptions about machine learning:

**Machine learning is not about algorithms.**

When you open a textbook or a university syllabus, you'll often be greeted by a grocery list of algorithms.

This has fueled the misconception that machine learning is about mastering dozens of algorithms. However, it's much more than that...

**Machine learning is a comprehensive approach to solving problems...**

...and individual algorithms are only one piece of the puzzle. The rest of the puzzle is *how you apply them the right way*.

Machine learning is **not** just a list of algorithms.


## What makes machine learning so special?

Machine learning is the practice of teaching computers how to learn patterns from data, often for making decisions or predictions.

For true machine learning, the computer must be able to learn patterns **that it's not explicitly programmed to identify**.

### Example: the curious child

A young child is playing at home... And he sees a **candle**! He cautiously waddles over.

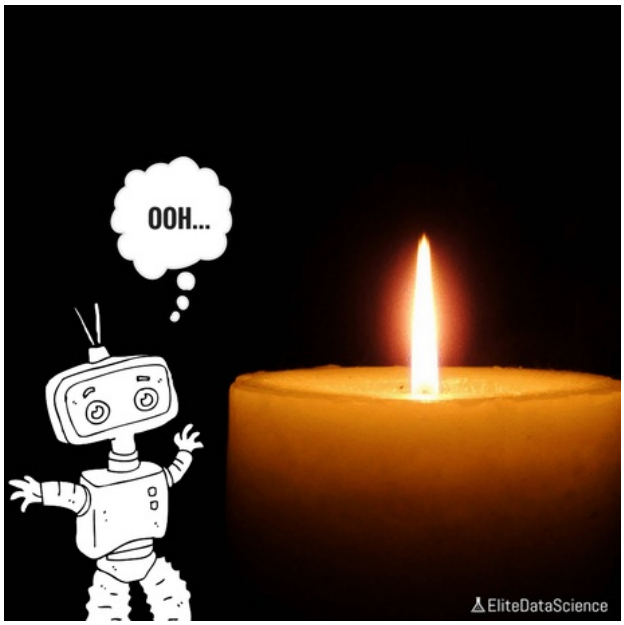
1. Out of curiosity, he sticks his hand over the candle flame.
2. "Ouch!," he yells, as he yanks his hand back.
3. "Hmm... that **red and bright** thing really hurts!"

Ooh a candle!

Two days later, he's playing in the kitchen... And he sees a **stove-top**! Again, he cautiously waddles over.

1. He's curious again, and he's thinking about sticking his hand over it.
2. Suddenly, he notices that it's **red and bright**!

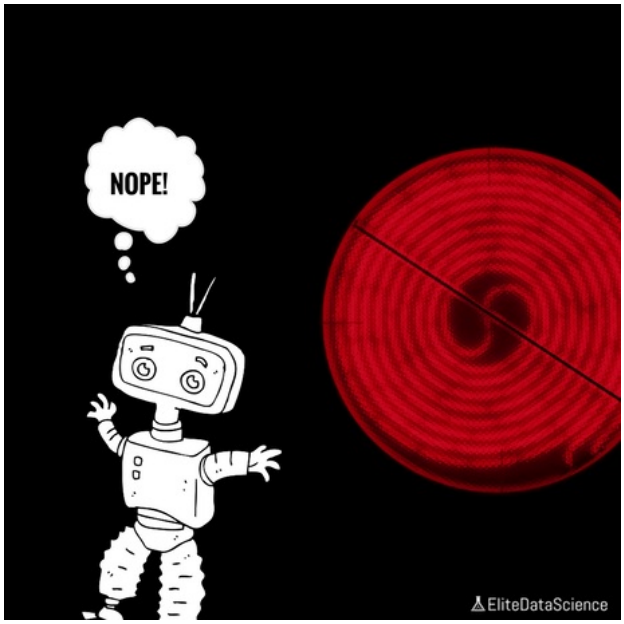
3. "Ahh..." he thinks to himself, "not today!"
4. He remembers that **red and bright** means pain, and he ignores the stove top.




To be clear, it's only machine learning because the child learned **patterns** from the candle.

- He learned that the pattern of **"red and bright means pain."**
- On the other hand, if he ignored the stove-top simply *because his parents warned him,* that'd be "explicit programming" instead of machine learning.

#thanksmachinelearning




## Key Terminology

For this mini-course, we will focus on developing practical intuition instead of diving into technicalities (which we'll save for Chapter 7: Next Steps).

Therefore, it's even more important to be clear and concise with our terminology.

Before going any further, let's just make sure we have a shared language for discussing these topics:

- **Model** - a set of patterns learned from data.
- **Algorithm** - a specific ML process used to train a model.
- **Training data** - the dataset from which the algorithm learns the model.

- **Test data** - a new dataset for reliably evaluating model performance.
- **Features** - Variables (columns) in the dataset used to train the model.
- **Target variable** - A specific variable you're trying to predict.
- **Observations** - Data points (rows) in the dataset.

### Example: Primary school students

| | Age | Gender | Weight | Height |
|---|---|---|---|---|
| Observation #1 | **12** | **M** | **80** lbs | **55** in |
| Observation #2 | **11** | **M** | **85** lbs | **58** in |
| Observation #3 | **12** | **F** | **73** lbs | **52** in |
| Observation #4 | **10** | **F** | **71** lbs | **49** in |
| ⋮ | | | | |
| Observation #150 | | | | |

Features | Target Variable

For example, let's say you have a dataset of 150 primary school students, and you wish to predict their Height based on their Age, Gender, and Weight...

- You have 150 observations...
- 1 target variable (Height)...
- 3 features (Age, Gender, Weight)...
- You might then separate your dataset into two subsets:
    1. Set of 120 used to train several models (training set)
    2. Set of 30 used to pick the best model (test set)

By the way, we'll explain why separate training and test sets are super important in Chapter 6: Model Training.

## Machine Learning Tasks

Academic machine learning starts with and focuses on individual algorithms. However, in applied machine learning, you should first pick the right machine learning task for the job.

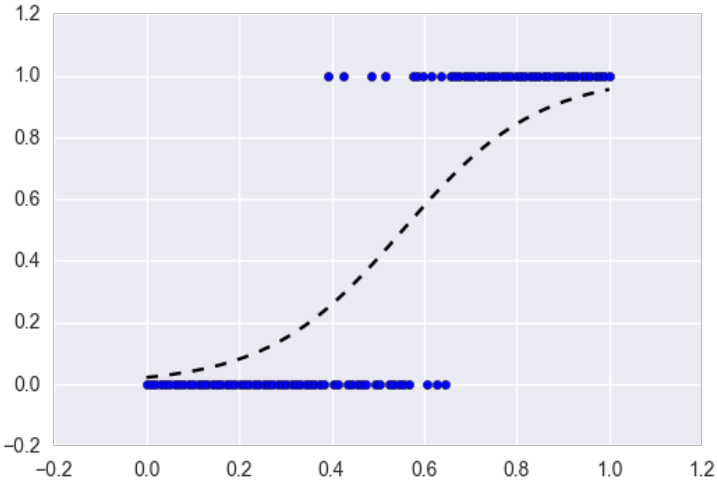
- A **task** is a specific objective for your algorithms.
- Algorithms can be swapped in and out, as long as you pick the right task.
- In fact, you should **always try multiple algorithms** because you most likely won't know which one will perform best for your dataset.

The two most common categories of tasks are supervised learning and unsupervised learning. (There are other tasks as well, but the concepts you'll learn in this course will be widely applicable.)

### Supervised Learning

Supervised learning includes tasks for "labeled" data (i.e. you have a target variable).

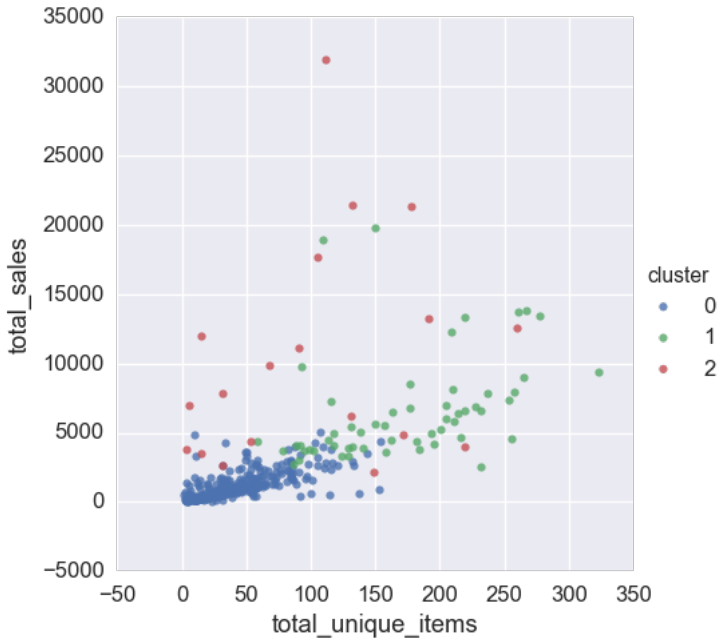
- In practice, it's often used as an advanced form of *predictive modeling*.
- Each observation must be labeled with a "correct answer."
- Only then can you build a predictive model because you must tell the algorithm what's "correct" while training it (hence, "supervising" it).
- **Regression** is the task for modeling *continuous target variables*.
- **Classification** is the task for modeling *categorical (a.k.a. "class") target variables*.




## Unsupervised Learning

Unsupervised learning includes tasks for "unlabeled" data (i.e. you do not have a target variable).

- In practice, it's often used either as a form of *automated data analysis* or *automated signal extraction*.
- Unlabeled data has no predetermined "correct answer."
- You'll allow the algorithm to directly learn patterns from the data (without "supervision").
- **Clustering** is the most common unsupervised learning task, and it's for *finding groups* within your data.




## The 3 Elements of Great Machine Learning

How to consistently build effective models that get great results.

### #1: A skilled chef (human guidance)

First, even though we are "teaching computers to learn on their own," human guidance plays a huge role.

- As you'll see, you'll need to make dozens of decisions along the way.
- In fact, the very first major decision is how to road-map your project for *guaranteed success*.

Don't worry, we'll share our step-by-step blueprint with you.

### #2: Fresh ingredients (clean, relevant data)

The second essential element is the *quality of your data*.

- Garbage In = Garbage Out, no matter which algorithms you use.
- Professional data scientists spend most their time understanding the data, cleaning it, and engineering new features.

While that sounds open-ended, you'll get our proven frameworks that you can always rely on as starting points.

### #3: Don't overcook it (avoid overfitting)

One of the most dangerous pitfalls in machine learning is **overfitting**. An overfit model has *"memorized" the noise* in the training set, instead of learning the true underlying patterns.

- An overfit model within a hedge fund can cost millions of dollars in losses.
- An overfit model within a hospital can costs thousands of lives.
- For most applications, the stakes won't be quite *that* high, but overfitting is still the single largest mistake you must avoid.

We'll teach you strategies for preventing overfitting by (A) choosing the right algorithms and (B) tuning them correctly.

## The Blueprint

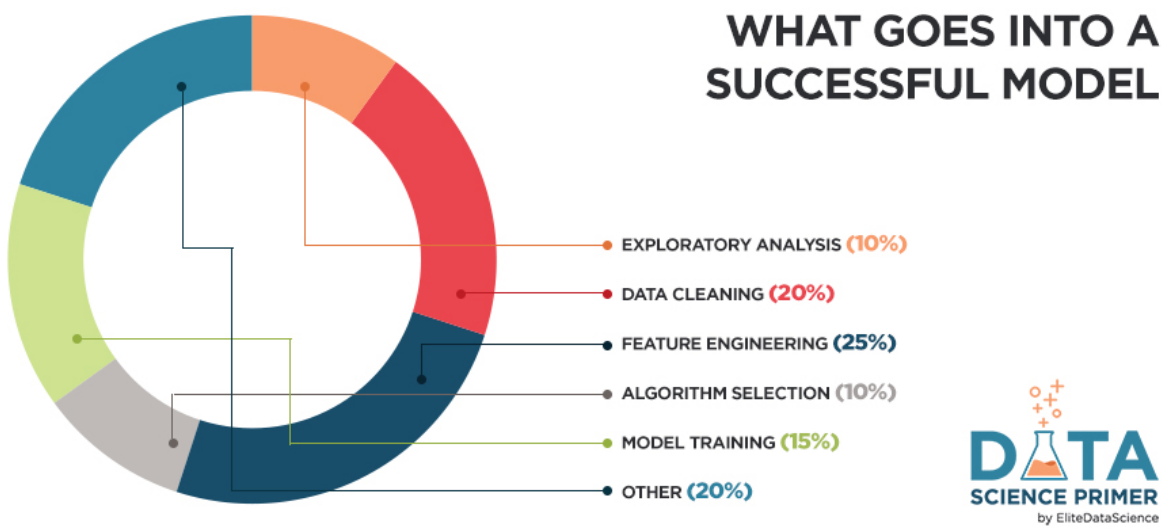
Our machine learning blueprint is designed around those 3 elements.

### There are 5 core steps:

- 1
  First, "get to know" the data. This step should be quick, efficient, and decisive.
- 2
  Then, clean your data to avoid many common pitfalls. Better data beats fancier algorithms.
- 3

Next, help your algorithms "focus" on what's important by creating new features.

- 4
  Choose the best, most appropriate algorithms without wasting your time.
- 5
  Finally, train your models. This step is pretty formulaic once you've done the first 4.




## Of course, there are other situational steps as well:

- S
  Sometimes, you'll need to roadmap the project and anticipate data needs.
- W
  You may also need to restructure your dataset into a format that algorithms can handle.
- P
  Often, transforming your features first can further improve performance.
- E
  You can squeeze out even more performance by combining multiple models.

However, for this mini-course, we're going to focus on the 5 core steps. The other ones slot in easily once you understand the core workflow.

"Get the fundamentals down and the level of everything you do will rise." ~ Michael Jordan

**Key takeaway:** Machine learning should not be haphazard and piecemeal. It should be systematic and organized.

Furthermore, even if you forget everything else taught in this course, please remember: **'Better data beats fancier algorithms'** - this insight will serve you well.


"GET THE
FUNDAMENTALS
DOWN AND THE LEVEL
OF EVERYTHING YOU
DO WILL RISE."
ELITEDATASCIENCE
Ft. Michael Jordan
Silo
True Value

# Exploratory Analysis

elitedatascience.com/exploratory-analysis


Welcome to our 7-part mini-course on data science and applied machine learning!

In the previous chapter, we saw an overview of the entire machine learning workflow. We saw how the "80/20" of data science includes 5 core steps.

In this chapter, we will dive into the first of those core steps: **exploratory analysis**.

This step should not be confused with data visualization or summary statistics. Those are merely tools... means to an end.

Proper exploratory analysis is about answering questions. It's about extracting enough insights from your dataset to course correct *before* you get lost in the weeds.

In this guide, we explain which insights to look for. Let's get started.

## Why explore your dataset upfront?

The purpose of exploratory analysis is to **"get to know"** the dataset. Doing so upfront will make the rest of the project much smoother, in 3 main ways:

1. You'll gain valuable hints for Data Cleaning (which can make or break your models).
2. You'll think of ideas for Feature Engineering (which can take your models from good to great).
3. You'll get a "feel" for the dataset, which will help you communicate results and deliver greater impact.

However, exploratory analysis for machine learning should be **quick, efficient, and decisive**... not long and drawn out!

Don't skip this step, but don't get stuck on it either.

You see, there are infinite possible plots, charts, and tables, but you only need a **handful** to "get to know" the data well enough to work with it.

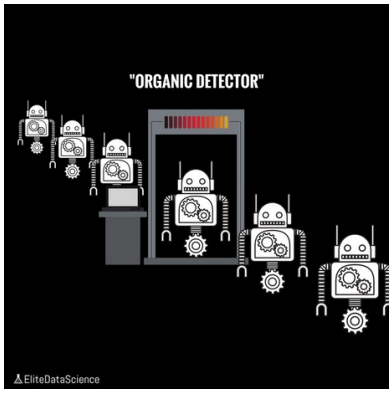
In this lesson, we'll show you the visualizations that provide the biggest bang for your buck.

## Start with Basics

First, you'll want to answer a set of basic questions about the dataset:

- How many observations do I have?
- How many features?
- What are the data types of my features? Are they numeric? Categorical?
- Do I have a target variable?

Know what you're working with.




### Example observations

Then, you'll want to display example observations from the dataset. This will give you a "feel" for the values of each feature, and it's a good way to check if everything makes sense.

Here's an example from the real-estate dataset used in Project 2 of our Machine Learning Masterclass (you can scroll right to see more columns):

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 295850 | 1 | 1 | 584 | 2013 | 0 | Apartment / Condo / Townhouse | Wood Siding | NaN | NaN | 107 | 9 | 30 | 19 | 89 | 6 | 47 | 58 | 33.0 | 65.0 | 84.0 | 234.0 |
| **1** | 216500 | 1 | 1 | 612 | 1965 | 0 | Apartment / Condo / Townhouse | Brick | Composition Shingle | 1.0 | 105 | 15 | 6 | 13 | 87 | 2 | 26 | 14 | 39.0 | 73.0 | 69.0 | 169.0 |
| **2** | 279900 | 1 | 1 | 615 | 1963 | 0 | Apartment / Condo / Townhouse | Wood Siding | NaN | NaN | 183 | 13 | 31 | 30 | 101 | 10 | 74 | 62 | 28.0 | 15.0 | 86.0 | 216.0 |
| **3** | 379900 | 1 | 1 | 618 | 2000 | 33541 | Apartment / Condo / Townhouse | Wood Siding | NaN | NaN | 198 | 9 | 38 | 25 | 127 | 11 | 72 | 83 | 36.0 | 25.0 | 91.0 | 265.0 |
| **4** | 340000 | 1 | 1 | 634 | 1992 | 0 | Apartment / Condo / Townhouse | Brick | NaN | NaN | 149 | 7 | 22 | 20 | 83 | 10 | 50 | 73 | 37.0 | 20.0 | 75.0 | 88.0 |

The purpose of displaying examples from the dataset is not to perform rigorous analysis. Instead, it's to get a **qualitative "feel"** for the dataset.

- Do the columns make sense?
- Do the values in those columns make sense?
- Are the values on the right scale?
- Is **missing data** going to be a big problem based on a quick eyeball test?

## Plot Numerical Distributions

Next, it can be very enlightening to plot the distributions of your numeric features.

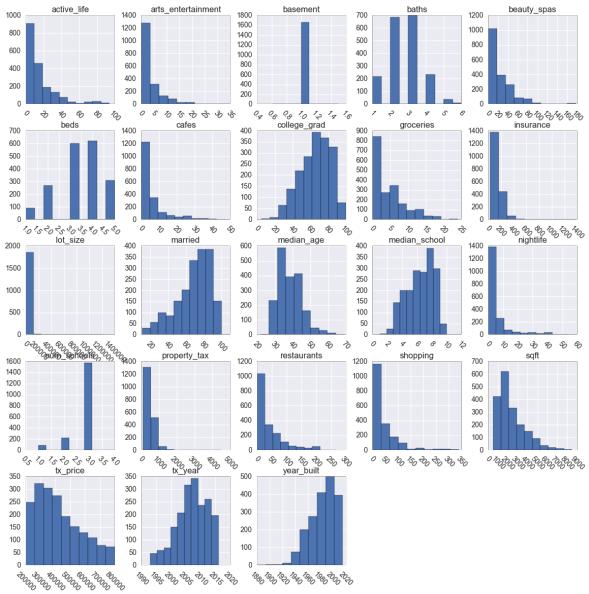
Often, a quick and dirty grid of **histograms** is enough to understand the distributions.

Here are a few things to look out for:

- Distributions that are unexpected
- Potential outliers that don't make sense
- Features that should be binary (i.e. "wannabe indicator variables")
- Boundaries that don't make sense
- Potential measurement errors

At this point, you should start making notes about potential fixes you'd like to make. If something looks out of place, such as a potential outlier in one of your features, now's a good time to ask the client/key stakeholder, or to dig a bit deeper.

However, we'll wait until Data Cleaning to make fixes so that we can keep our steps organized.

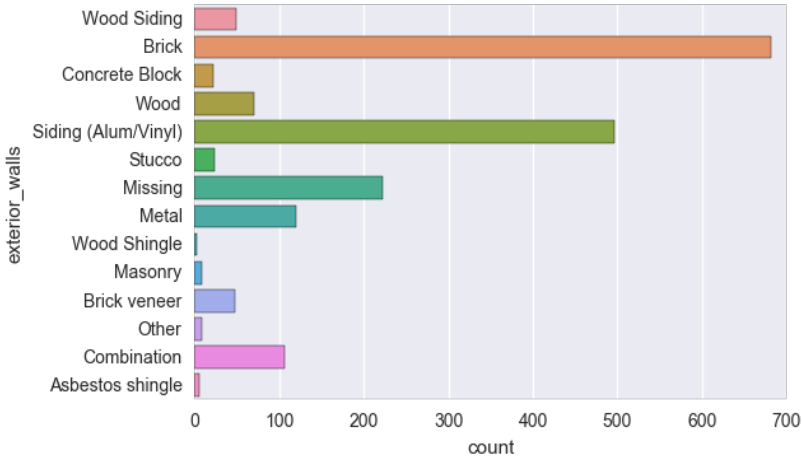



## Plot Categorical Distributions

Categorical features cannot be visualized through histograms. Instead, you can use **bar plots**.

In particular, you'll want to look out for **sparse classes**, which are classes that have a very small number of observations.

By the way, a **"class"** is simply a unique value for a categorical feature. For example, the following bar plot shows the distribution for a feature called *'exterior_walls'*. So Wood Siding, Brick, and Stucco are each classes for that feature.




Anyway, back to sparse classes... as you can see, some of the classes for *'exterior_walls'* have very short bars. Those are sparse classes.

They tend to be problematic when building models.

- In the best case, they don't influence the model much.
- In the worse case, they can cause the model to be **overfit**.

Therefore, we recommend making a note to *combine*or *reassign* some of these classes later. We prefer saving this until Feature Engineering (Lesson 4).

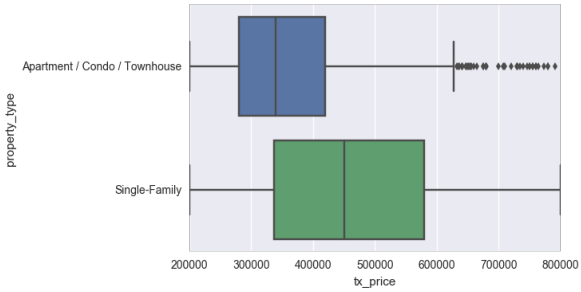
## Plot Segmentations

Segmentations are powerful ways to observe the *relationship between categorical features and numeric features*.

**Box plots** allow you to do so.

Here are a few insights you could draw from the following chart.

- The **median** transaction price (middle vertical bar in the box) for Single-Family homes was much higher than that for Apartments / Condos / Townhomes.
- The **min** and **max** transaction prices are comparable between the two classes.
- In fact, the round-number min ($200k) and max ($800k) suggest possible **data truncation**...
- ...which is very important to remember when assessing the **generalizability** of your models later!



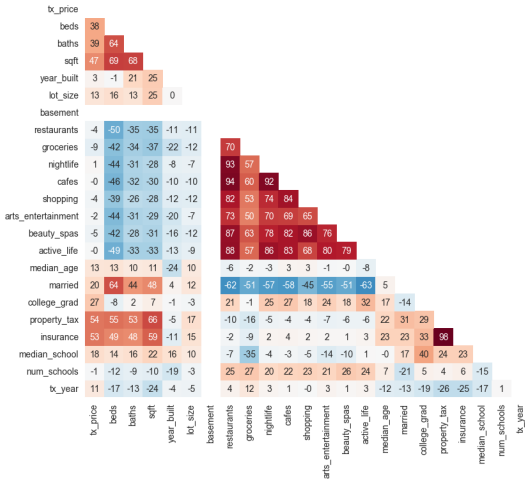

## Study Correlations

Finally, correlations allow you to look at the *relationships between numeric features and other numeric features*.

Correlation is a value between -1 and 1 that represents how closely two features move in unison. You don't need to remember the math to calculate them. Just know the following intuition:

- **Positive** correlation means that as one feature increases, the other increases. E.g. a child's age and her height.
- **Negative** correlation means that as one feature increases, the other decreases. E.g. hours spent studying and number of parties attended.
- Correlations near -1 or 1 indicate a **strong relationship**.
- Those closer to 0 indicate a **weak relationship**.
- 0 indicates **no relationship**.

Correlation **heatmaps** help you visualize this information. Here's an example (note: all correlations were multiplied by 100):




In general, you should look out for:

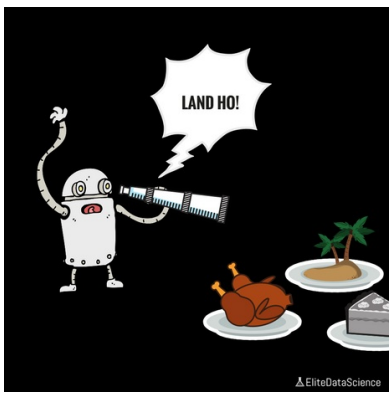
- Which features are strongly correlated with the **target variable**?
- Are there interesting or **unexpected** strong correlations between other features?

Again, your aim is to gain intuition about the data, which will help you throughout the rest of the workflow.

## Checkpoint Quiz

By the end of your Exploratory Analysis step, you'll have a pretty good understanding of the dataset, some notes for data cleaning, and possibly some ideas for feature engineering.

No one had the heart to tell Jerry that all he discovered was the "Bahamas Bashed Potatoes" weekly special...



**Here's a short quiz to check you got everything:**

- What types of features can have sparse classes? How would you check for them?
- What does it mean if *'sqft'* (size of property) has a correlation of 0.68 with *'baths'* (# of bathrooms)?
- What are 3 sanity checks to make by looking at example observations from the dataset?

---

Data Visualization with Python's Seaborn Library

# Data Cleaning

elitedatascience.com/data-cleaning


Welcome to our 7-part mini-course on data science and applied machine learning!

In the previous chapter, you learned about essential data visualizations for "getting to know" the data. More importantly, we explained the types of insights to look for.

Based on those insights, it's time to get our dataset into tip-top shape through **data cleaning**.

The steps and techniques for data cleaning will vary from dataset to dataset. As a result, it's impossible for a single guide to cover *everything* you might run into.

However, this guide provides **a reliable starting framework** that can be used every time. We cover common steps such as fixing structural errors, handling missing data, and filtering observations.

So let's put on our boots and clean up this mess!

## Better Data > Fancier Algorithms

Data cleaning is one those things that everyone does but no one really talks about. Sure, it’s not the "sexiest" part of machine learning. And no, there aren’t hidden tricks and secrets to uncover.

However, proper data cleaning can make or break your project. Professional data scientists usually spend a very large portion of their time on this step.

Why? Because of a simple truth in machine learning:

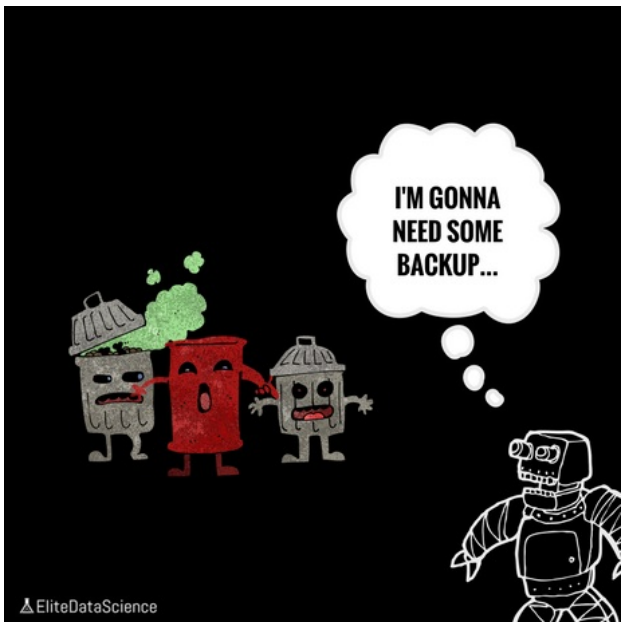
**Better data beats fancier algorithms.**

In other words... garbage in gets you garbage out. Even if you forget everything else from this course, please remember this point.

In fact, if you have a properly cleaned dataset, even simple algorithms can learn impressive insights from the data!

Obviously, different types of data will require different types of cleaning. However, the systematic approach laid out in this lesson can always serve as a good starting point.

Time to take out the garbage!

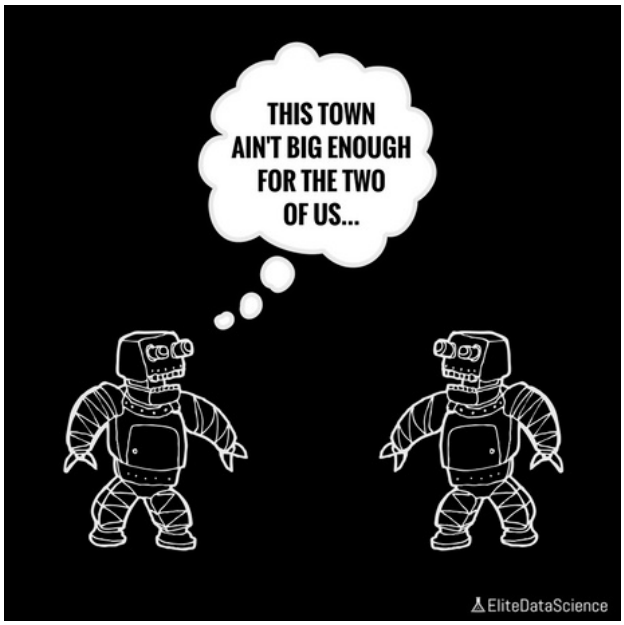



## Remove Unwanted observations

The first step to data cleaning is removing unwanted observations from your dataset.

This includes **duplicate** or **irrelevant** observations.

This town ain't big enough.

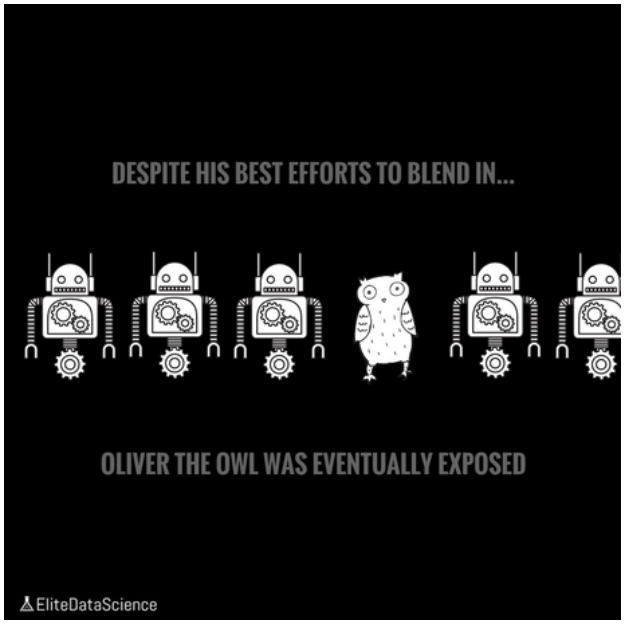



## Duplicate observations

Duplicate observations most frequently arise during **data collection**, such as when you:

- Combine datasets from multiple places
- Scrape data
- Receive data from clients/other departments

> It just wasn't meant to be.




## Irrelevant observations

*Irrelevant observations* are those that don't actually fit the **specific problem** that you're trying to solve.

- For example, if you were building a model for Single-Family homes only, you wouldn't want observations for Apartments in there.

- This is also a great time to review your charts from Exploratory Analysis. You can look at the distribution charts for categorical features to see if there are any classes that shouldn't be there.
- Checking for irrelevant observations **before engineering features** can save you many headaches down the road.

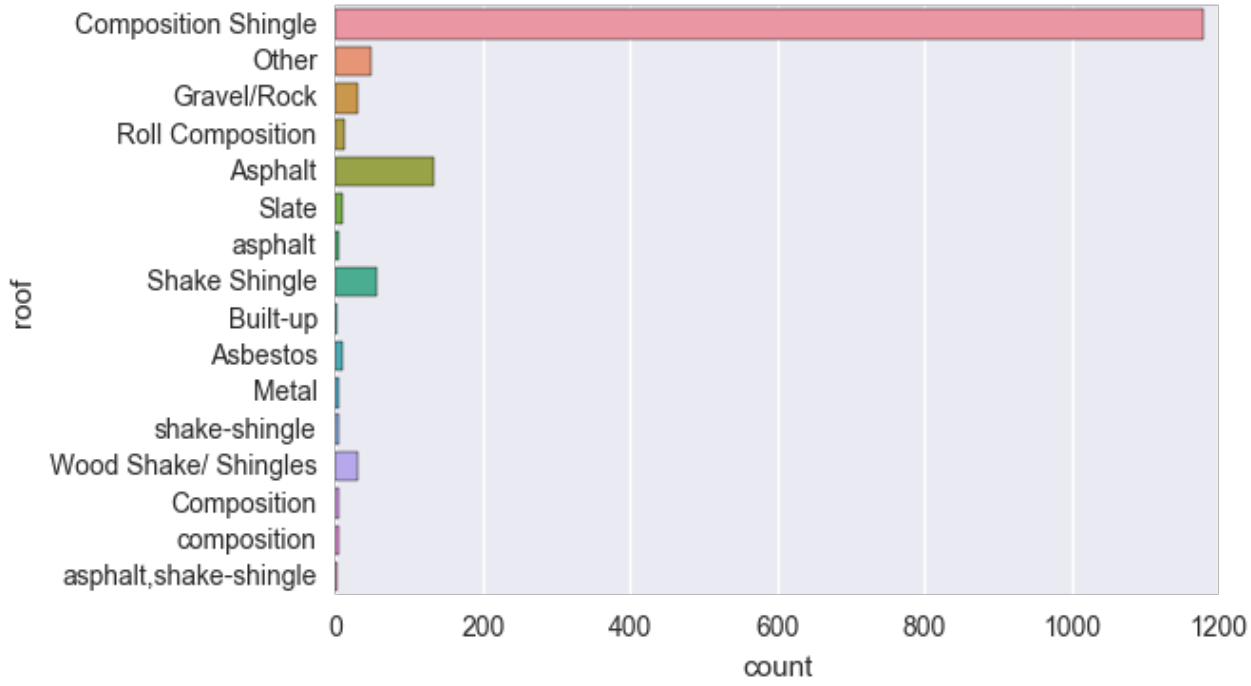
## Fix Structural Errors

The next bucket under data cleaning involves fixing structural errors.

Structural errors are those that arise during measurement, data transfer, or other types of **"poor housekeeping."**

For instance, you can check for **typos** or **inconsistent capitalization.** This is mostly a concern for categorical features, and you can look at your bar plots to check.

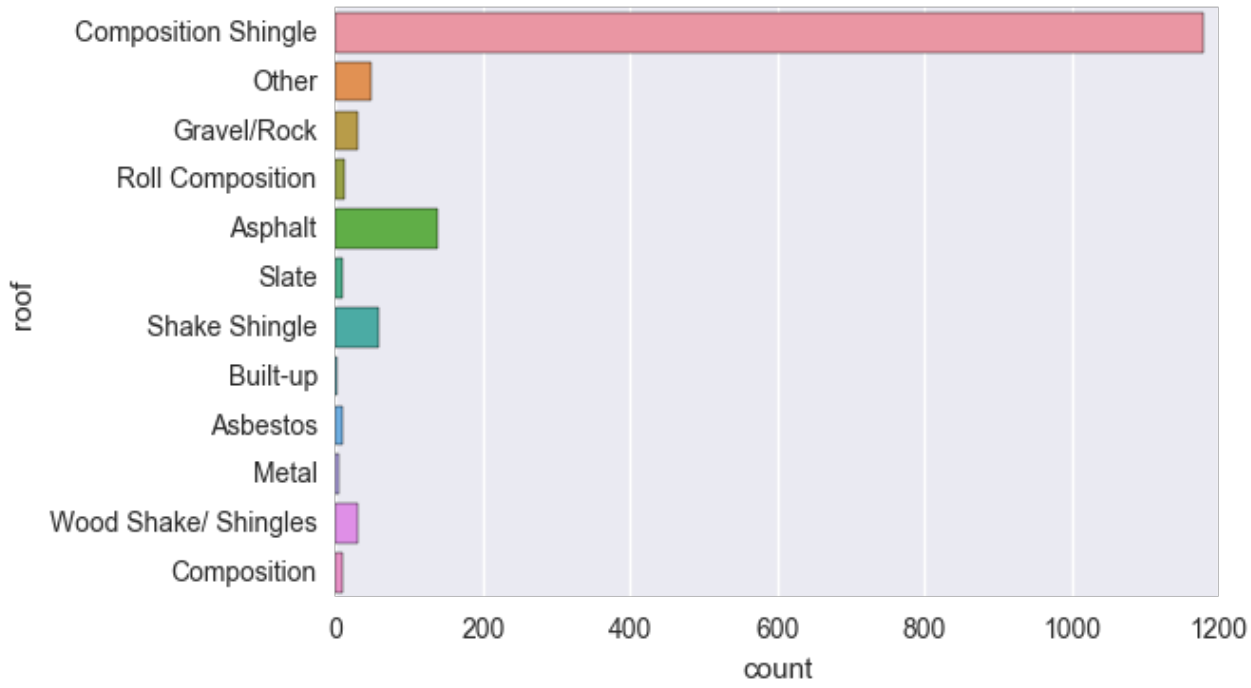
Here's an example:




As you can see:

- 'composition' is the same as 'Composition'
- 'asphalt' should be 'Asphalt'
- 'shake-shingle' should be 'Shake Shingle'
- 'asphalt,shake-shingle' could probably just be 'Shake Shingle' as well

After we replace the typos and inconsistent capitalization, the class distribution becomes much cleaner:




Finally, check for **mislabeled classes**, i.e. separate classes that should really be the same.

- e.g. If 'N/A' and 'Not Applicable' appear as two separate classes, you should combine them.
- e.g. 'IT' and 'information_technology' should be a single class.

## Filter Unwanted Outliers

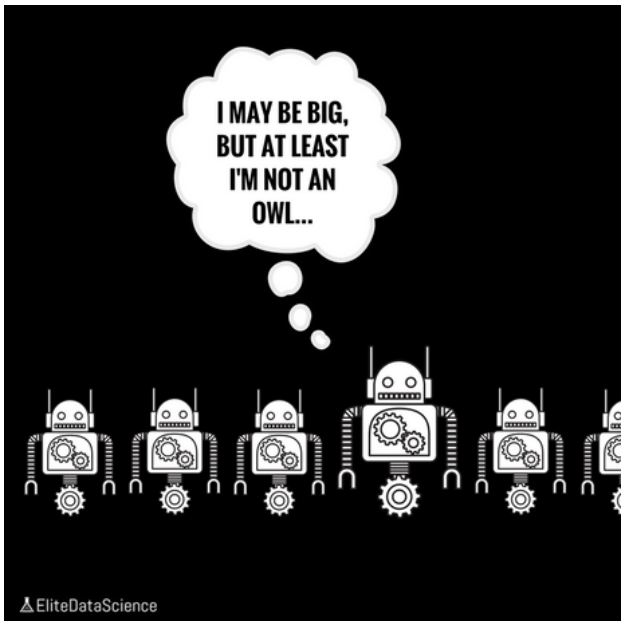
Outliers can cause problems with certain types of models. For example, linear regression models are less robust to outliers than decision tree models.

In general, if you have a **legitimate** reason to remove an outlier, it will help your model's performance.

However, outliers are **innocent until proven guilty.** You should never remove an outlier just because it's a "big number." That big number could be very informative for your model.

We can't stress this enough: you must have a good reason for removing an outlier, such as suspicious measurements that are unlikely to be real data.

He's got a point.




## Handle Missing Data

Missing data is a deceptively tricky issue in applied machine learning.

First, just to be clear, **you cannot simply ignore missing values in your dataset.** You must handle them in some way for the very practical reason that most algorithms do not accept missing values.

### "Common sense" is not sensible here

Unfortunately, from our experience, the 2 most commonly recommended ways of dealing with missing data actually suck.

They are:

1. **Dropping** observations that have missing values
2. **Imputing** the missing values based on other observations

Dropping missing values is sub-optimal because when you drop observations, you **drop information.**

- The fact that the value was missing may be informative in itself.
- Plus, in the real world, you often need to make predictions on new data even if some of the features are missing!

Imputing missing values is sub-optimal because the value was originally missing but you filled it in, which always leads to a loss in information, no matter how sophisticated your imputation method is.

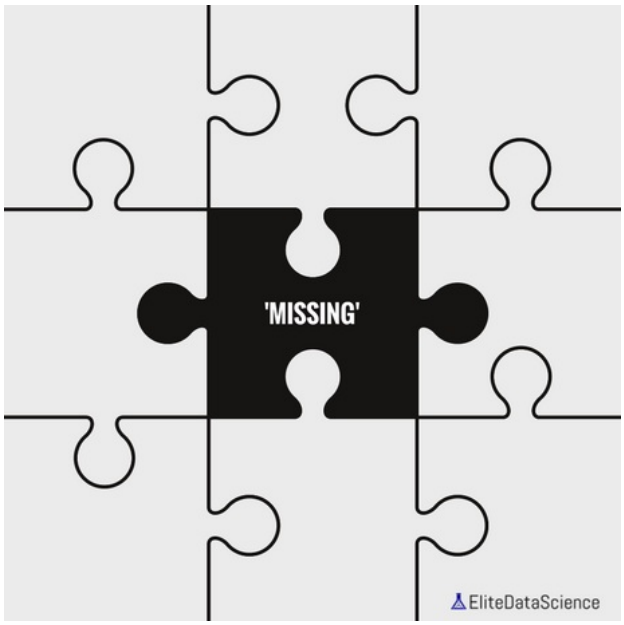
- Again, **"missingness"** is almost always informative in itself, and you should **tell your algorithm** if a value was missing.
- Even if you build a model to impute your values, you're not adding any real

information. You're just reinforcing the patterns already provided by other features.

Missing data is like missing a puzzle piece. If you drop it, that's like pretending the puzzle slot isn't there. If you impute it, that's like trying to squeeze in a piece from somewhere else in the puzzle.
In short, you should always tell your algorithm that a value was missing because **missingness is informative**.

So how can you do so?

The key is to tell your algorithm that the value was originally missing.




## Missing categorical data

The best way to handle missing data for *categorical* features is to simply label them as 'Missing'!

- You're essentially adding a *new class* for the feature.
- This tells the algorithm that the value was missing.
- This also gets around the technical requirement for no missing values.

## Missing numeric data

For missing *numeric* data, you should **flag and fill** the values.

1. Flag the observation with an indicator variable of missingness.
2. Then, fill the original missing value with 0 just to meet the technical requirement of no missing values.

By using this technique of flagging and filling, you are essentially **allowing the algorithm to estimate the optimal constant for missingness**, instead of just filling it in with the mean.

## Checkpoint Quiz

After properly completing the Data Cleaning step, you'll have a robust dataset that avoids many of the most common pitfalls.

This can really save you from a ton of headaches down the road, so please don't rush this step.

---

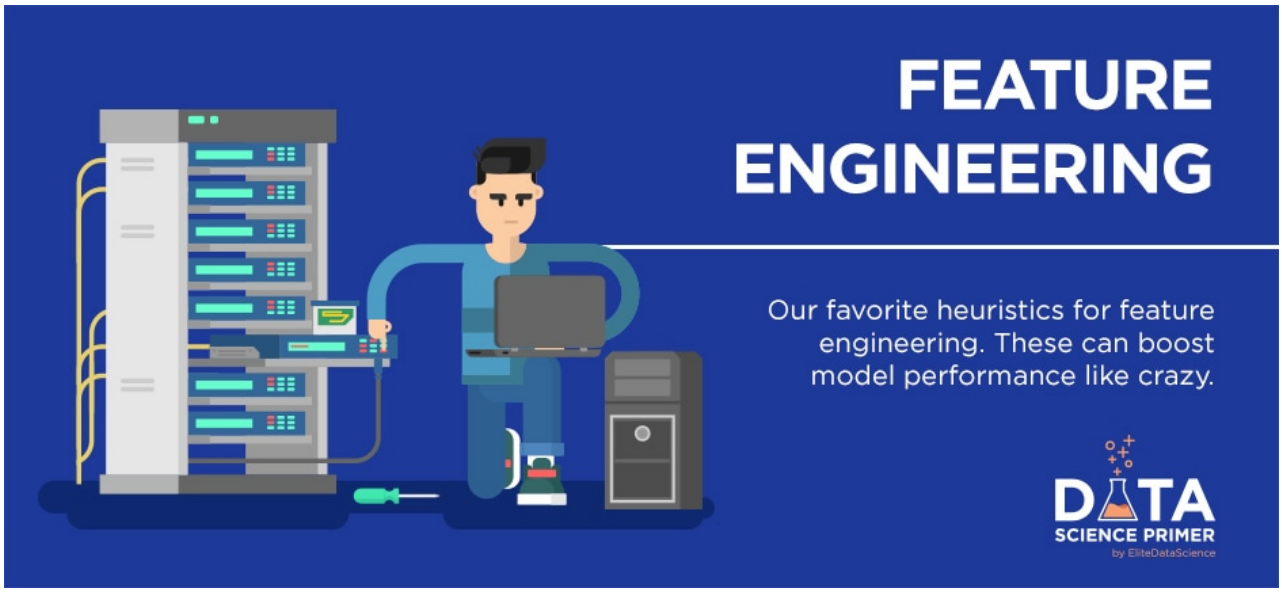
**Here's a quick quiz to check that you got everything:**

- What are 2 types of unwanted observations to remove from the start?
- What are 3 types of structural errors to look out for?
- How should you handle missing data?
- Why is it sub-optimal to drop observations with missing data or impute missing values?

---

# Feature Engineering

elitedatascience.com/feature-engineering

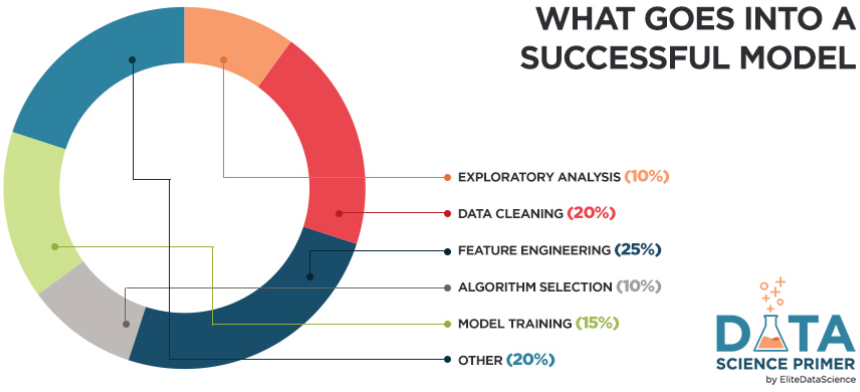



Welcome to our 7-part mini-course on data science and applied machine learning!

In the previous chapter, you learned a reliable framework for cleaning your dataset. We fixed structural errors, handled missing data, and filtered observations.

In this guide, we'll see how we can perform **feature engineering** to help out our algorithms and improve model performance.

Remember, out of all the core steps, data scientists usually spend the most time on feature engineering:




## What is Feature Engineering?

Feature engineering is about **creating new input features** from your existing ones.

In general, you can think of data cleaning as a process of subtraction and feature engineering as a process of addition.

This is often one of the most valuable tasks a data scientist can do to improve model performance, for 3 big reasons:

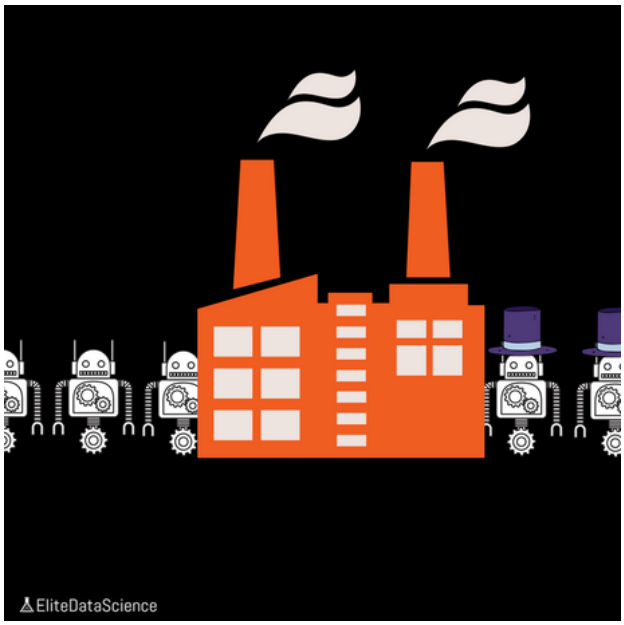
1. You can isolate and highlight key information, which helps your algorithms "focus" on what's important.
2. You can bring in your own **domain expertise**.
3. Most importantly, once you understand the "vocabulary" of feature engineering, you can bring in other people's domain expertise!

In this lesson, we will introduce several **heuristics** to help spark new ideas.

Before moving on, we just want to note that this is not an exhaustive compendium of all feature engineering because there are limitless possibilities for this step.

The good news is that this skill will naturally improve as you gain more experience.

Getting classy.

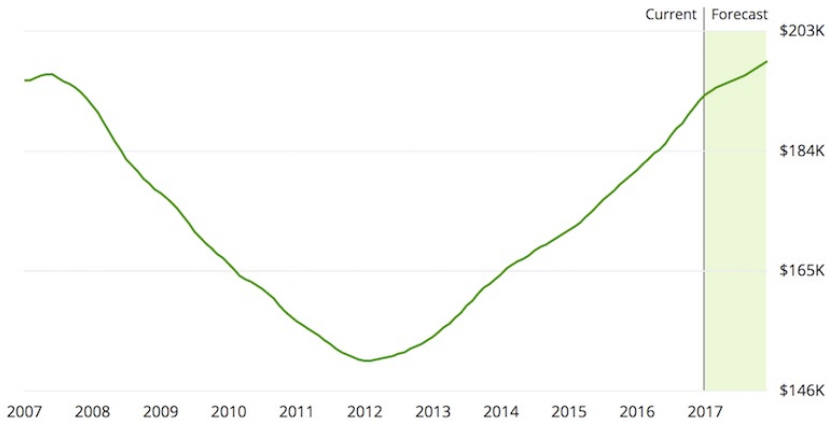



## Infuse Domain Knowledge

You can often engineer informative features by tapping into your (or others') expertise about the domain.

Try to think of specific information you might want to **isolate**. Here, you have a lot of "creative freedom."

Going back to our example with the real-estate dataset, let's say you remembered that the housing crisis occurred in the same timeframe...




Screenshot taken from *Zillow Home Values*

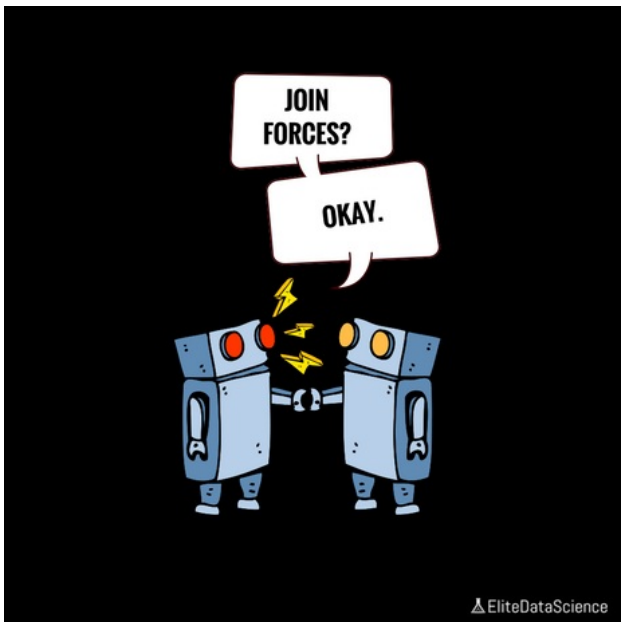
Well, if you suspect that prices would be affected, you could create an **indicator variable** for transactions during that period. Indicator variables are binary variables that can be either 0 or 1. They "indicate" if an observation meets a certain condition, and they are very useful for isolating key properties.

As you might suspect, "domain knowledge" is very broad and open-ended. At some point, you'll get stuck or exhaust your ideas.

That's where these next few steps come in. These are a few specific heuristics that can help spark more.

## Create Interaction Features

Joining forces.




The first of these heuristics is checking to see if you can create any **interaction features** that make sense. These are combinations of two or more features.

By the way, in some contexts, "interaction terms" must be products between two variables. In our context, interaction features can be **products**, **sums**, or **differences** between two features.

A general tip is to look at each pair of features and ask yourself, "could I combine this information in any way that might be even more useful?"

### Example (real-estate)

- Let's say we already had a feature called **'num_schools'**, i.e. the number of schools within 5 miles of a property.

- Let's say we also had the feature **'median_school'**, i.e. the median quality score of those schools.
- However, we might suspect that what's really important is **having many school options, but only if they are good.**
- Well, to capture that interaction, we could simple create a new feature **'school_score' = 'num_schools' x 'median_school'**

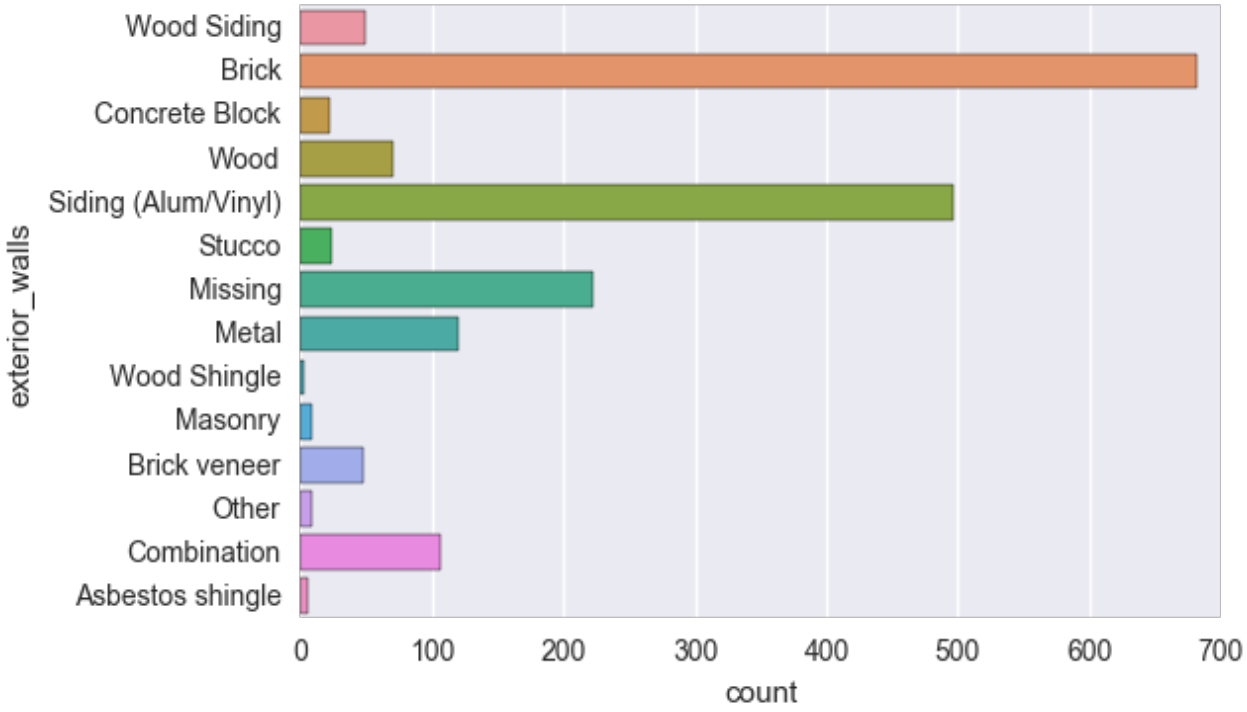
## Combine Sparse Classes

The next heuristic we'll consider is grouping sparse classes.

**Sparse classes** (in categorical features) are those that have very few total observations. They can be problematic for certain machine learning algorithms, causing models to be overfit.

- There's no formal rule of how many each class needs.
- It also depends on the size of your dataset and the number of other features you have.
- As a *rule of thumb*, we recommend combining classes until each one has at least ~50 observations. As with any "rule" of thumb, use this as a guideline (not actually as a *rule*).

Let's take a look at the real-estate example:




To begin, we can group **similar classes**. In the chart above, the *'exterior_walls'* feature has several classes that are quite similar.

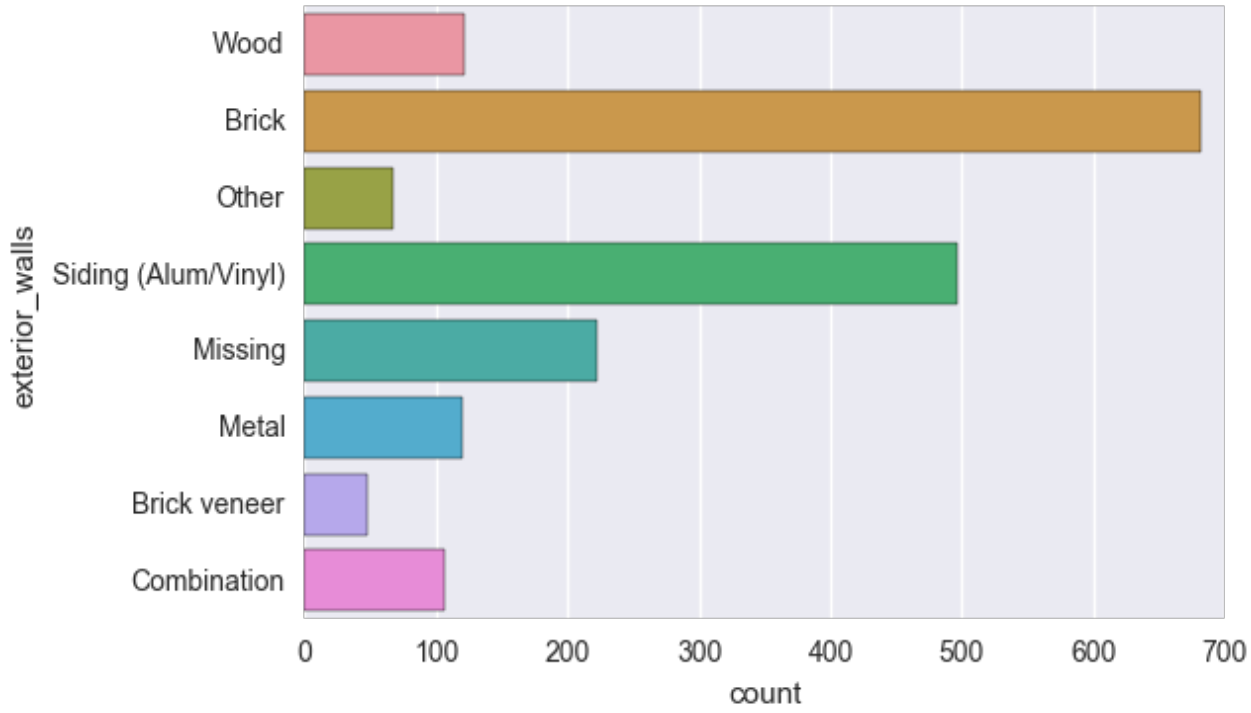
> We might want to group 'Wood Siding', 'Wood Shingle', and 'Wood' into a single

class. In fact, let's just label all of them as 'Wood'.

Next, we can group the remaining sparse classes into a single **'Other' class**, even if there's already an 'Other' class.

> We'd group 'Concrete Block', 'Stucco', 'Masonry', 'Other', and 'Asbestos shingle' into just 'Other'.

Here's how the class distributions look after combining similar and other classes:




After combining sparse classes, we have fewer unique classes, but each one has more observations.

Often, an **eyeball test** is enough to decide if you want to group certain classes together.

## Add Dummy Variables

Most machine learning algorithms cannot directly handle categorical features. Specifically, they cannot handle text values.

Therefore, we need to create dummy variables for our categorical features.

**Dummy variables** are a set of binary (0 or 1) variables that each represent a single class from a categorical feature.

The information you represent is exactly the same, but this numeric representation allows you to pass the technical requirements for algorithms.

In the example above, after grouping sparse classes, we were left with 8 classes, which translate to 8 dummy variables:

| Original class | Dummy variable | E.g. |
|---|---|---|
| Wood | exterior_walls_Wood | = 0 |
| Brick | exterior_walls_Brick | **= 1** |
| Other | exterior_walls_Other | = 0 |
| Siding (Alum/Vinyl) | exterior_walls_Siding (Alum/Vinyl) | = 0 |
| Missing | exterior_walls_Missing | = 0 |
| Metal | exterior_walls_Metal | = 0 |
| Brick veneer | exterior_walls_Brick veneer | = 0 |
| Combination | exterior_walls_Combination | = 0 |

(The 3rd column depicts an example for an observation with brick walls)

## Remove Unused Features

Finally, remove unused or redundant features from the dataset.

**Unused** features are those that don't make sense to pass into our machine learning algorithms. Examples include:

- ID columns
- Features that wouldn't be available at the time of prediction
- Other text descriptions

**Redundant** features would typically be those that have been replaced by other features that you've added during feature engineering.

## Checkpoint Quiz

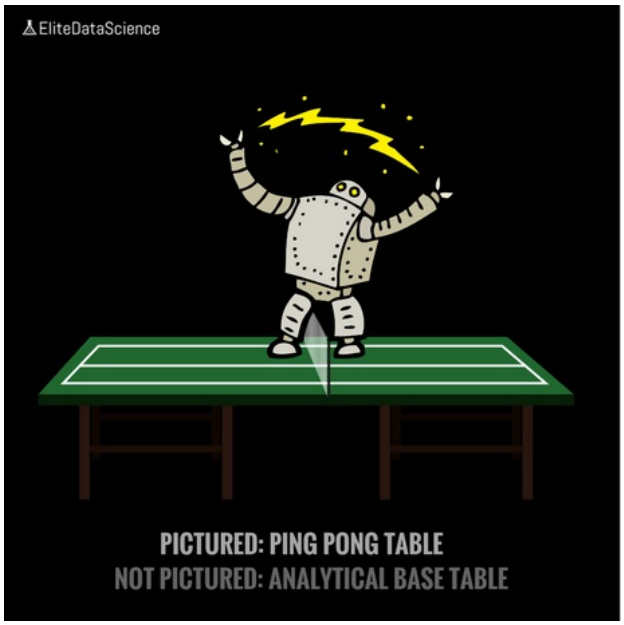
After completing Data Cleaning and Feature Engineering, you'll have transformed your raw dataset into an **analytical base table (ABT)**. We call it an "ABT" because it's what you'll be building your models on.

As a final tip: Not all of the features you engineer need to be winners. In fact, you'll often find that many of them don't improve your model. That's fine because one highly predictive feature makes up for 10 duds.

The key is choosing machine learning algorithms that can **automatically select the best features** among many options (**built-in feature selection**).

This will allow you to **avoid overfitting** your model despite providing many input features.

"Would someone please ask Alex to get off the ping-pong table? We're waiting to play!"




---

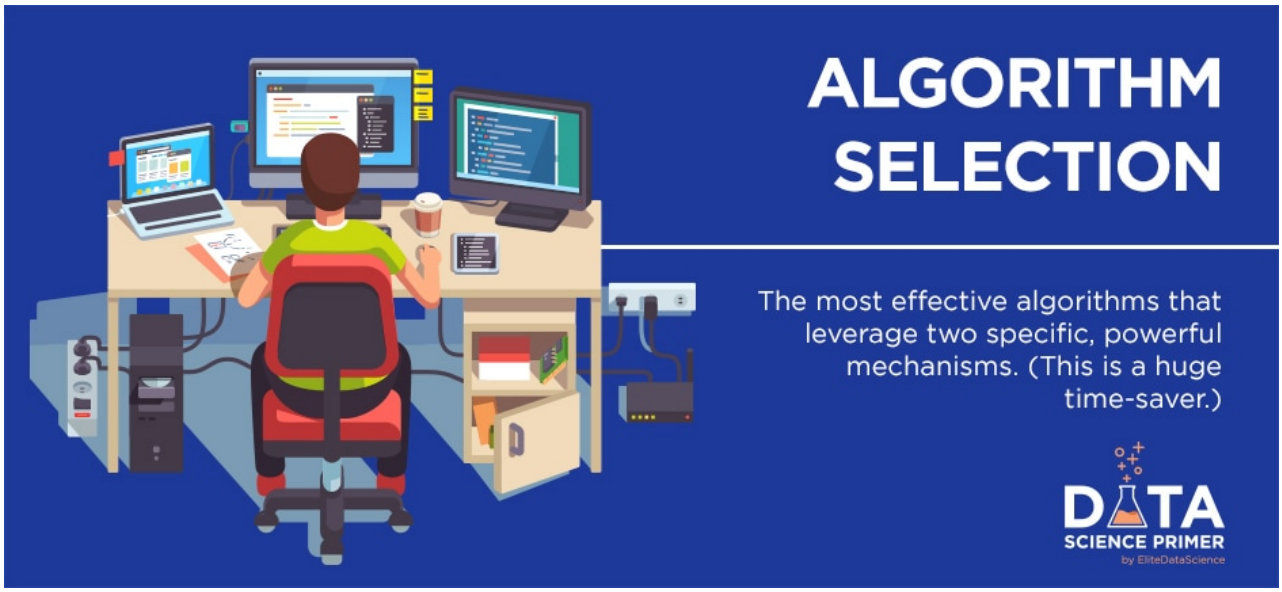
**Quiz time!**

- What are indicator variables and why are they useful?
- What are two criteria you can use to group sparse classes?
- In a set of dummy variables created from the same feature, would there ever be multiple variables with value 1 (per observation)?
- In our real-estate example, what would be the values for the *'exterior_walls'* dummy variables if a property had metal walls?

---

# Algorithm Selection

elitedatascience.com/algorithm-selection




Welcome to our 7-part mini-course on data science and applied machine learning!

In the previous chapter, you learned several different heuristics for effective feature engineering. Those included tapping into domain knowledge and grouping sparse classes.

This guide will explain **algorithm selection** for machine learning.

However, rather than bombarding you with options, we're going to jump straight to best practices.

We'll introduce two powerful mechanisms in modern algorithms: **regularization** and **ensembles**. As you'll see, these mechanisms "fix" some fatal flaws in older methods, which has lead to their popularity.

Let's get started!

## How to Pick ML Algorithms

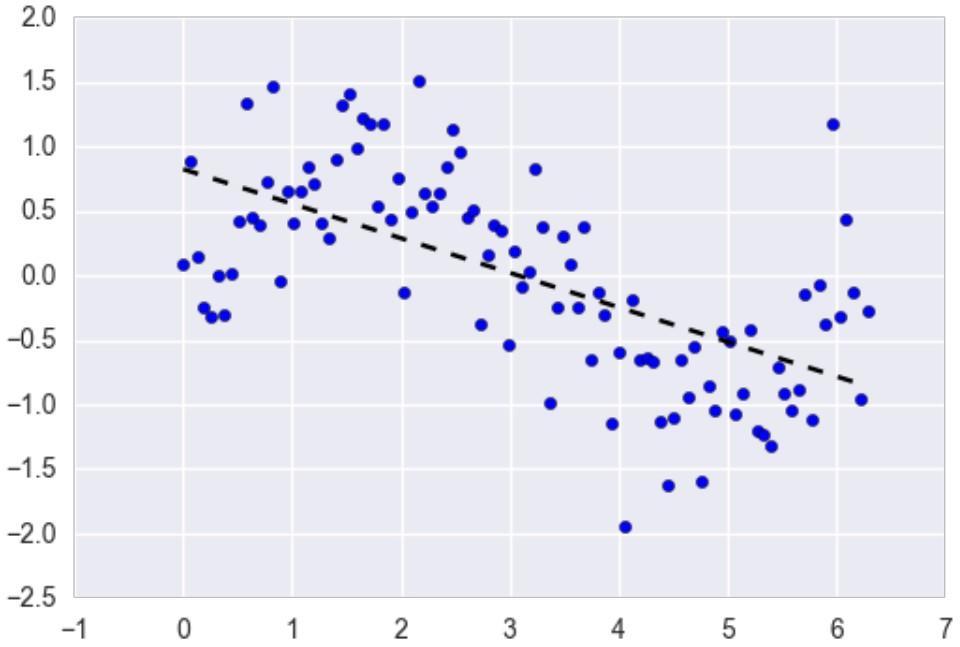
In this lesson, we'll introduce 5 very effective machine learning algorithms for **regression** tasks. They each have classification counterparts as well.

And yes, just 5 for now. Instead of giving you a long list of algorithms, our goal is to explain a few essential concepts (e.g. *regularization, ensembling, automatic feature selection*) that will teach you ***why*** some algorithms tend to perform better than others.

In applied machine learning, individual algorithms should be swapped in and out depending on which performs best for the problem and the dataset. Therefore, we will focus on **intuition** and **practical benefits** over math and theory.

## Why Linear Regression is Flawed

To introduce the reasoning for some of the advanced algorithms, let's start by discussing basic linear regression. Linear regression models are very common, yet deeply flawed.




Simple linear regression models fit a "straight line" (technically a *hyperplane* depending on the number of features, but it's the same idea). In practice, they rarely perform well. We actually recommend skipping them for most machine learning problems.

Their main advantage is that they are easy to interpret and understand. However, our goal is not to study the data and write a research report. Our goal is to build a model that can make accurate predictions.

In this regard, simple linear regression suffers from two major flaws:

1. **It's prone to overfit with many input features.**
2. **It cannot easily express non-linear relationships.**

Let's take a look at how we can address the first flaw.

## Regularization in Machine Learning

This is the first "advanced" tactic for improving model performance. It’s considered pretty "advanced" in many ML courses, but it’s really pretty easy to understand and implement.

The first flaw of linear models is that they are prone to be overfit with many input features.

Let's take an extreme example to illustrate why this happens:


- Let's say you have 100 observations in your training dataset.
- Let's say you also have 100 features.
- If you fit a linear regression model with all of those 100 features, you can perfectly "memorize" the training set.
- Each **coefficient** would simply **memorize one observation**. This model would have perfect accuracy on the training data, but perform poorly on unseen data.
- It hasn't learned the true underlying patterns; it has only memorized the noise in the training data.

**Regularization** is a technique used to prevent overfitting by *artificially penalizing model coefficients*.

- It can discourage large coefficients (by dampening them).
- It can also remove features entirely (by setting their coefficients to 0).
- The "strength" of the penalty is **tunable**. (More on this tomorrow...)

## Regularized Regression Algos

There are 3 common types of regularized linear regression algorithms.

### Lasso Regression

Lasso, or LASSO, stands for **L**east **A**bsolute **S**hrinkage and **S**election **O**perator.

- Lasso regression penalizes the **absolute size** of coefficients.
- Practically, this leads to coefficients that can be exactly 0.
- Thus, Lasso offers **automatic feature selection** because it can completely remove some features.

- Remember, the "strength" of the penalty should be tuned.
- A stronger penalty leads to *more* coefficients pushed to zero.

### Ridge Regression

Ridge stands **R**eally **I**ntense **D**angerous **G**rapefruit **E**ating (just kidding... it's just ridge).

- Ridge regression penalizes the **squared size** of coefficients.
- Practically, this leads to smaller coefficients, but it doesn't force them to 0.
- In other words, Ridge offers **feature shrinkage**.
- Again, the "strength" of the penalty should be tuned.
- A stronger penalty leads to coefficients pushed *closer* to zero.

### Elastic-Net

Elastic-Net is a compromise between Lasso and Ridge.

- Elastic-Net penalizes a **mix** of both absolute and squared size.
- The **ratio** of the two penalty types should be tuned.
- The overall strength should also be tuned.

Oh and in case you're wondering, there's no "best" type of penalty. It really depends on the dataset and the problem. We recommend trying different algorithms that use a range of penalty strengths as part of the tuning process, which we'll cover in detail tomorrow.

## Decision Tree Algos

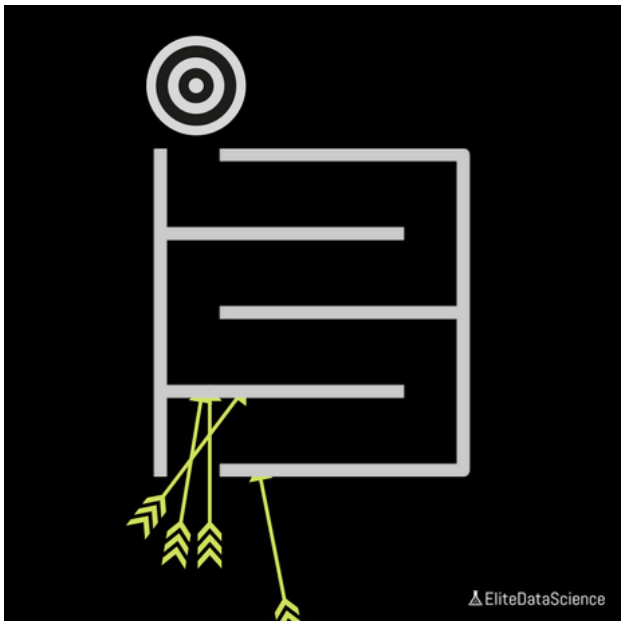
Awesome, we've just seen 3 algorithms that can protect linear regression from overfitting. But if you remember, linear regression suffers from *two* main flaws:

1. It's prone to overfit with many input features.

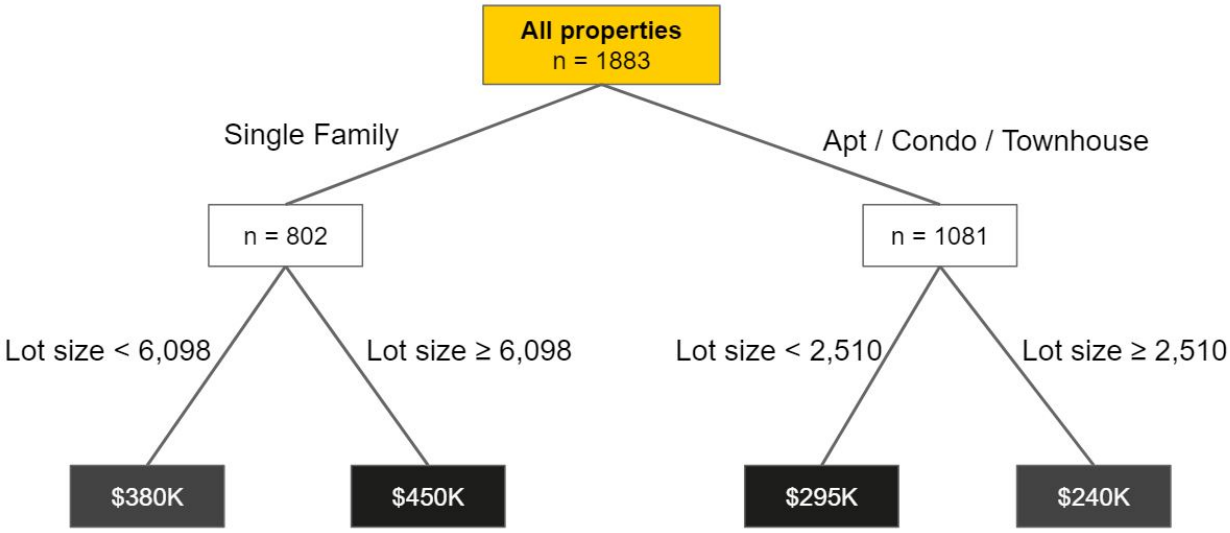
2. **It cannot easily express non-linear relationships.**

How can we address the second flaw?

We need a different strategy... Well, we need to move away from linear models to do so.... we need to bring in a new category of algorithms.




Decision trees model data as a "tree" of hierarchical branches. They make branches until they reach "leaves" that represent predictions.




Due to their **branching structure**, decision trees can easily model nonlinear relationships.

- For example, let's say for Single Family homes, larger lots command higher prices.
- However, let's say for Apartments, *smaller* lots command higher prices (i.e. it's a proxy for urban / rural).
- This *reversal of correlation* is difficult for linear models to capture unless you explicitly add an interaction term (i.e. you can anticipate it ahead of time).
- On the other hand, decision trees can capture this relationship naturally.

Unfortunately, decision trees suffer from a major flaw as well. If you allow them to grow limitlessly, they can completely "memorize" the training data, just from creating more and more and more branches.

**As a result, individual unconstrained decision trees are very prone to being overfit.**

So, how can we take advantage of the flexibility of decision trees while preventing them from overfitting the training data?

## Tree Ensembles

Ensembles are machine learning methods for combining predictions from multiple separate models. There are a few different methods for ensembling, but the two most common are:

### Bagging

**Bagging** attempts to *reduce the chance overfitting complex models*.

- It trains a large number of "strong" learners in parallel.
- A **strong learner** is a model that's relatively *unconstrained*.
- Bagging then combines all the strong learners together in order to "smooth out" their predictions.

### Boosting

**Boosting** attempts to *improve the predictive flexibility of simple models*.

- It trains a large number of "weak" learners in sequence.
- A **weak learner** is a *constrained* model (i.e. you could limit the max depth of each decision tree).
- Each one in the sequence focuses on learning from the mistakes of the one before it.
- Boosting then combines all the weak learners into a single strong learner.

While bagging and boosting are both ensemble methods, they approach the problem from opposite directions. Bagging uses complex base models and tries to "smooth out" their predictions, while boosting uses simple base models and tries to "boost" their aggregate complexity.

Ensembling is a general term, but when the **base models** are decision trees, they have special names: random forests and boosted trees!

## Random forests

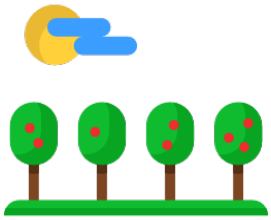
Random forests train a **large number of "strong" decision trees** and combine their predictions through bagging.

In addition, there are two sources of "randomness" for random forests:

1. Each tree is only allowed to choose from a random subset of features to split on (leading to **feature selection**).
2. Each tree is only trained on a random subset of observations (a process called **resampling**).

In practice, random forests tend to *perform very well right out of the box*.

- They often beat many other models that take up to weeks to develop.
- They are the perfect **"swiss-army-knife"** algorithm that almost always gets good results.
- They don't have many complicated parameters to tune.



## Boosted trees

Boosted trees train a **sequence of "weak", constrained decision trees** and combine their predictions through boosting.

- Each tree is allowed a **maximum depth**, which should be tuned.
- Each tree in the sequence tries to correct the prediction errors of the one before it.

In practice, boosted trees tend to have the *highest performance ceilings*.

- They often beat many other types of models **after proper tuning**.
- They are more complicated to tune than random forests.

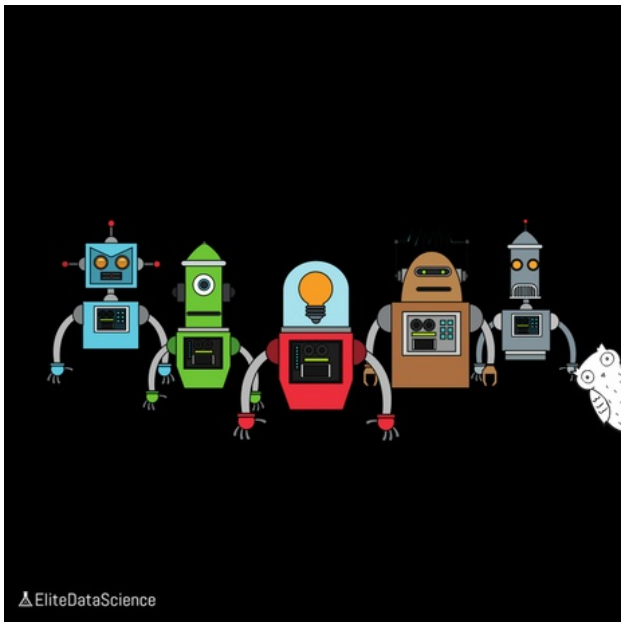
# Checkpoint Quiz

Whew, that was a lot! If you need to, feel free to let it sink in a bit and then re-read the lesson.

**Key takeaway:** The most effective algorithms typically offer a combination of regularization, automatic feature selection, ability to express nonlinear relationships, and/or ensembling. Those algorithms include:

1. Lasso regression
2. Ridge regression
3. Elastic-Net
4. Random forest
5. Boosted tree

Despite our best efforts, Oliver the Owl still managed to photobomb the "Power Rangers Group Photo."




---

**Here's a quiz to make sure you caught everything:**

- What are the two biggest flaws of linear regression?
- How can you address the first flaw? (Which mechanism, and which algorithms?)
- How can you address the second flaw? (Which mechanism, and which algorithms?)
- What are two types of regularization penalty, and what do they do in practice?
- What are two methods for ensembling and how do they work?

---

# Model Training

elitedatascience.com/model-training


Welcome to our 7-part mini-course on data science and applied machine learning!

In the previous chapter, we introduced 5 effective ML algorithms. They tap into the powerful mechanisms of regularization and ensembles.

In this guide, we will take you step-by-step through the **model training** process.

Since we've already done the hard part, actually fitting (a.k.a. training) our model will be fairly straightforward.

There are a few key techniques that we'll discuss, and these have become widely-accepted **best practices** in the field.

Again, this mini-course is meant to be a gentle introduction to data science and machine learning, so we won't get into the nitty gritty yet. We'll save that for the next steps.

## How to Train ML Models

At last, it's time to build our models!

It might seem like it took us a while to get here, but professional data scientists actually spend the bulk of their time on the steps leading up to this one:

1. Exploring the data.
2. Cleaning the data.
3. Engineering new features.

Again, that's because **better data beats fancier algorithms**.

In this lesson, you'll learn how to set up the entire modeling process to maximize performance while **safeguarding against overfitting**. We will swap algorithms in and out and automatically find the best parameters for each one.

"Model training, at last!"


## Split Dataset

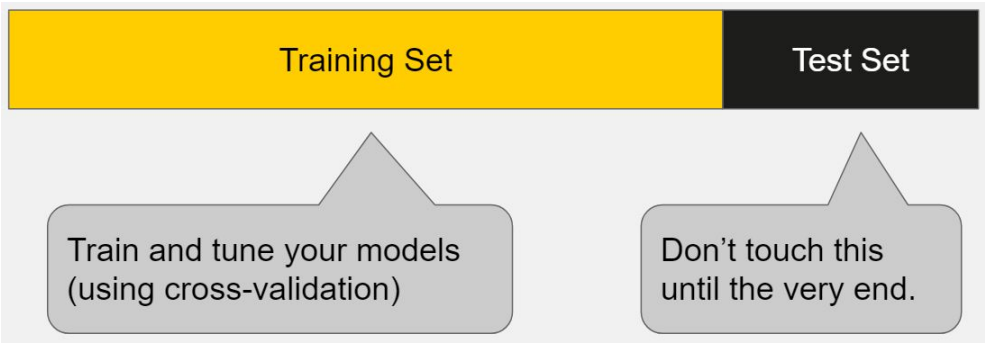
Let's start with a crucial but sometimes overlooked step: Spending your data.

Think of your data as a **limited resource**.

- You can spend some of it to train your model (i.e. feed it to the algorithm).
- You can spend some of it to evaluate (test) your model.
- But you can't reuse the same data for both!

If you evaluate your model on the same data you used to train it, your model could be very overfit and you wouldn't even know! A model should be judged on its ability to predict *new, unseen data*.

Therefore, you should have separate training and test subsets of your dataset.




**Training sets** are used to fit and tune your models. **Test sets** are put aside as "unseen" data to evaluate your models.

- You should always split your data *before* doing anything else.
- This is the best way to get reliable estimates of your models' performance.
- After splitting your data, **don't touch your test set** until you're ready to choose your final model!

Comparing test vs. training performance allows us to avoid overfitting… **If the model performs very well on the training data but poorly on the test data, then it's overfit.**

## What are Hyperparameters?

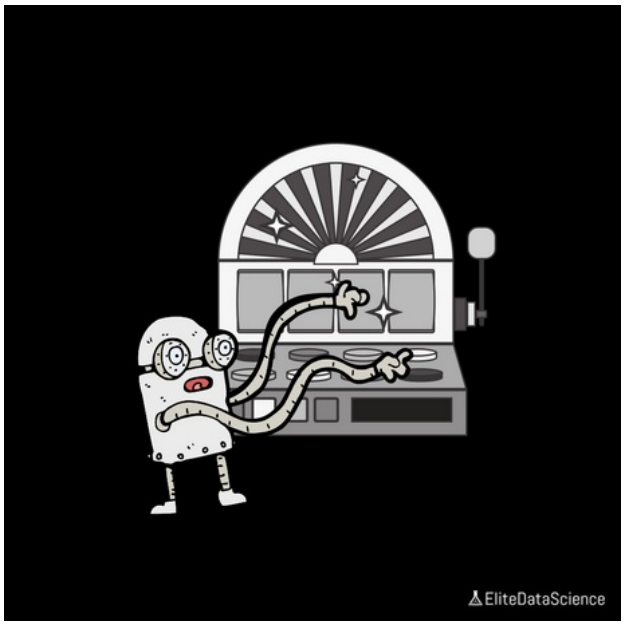
So far, we've been casually talking about "tuning" models, but now it's time to treat the topic more formally.

When we talk of tuning models, we specifically mean tuning **hyperparameters**.

There are two types of parameters in machine learning algorithms.

The key distinction is that model parameters can be learned directly from the training data while hyperparameters cannot.

"Don't mind me. Just ~~randomly pressing buttons~~ carefully tuning parameters."




### Model parameters

Model parameters are learned attributes that define *individual models*.

- e.g. regression coefficients
- e.g. decision tree split locations
- They can be **learned directly** from the training data

### Hyperparameters

Hyperparameters express "higher-level" *structural settings* for algorithms.

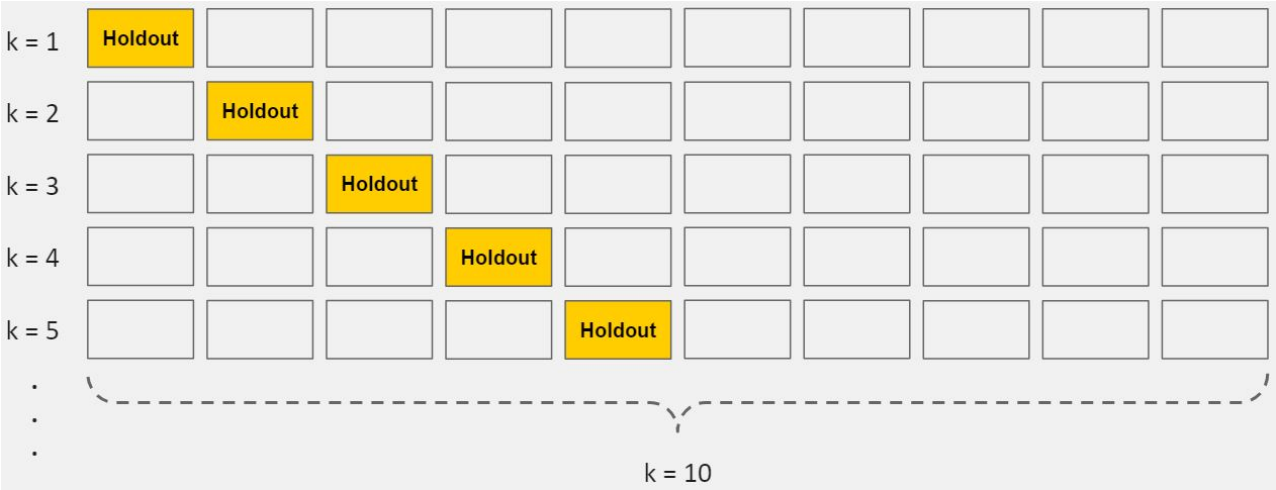
- e.g. strength of the penalty used in regularized regression
- e.g. the number of trees to include in a random forest
- They are **decided** before fitting the model because they can't be learned from the data

## What is Cross-Validation?

Next, it's time to introduce a concept that will help us tune our models: cross-validation.

Cross-validation is a method for getting a *reliable estimate of model performance using only your training data*.

There are several ways to cross-validate. The most common one, **10-fold cross-validation**, breaks your training data into 10 equal parts (a.k.a. folds), essentially creating 10 miniature train/test splits.




These are the steps for 10-fold cross-validation:

1. Split your data into 10 equal parts, or "folds".
2. Train your model on 9 folds (e.g. the first 9 folds).
3. Evaluate it on the 1 remaining "hold-out" fold.
4. Perform steps (2) and (3) 10 times, each time holding out a different fold.
5. Average the performance across all 10 hold-out folds.

The average performance across the 10 hold-out folds is your final performance estimate, also called your **cross-validated score**. Because you created 10 mini train/test splits, this score is usually pretty reliable.

## Fit and Tune Models

Now that we've split our dataset into training and test sets, and we've learned about hyperparameters and cross-validation, we're ready fit and tune our models.

Basically, all we need to do is perform the entire cross-validation loop detailed above on each **set of hyperparameter values** we'd like to try.

The high-level pseudo-code looks like this:

```
For each algorithm (i.e. regularized regression, random forest, etc.):
  For each set of hyperparameter values to try:
    Perform cross-validation using the training set.
    Calculate cross-validated score.
```

At the end of this process, you will have a cross-validated score for each set of hyperparameter values... for each algorithm.

For example:

Then, we'll pick the best set of hyperparameters *within each algorithm*.

```
For each algorithm:
  Keep the set of hyperparameter
values with best cross-validated score.
  Re-train the algorithm on the entire
training set (without cross-validation).
```

It's kinda like the Hunger Games... each algorithm sends its own "representatives" (i.e. model trained on the best set of hyperparameter values) to the final selection, which is coming up next...

| Elastic-Net | | |
|---|---|---|
| Penalty Ratio | Penalty Strength | CV-Score |
| 75/25 | 0.01 | 0.63 |
| 75/25 | 0.05 | 0.64 |
| 75/25 | 0.10 | **0.67** |
| 50/50 | 0.01 | 0.62 |
| 50/50 | 0.05 | 0.63 |
| 50/50 | 0.10 | 0.66 |
| . | . | . |
| . | . | . |

## Select Winning Model

By now, you'll have 1 "best" model *for each algorithm* that has been tuned through cross-validation. Most importantly, you've only used the training data so far.

Now it's time to evaluate each model and pick the best one, a la Hunger Games style.

"I volunteer as tribute!"

Because you've saved your **test set** as a truly unseen dataset, you can now use it get a reliable estimate of each models' performance.

There are a variety of **performance metrics** you could choose from. We won't spend too much time on them here, but in general:

- For regression tasks, we recommend **Mean Squared Error (MSE)** or **Mean**

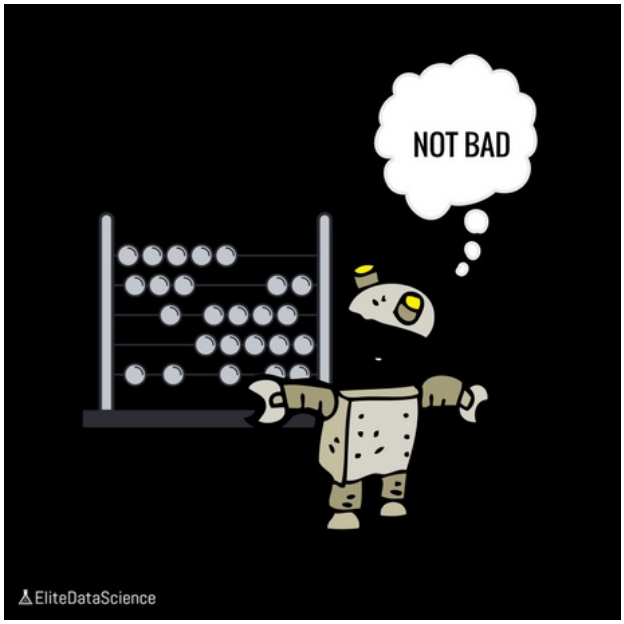
**Absolute Error (MAE)**. *(Lower values are better)*

- For classification tasks, we recommend **Area Under ROC Curve (AUROC)**. *(Higher values are better)*

The process is very straightforward:

1. For each of your models, make predictions on your test set.
2. Calculate performance metrics using those predictions and the "ground truth" target variable from the test set.

Doing calculations on an abacus #TBT #math #retro #oldschool #oldskool #oldscool




Finally, use these questions to help you pick the winning model:

- Which model had the best performance on the test set? **(performance)**
- Does it perform well across various performance metrics? **(robustness)**
- Did it also have (one of) the best cross-validated scores from the training set? **(consistency)**
- Does it solve the original business problem? **(win condition)**

## Checkpoint Quiz

**Ready for the quiz?**

- (Hopefully a freebie) Pick one: better data or fancier algorithms.
- When should you split your dataset into training and test sets, and why?
- What's the key difference between model parameters and hyperparameters?
- Explain how cross-validation helps you "tune" your models?

---

# WTF is the Bias-Variance Tradeoff? (Infographic)

elitedatascience.com/bias-variance-tradeoff

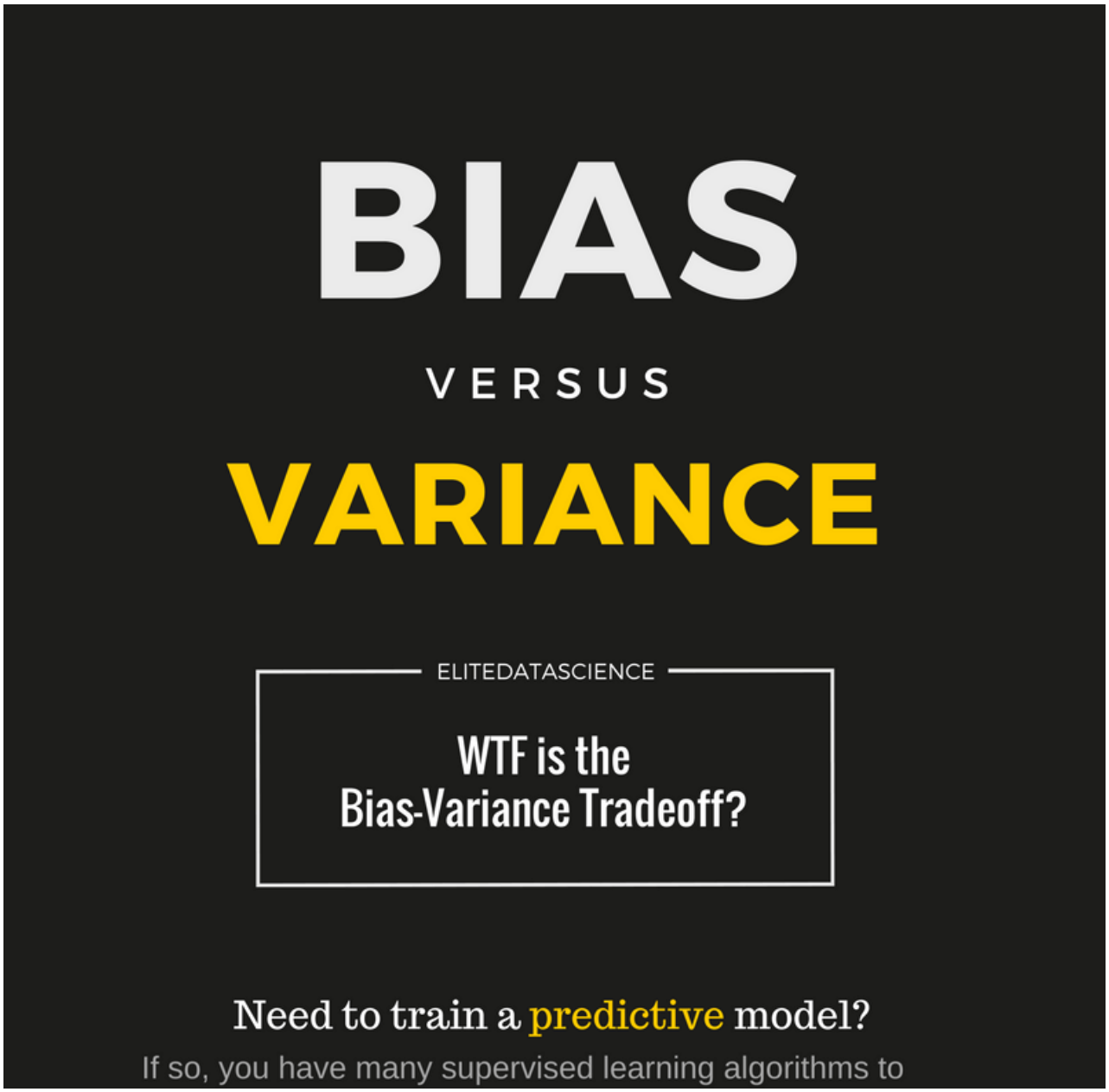
June 6, 2017

Overheard after class: “doesn’t the Bias-Variance Tradeoff sound like the name of a treaty from a history documentary?”

Ok, that’s fair… but it’s *also* one of the most important concepts to understand for supervised machine learning and predictive modeling.

Unfortunately, because it’s often taught through dense math formulas, it’s earned a tough reputation.

But as you’ll see in this guide, it’s not that bad. In fact, the Bias-Variance Tradeoff has simple, *practical* implications around model complexity, over-fitting, and under-fitting.



choose from! i.e. Regressions, Decision Trees, Neural Networks, SVM's, etc...

## Well, those algos are distinct...

...in many ways. But a major difference is their amount of Bias vs. Variance, which are two types of prediction error.

**Bias** occurs when an algo has *limited flexibility* to learn the true signal from a dataset.

**Variance** refers to an algo's *sensitivity* to specific sets of training data.

## Wait, "sensitivity" to training data?

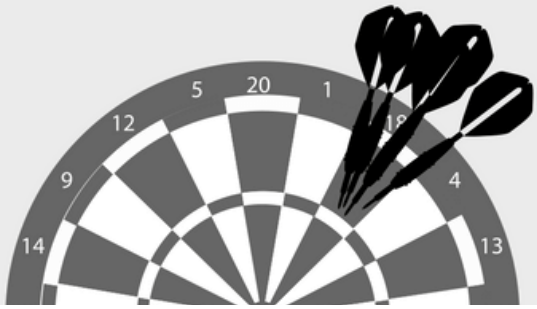
Yes. It's much easier to wrap your head around this concept if you think of algorithms not as one-time methods for training individual models, but instead as repeatable processes.

## Ok, let's do a thought experiment...

Imagine you've collected 5 different training datasets for the same problem. Now, imagine using one algorithm to train 5 models (one for each training set).

## Here's what those 5 models tell you about your chosen algorithm:

**High Bias**
Low Variance




**High Variance**
Low Bias



**High bias**, low variance algorithms train models that are consistent, but inaccurate *on average*.

**High variance**, low bias algorithms train models that are accurate *on average*, but inconsistent.

## But why is there a tradeoff?

Low variance algos tend to be **less complex**, with simple or rigid underlying structure.

- e.g. Regression
- e.g. Naive Bayes
- *Linear algos*
- *Parametric algos*

Low bias algos tend to be **more complex**, with flexible underlying structure.

- e.g. Decision trees
- e.g. Nearest neighbors
- *Non-linear algos*
- *Non-parametric algos*

## Within each algo family, there's a tradeoff too...

For example, regression can be **regularized** to further reduce complexity.

For example, decision trees can be **pruned** to reduce complexity.

## That's why a solid model training methodology is key.

Algos that are not complex enough produce **underfit** models that can't learn the signal from the data

Algo that are too complex produce **overfit** models that memorize the noise instead of the signal

signal from the data.

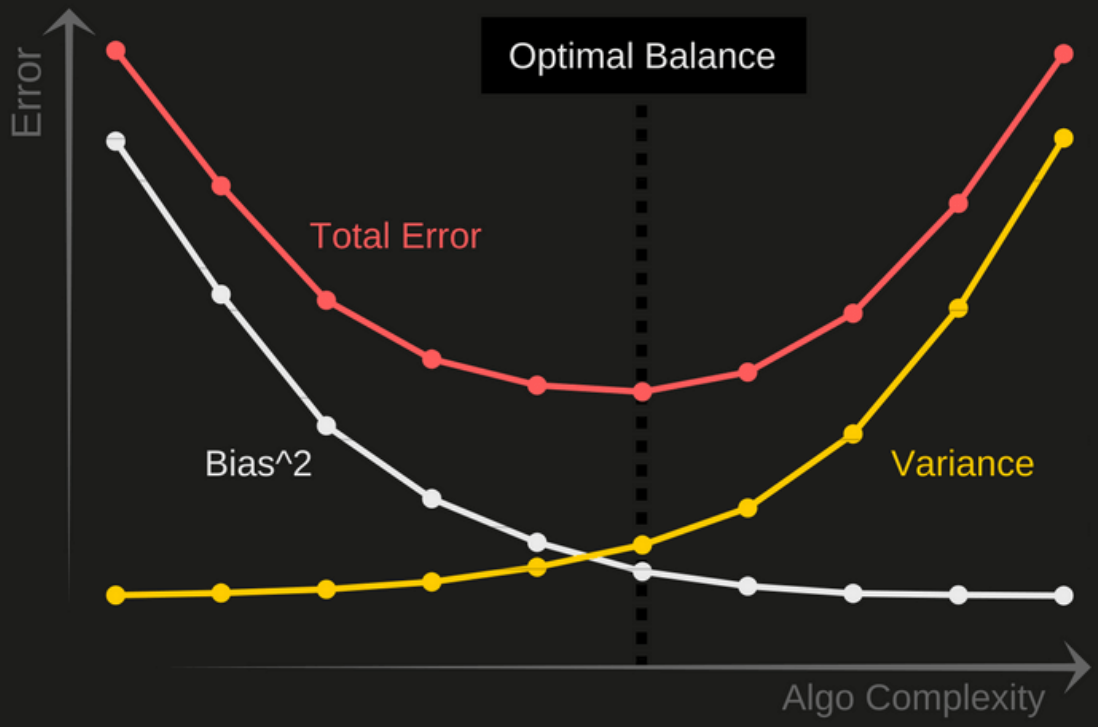
the signal.

To get good predictions, you'll need to find a balance of Bias and Variance that minimizes total error.

## Total error breaks down as:

Total Error = Bias^2 + Variance + Irreducible Error

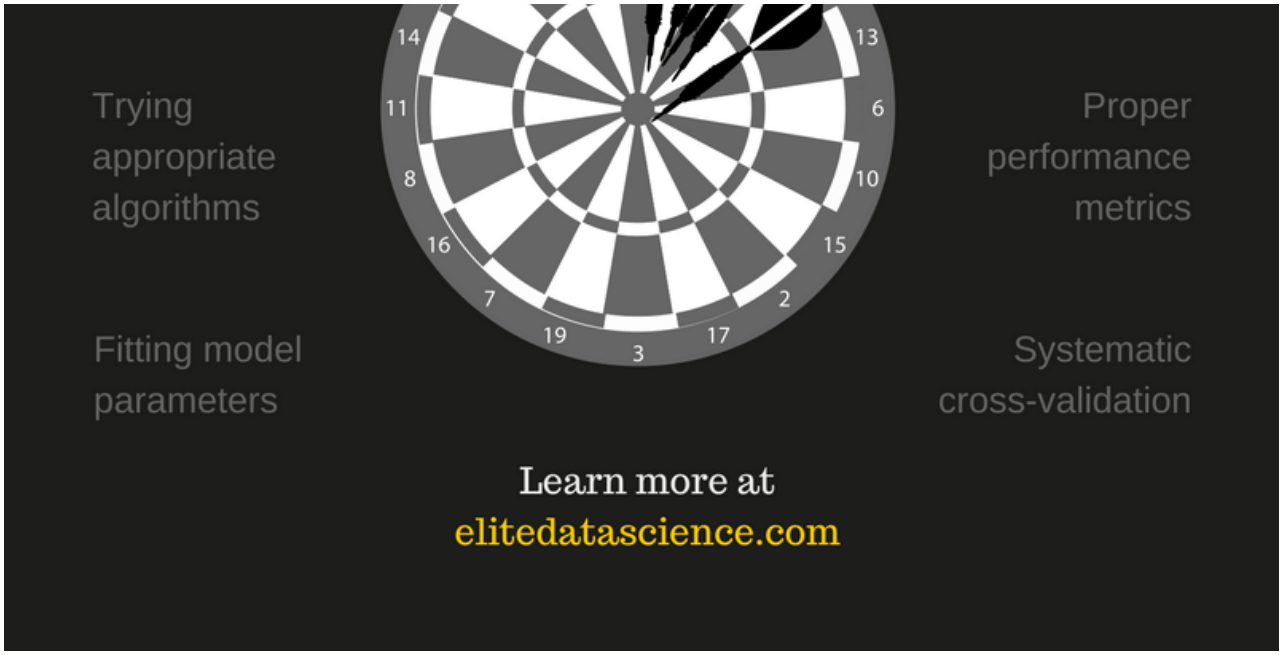
(Irreducible error is "noise" that can't be reduced by algorithms. It can sometimes be reduced by better data cleaning)




## A proper ML workflow finds that optimal balance!

e.g.
Separate training and test sets

e.g.
Tuning impactful hyperparameters




**Share this Infographic on the Bias-Variance Tradeoff:**

## Here's our take on the insights from the infographic.

### Supervised Learning

The Bias-Variance Tradeoff is relevant for supervised machine learning - specifically for predictive modeling. It's a way to diagnose the performance of an algorithm by breaking down its prediction error.

In machine learning, an **algorithm** is simply a repeatable process used to train a **model** from a given set of **training data**.

- You have many algorithms to choose from, such as Linear Regression, Decision Trees, Neural Networks, SVM's, and so on.
- You can learn more about them in our practical tour through modern machine learning algorithms.

As you might imagine, each of those algorithms behave very differently, each shining in different situations. One of the key distinctions is how much bias and variance they produce.

There are 3 types of prediction error: bias, variance, and irreducible error.

Irreducible error is also known as "noise," and it can't be reduced by your choice in algorithm. It typically comes from inherent randomness, a mis-framed problem, or an incomplete feature set.

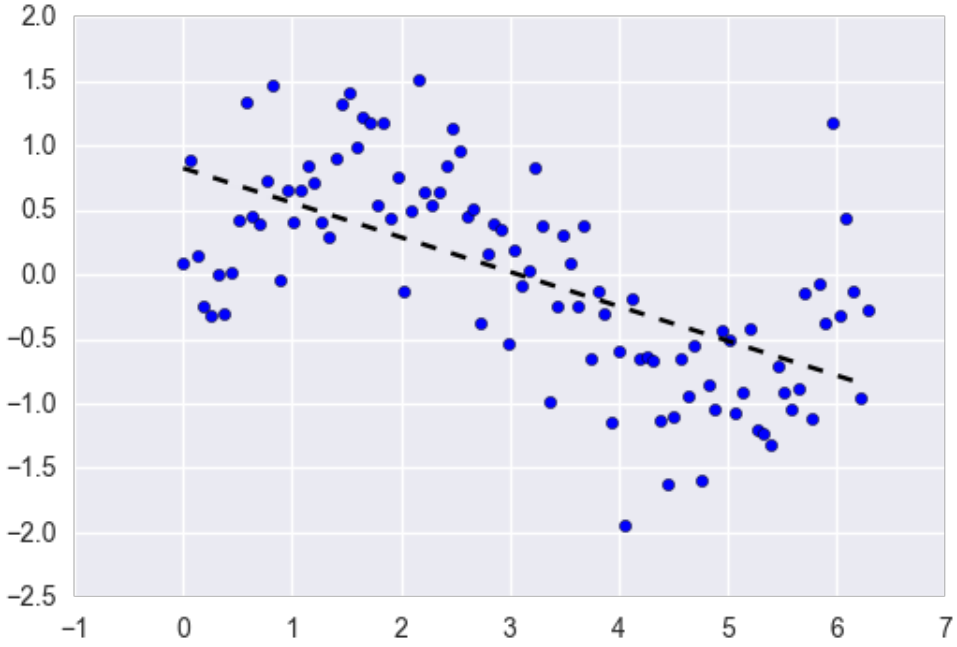
The other two types of errors, however, can be reduced because they stem from your algorithm choice.

## Error from Bias

Bias is the difference between your model's expected predictions and the true values.

That might sound strange because shouldn't you "expect" your predictions to be close to the true values? Well, it's not always that easy because some algorithms are simply too rigid to learn complex signals from the dataset.

Imagine fitting a linear regression to a dataset that has a non-linear pattern:



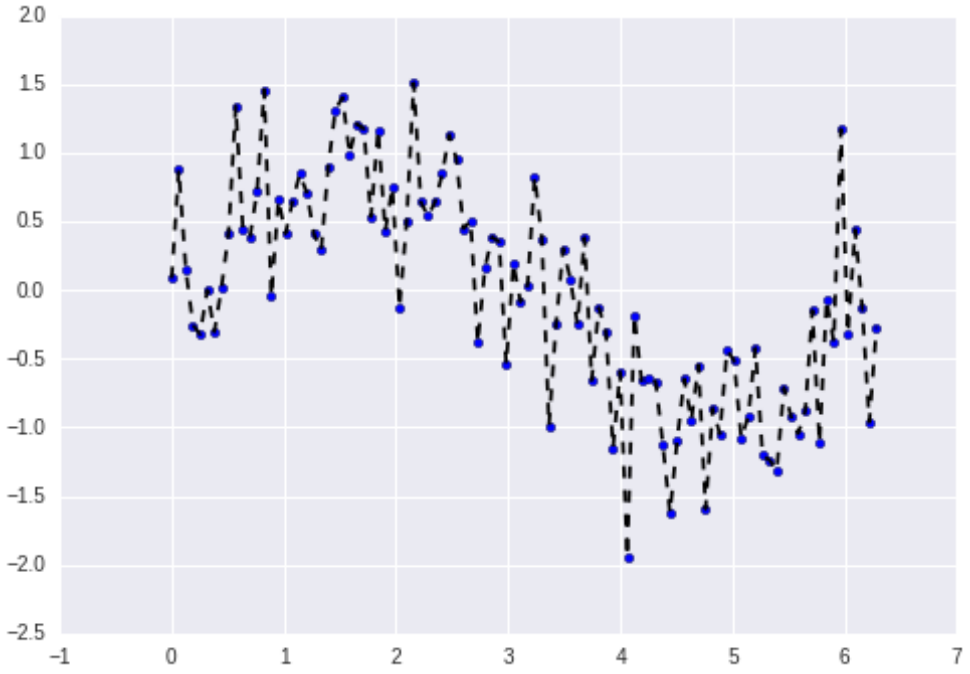

No matter how many more observations you collect, a linear regression won't be able to model the curves in that data! This is known as **under-fitting**.

## Error from Variance

Variance refers to your algorithm's sensitivity to specific sets of training data.

High variance algorithms will produce drastically different models depending on the training set.

For example, imagine an algorithm that fits a completely unconstrained, super-flexible model to the same dataset from above:




As you can see, this unconstrained model has basically memorized the training set, including all of the noise. This is known as **over-fitting**.

## The Bias-Variance Tradeoff

It's much easier to wrap your head around these concept if you think of algorithms not as one-time methods for training individual models, but instead as repeatable processes.

**Let's do a thought experiment:**

1. Imagine you've collected 5 *different* training sets for the *same* problem.
2. Now imagine using one algorithm to train 5 models, one for each of your training sets.
3. Bias vs. variance refers to the accuracy vs. consistency of the models trained by your algorithm.

We can diagnose them as follows.


Low variance (high bias) algorithms tend to be **less complex**, with simple or rigid underlying structure.

- They train models that are consistent, but inaccurate *on average*.
- These include linear or parametric algorithms such as regression and naive Bayes.

On the other hand, low bias (high variance) algorithms tend to be **more complex**, with flexible underlying structure.

- They train models that are accurate *on average*, but inconsistent.
- These include non-linear or non-parametric algorithms such as decision trees and nearest neighbors.

This **tradeoff in complexity** is why there's a tradeoff in bias and variance - an algorithm cannot simultaneously be more complex and less complex.

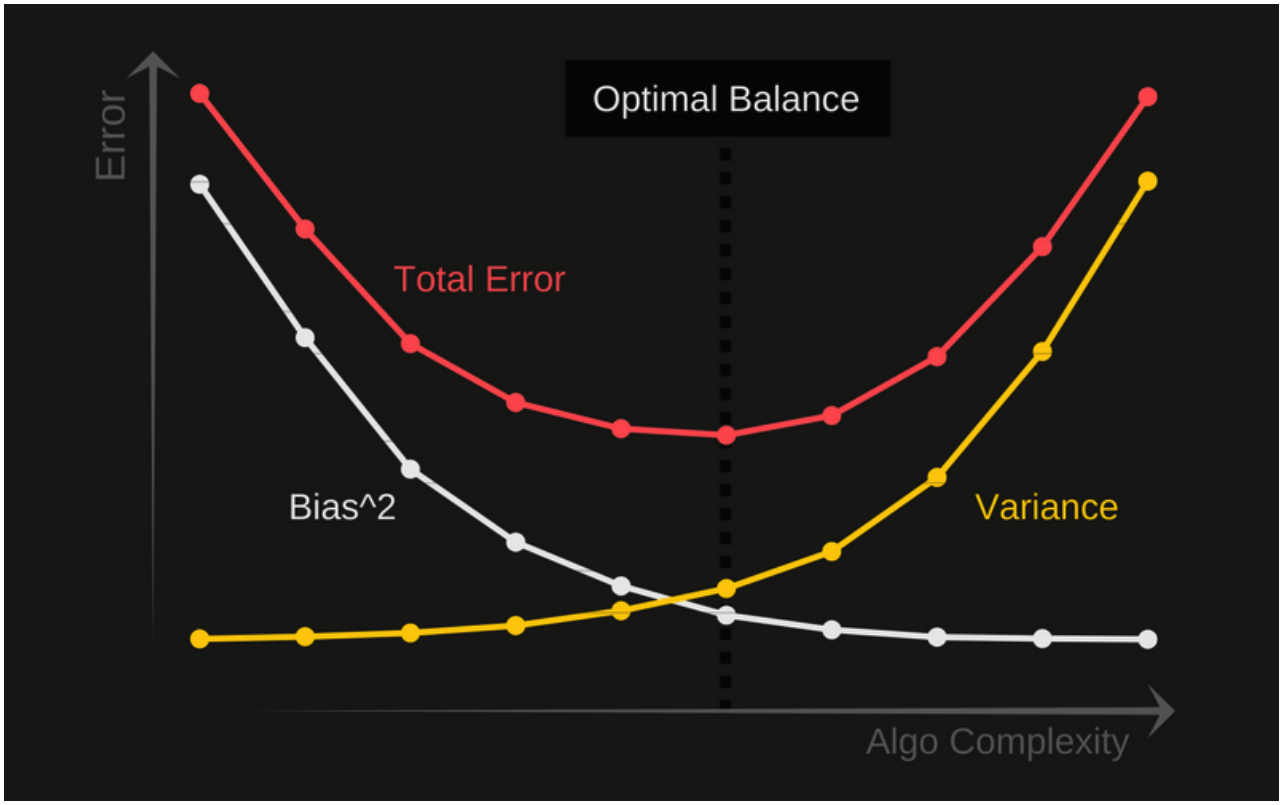
**Note: For certain problems, it's possible for some algorithms to have less of both errors than others. For example, ensemble methods (i.e. Random Forests) often perform better than other algorithms in practice. Our recommendation is to always try multiple reasonable algorithms for each problem.*

## Total Error

To build a good predictive model, you'll need to find a balance between bias and variance that minimizes the total error.

**Total Error = Bias^2 + Variance + Irreducible Error**

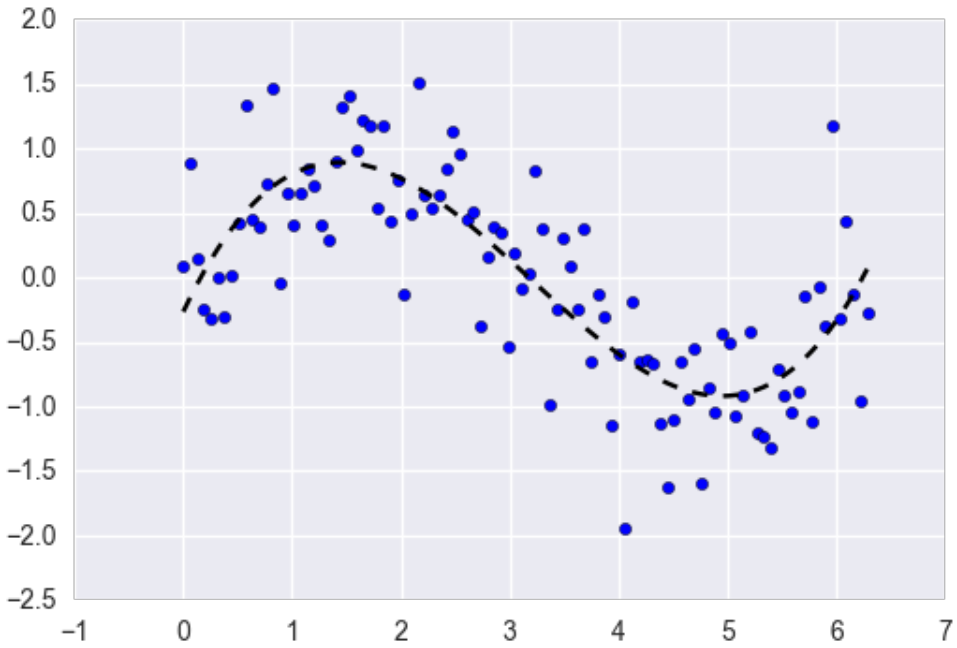
Machine learning processes find that optimal balance:




A proper machine learning workflow includes:

- Separate training and test sets
- Trying appropriate algorithms (No Free Lunch)
- Fitting model parameters
- Tuning impactful hyperparameters
- Proper performance metrics
- Systematic cross-validation

Finally, as you might have already concluded, an optimal balance of bias and variance leads to a model that is neither overfit nor underfit:



This is the ultimate goal of supervised machine learning - to isolate the **signal** from the dataset while ignoring the noise!

# Modern Machine Learning Algorithms: Strengths and Weaknesses

elitedatascience.com/machine-learning-algorithms

May 16, 2017

In this guide, we'll take a practical, concise tour through modern machine learning algorithms. While other such lists exist, they don't really explain the practical tradeoffs of each algorithm, which we hope to do here. We'll discuss the advantages and disadvantages of each algorithm based on our experience.

Categorizing machine learning algorithms is tricky, and there are several reasonable approaches; they can be grouped into generative/discriminative, parametric/non-parametric, supervised/unsupervised, and so on.

For example, Scikit-Learn's documentation page groups algorithms by their **learning mechanism.** This produces categories such as:

- Generalized linear models
- Support vector machines
- Nearest neighbors
- Decision trees
- Neural networks
- And so on...

However, from our experience, this isn't always the most practical way to group algorithms. That's because for applied machine learning, you're usually not thinking, *"boy do I want to train a support vector machine today!"*

Instead, you usually have an end goal in mind, such as predicting an outcome or classifying your observations.

Therefore, we want to introduce another approach to categorizing algorithms, which is by **machine learning task.**

## No Free Lunch

In machine learning, there's something called the "No Free Lunch" theorem. In a nutshell, it states that no one algorithm works best for every problem, and it's especially relevant for supervised learning (i.e. predictive modeling).

For example, you can't say that neural networks are always better than decision trees or vice-versa. There are many factors at play, such as the size and structure of your dataset.

As a result, you should **try many different algorithms for your problem**, while using a hold-out "test set" of data to evaluate performance and select the winner.


Of course, the algorithms you try must be appropriate for your problem, which is where picking the right machine learning task comes in. As an analogy, if you need to clean your house, you might use a vacuum, a broom, or a mop, but you wouldn't bust out a shovel and start digging.

## Machine Learning Tasks

This is Part 1 of this series. In this part, we will cover the "Big 3" machine learning tasks, which are by far the most common ones. They are:

1. Regression
2. Classification
3. Clustering

In Part 2, we will cover dimensionality reduction, including:

4. Feature Selection
5. Feature Extraction

Two notes before continuing:

- We will not cover domain-specific adaptations, such as natural language processing.
- We will not cover every algorithm. There are too many to list, and new ones pop up all the time. However, this list *will* give you a representative overview of successful contemporary algorithms for each task.

# 1. Regression

Regression is the supervised learning task for modeling and predicting **continuous, numeric** variables. Examples include predicting real-estate prices, stock price movements, or student test scores.

Regression tasks are characterized by *labeled datasets that have a numeric target variable*. In other words, you have some "ground truth" value for each observation that you can use to *supervise* your algorithm.


Linear Regression

## 1.1. (Regularized) Linear Regression

Linear regression is one of the most common algorithms for the regression task. In its simplest form, it attempts to fit a straight hyperplane to your dataset (i.e. a straight line when you only have 2 variables). As you might guess, it works well when there are linear relationships between the variables in your dataset.

In practice, simple linear regression is often outclassed by its regularized counterparts (LASSO, Ridge, and Elastic-Net). Regularization is a technique for penalizing large coefficients in order to avoid overfitting, and the strength of the penalty should be tuned.

- **Strengths:** Linear regression is straightforward to understand and explain, and can be regularized to avoid overfitting. In addition, linear models can be updated easily with new data using stochastic gradient descent.
- **Weaknesses:**Linear regression performs poorly when there are non-linear relationships. They are not naturally flexible enough to capture more complex patterns, and adding the right interaction terms or polynomials can be tricky and time-consuming.
- **Implementations:** Python / R

## 1.2. Regression Tree (Ensembles)

Regression trees (a.k.a. decision trees) learn in a hierarchical fashion by repeatedly splitting your dataset into separate branches that maximize the information gain of each split. This branching structure allows regression trees to naturally learn non-linear relationships.

Ensemble methods, such as Random Forests (RF) and Gradient Boosted Trees (GBM), combine predictions from many individual trees. We won't go into their underlying mechanics here, but in practice, RF's often perform very well out-of-the-box while GBM's are harder to tune but tend to have higher performance ceilings.

- **Strengths:** Decision trees can learn non-linear relationships, and are fairly robust to outliers. Ensembles perform very well in practice, winning many classical (i.e. non-deep-learning) machine learning competitions.
- **Weaknesses:** Unconstrained, individual trees are prone to overfitting because they can keep branching until they memorize the training data. However, this can be alleviated by using ensembles.
- **Implementations:** Random Forest - Python / R, Gradient Boosted Tree - Python / R

## 1.3. Deep Learning

Deep learning refers to multi-layer neural networks that can learn extremely complex patterns. They use "hidden layers" between inputs and outputs in order to model *intermediary representations* of the data that other algorithms cannot easily learn.

They have several important mechanisms, such as convolutions and drop-out, that allows them to efficiently learn from high-dimensional data. However, deep learning still requires much more data to train compared to other algorithms because the models have orders of magnitudes more parameters to estimate.

- **Strengths:** Deep learning is the current state-of-the-art for certain domains, such as computer vision and speech recognition. Deep neural networks perform very well on image, audio, and text data, and they can be easily updated with new data using batch propagation. Their architectures (i.e. number and structure of layers) can be adapted to many types of problems, and their hidden layers reduce the need for feature engineering.
- **Weaknesses:** Deep learning algorithms are usually not suitable as general-purpose algorithms because they require a very large amount of data. In fact, they are usually outperformed by tree ensembles for classical machine learning problems. In addition, they are computationally intensive to train, and they require much more expertise to tune (i.e. set the architecture and hyperparameters).
- **Implementations:** Python / R

## 1.4. Honorable Mention: Nearest Neighbors

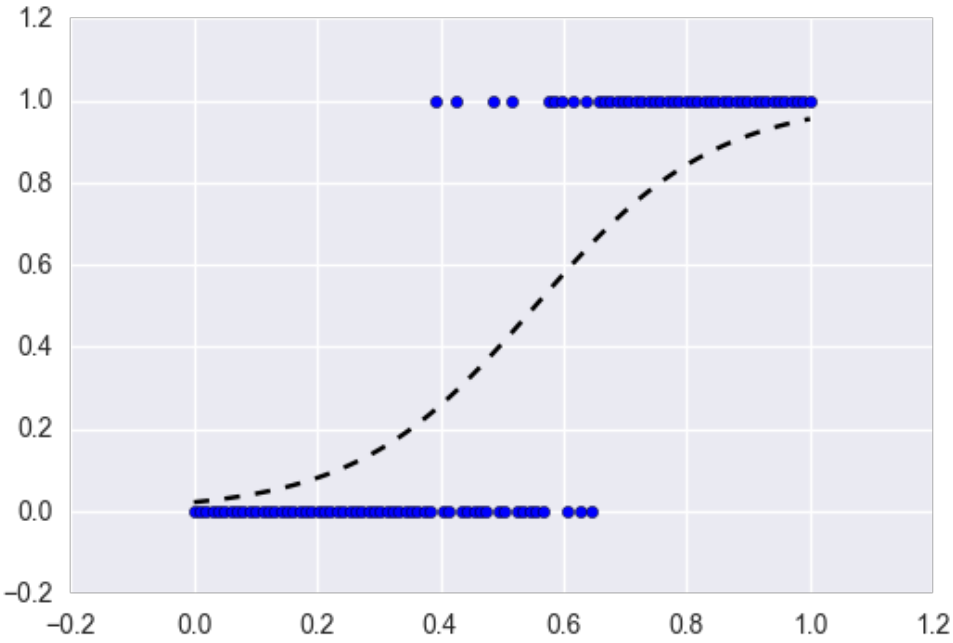
Nearest neighbors algorithms are "instance-based," which means that that save each training observation. They then make predictions for new observations by searching for the most similar training observations and pooling their values.

These algorithms are memory-intensive, perform poorly for high-dimensional data, and require a meaningful distance function to calculate similarity. In practice, training regularized regression or tree ensembles are almost always better uses of your time.

## 2. Classification

Classification is the supervised learning task for modeling and predicting **categorical** variables. Examples include predicting employee churn, email spam, financial fraud, or student letter grades.

As you'll see, many regression algorithms have classification counterparts. The algorithms are adapted to predict a class (or class probabilities) instead of real numbers.




Logistic Regression

### 2.1. (Regularized) Logistic Regression

Logistic regression is the classification counterpart to linear regression. Predictions are mapped to be between 0 and 1 through the logistic function, which means that predictions can be interpreted as class probabilities.

The models themselves are still "linear," so they work well when your classes are linearly separable (i.e. they can be separated by a single decision surface). Logistic regression can also be regularized by penalizing coefficients with a tunable penalty strength.

- **Strengths:** Outputs have a nice probabilistic interpretation, and the algorithm can be regularized to avoid overfitting. Logistic models can be updated easily with new data using stochastic gradient descent.
- **Weaknesses:** Logistic regression tends to underperform when there are multiple or non-linear decision boundaries. They are not flexible enough to naturally capture more complex relationships.
- **Implementations:** Python / R

## 2.2. Classification Tree (Ensembles)

Classification trees are the classification counterparts to regression trees. They are both commonly referred to as "decision trees" or by the umbrella term "classification and regression trees (CART)."

- **Strengths:** As with regression, classification tree ensembles also perform very well in practice. They are robust to outliers, scalable, and able to naturally model non-linear decision boundaries thanks to their hierarchical structure.
- **Weaknesses:** Unconstrained, individual trees are prone to overfitting, but this can be alleviated by ensemble methods.
- **Implementations:** Random Forest - Python / R, Gradient Boosted Tree - Python / R

## 2.3. Deep Learning

To continue the trend, deep learning is also easily adapted to classification problems. In fact, classification is often the more common use of deep learning, such as in image classification.

- **Strengths:** Deep learning performs very well when classifying for audio, text, and image data.
- **Weaknesses:** As with regression, deep neural networks require very large amounts of data to train, so it's not treated as a general-purpose algorithm.
- **Implementations:** Python / R

## 2.4. Support Vector Machines

Support vector machines (SVM) use a mechanism called kernels, which essentially calculate distance between two observations. The SVM algorithm then finds a decision boundary that maximizes the distance between the closest members of separate classes.

For example, an SVM with a linear kernel is similar to logistic regression. Therefore, in practice, the benefit of SVM's typically comes from using non-linear kernels to model non-linear decision boundaries.

- **Strengths:** SVM's can model non-linear decision boundaries, and there are many kernels to choose from. They are also fairly robust against overfitting, especially in high-dimensional space.
- **Weaknesses:** However, SVM's are memory intensive, trickier to tune due to the

importance of picking the right kernel, and don't scale well to larger datasets. Currently in the industry, random forests are usually preferred over SVM's.
- **Implementations:** Python / R

### 2.5. Naive Bayes

Naive Bayes (NB) is a very simple algorithm based around conditional probability and counting. Essentially, your model is actually a probability table that gets updated through your training data. To predict a new observation, you'd simply "look up" the class probabilities in your "probability table" based on its feature values.

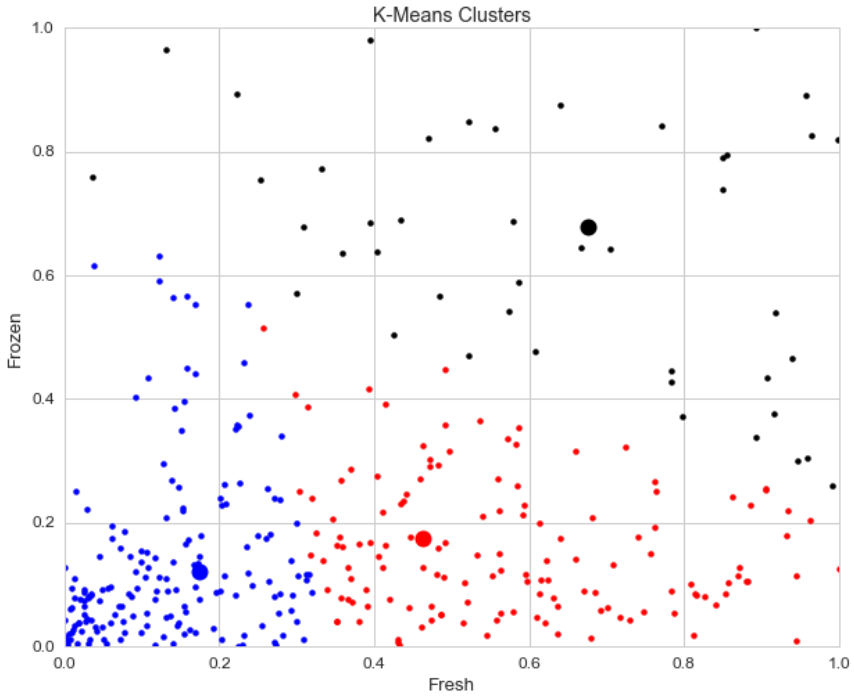
It's called "naive" because its core assumption of conditional independence (i.e. all input features are independent from one another) rarely holds true in the real world.

- **Strengths:** Even though the conditional independence assumption rarely holds true, NB models actually perform surprisingly well in practice, especially for how simple they are. They are easy to implement and can scale with your dataset.
- **Weaknesses:** Due to their sheer simplicity, NB models are often beaten by models properly trained and tuned using the previous algorithms listed.
- **Implementations:** Python / R

## 3. Clustering

Clustering is an unsupervised learningtask for finding natural groupings of observations (i.e. clusters) based on the inherent structure within your dataset. Examples include customer segmentation, grouping similar items in e-commerce, and social network analysis.

Because clustering is unsupervised (i.e. there's no "right answer"), data visualization is usually used to evaluate results. If there is a "right answer" (i.e. you have pre-labeled clusters in your training set), then classification algorithms are typically more appropriate.




K-Means

## 3.1. K-Means

K-Means is a general purpose algorithm that makes clusters based on *geometric distances* (i.e. distance on a coordinate plane) between points. The clusters are grouped around centroids, causing them to be globular and have similar sizes.

This is our recommended algorithm for beginners because it's simple, yet flexible enough to get reasonable results for most problems.

- **Strengths:** K-Means is hands-down the most popular clustering algorithm because it's fast, simple, and surprisingly flexible if you pre-process your data and engineer useful features.
- **Weaknesses:** The user must specify the number of clusters, which won't always be easy to do. In addition, if the true underlying clusters in your data are not globular, then K-Means will produce poor clusters.
- **Implementations:** Python / R

## 3.2. Affinity Propagation

Affinity Propagation is a relatively new clustering technique that makes clusters based on *graph distances* between points. The clusters tend to be smaller and have uneven sizes.

- **Strengths:** The user doesn't need to specify the number of clusters (but does need to specify 'sample preference' and 'damping' hyperparameters).
- **Weaknesses:** The main disadvantage of Affinity Propagation is that it's quite slow and memory-heavy, making it difficult to scale to larger datasets. In addition, it also assumes the true underlying clusters are globular.
- **Implementations:** Python / R

### 3.3. Hierarchical / Agglomerative

Hierarchical clustering, a.k.a. agglomerative clustering, is a suite of algorithms based on the same idea: (1) Start with each point in its own cluster. (2) For each cluster, merge it with another based on some criterion. (3) Repeat until only one cluster remains and you are left with a *hierarchy* of clusters.

- **Strengths:** The main advantage of hierarchical clustering is that the clusters are not assumed to be globular. In addition, it scales well to larger datasets.
- **Weaknesses:** Much like K-Means, the user must choose the number of clusters (i.e. the level of the hierarchy to "keep" after the algorithm completes).
- **Implementations:** Python / R

### 3.4. DBSCAN

DBSCAN is a density based algorithm that makes clusters for *dense regions* of points. There's also a recent new development called HDBSCAN that allows varying density clusters.

- **Strengths:** DBSCAN does not assume globular clusters, and its performance is scalable. In addition, it doesn't require every point to be assigned to a cluster, reducing the noise of the clusters (this may be a weakness, depending on your use case).
- **Weaknesses:** The user must tune the hyperparameters 'epsilon' and 'min_samples,' which define the density of clusters. DBSCAN is quite sensitive to these hyperparameters.
- **Implementations:** Python / R

## Parting Words

We've just taken a whirlwind tour through modern algorithms for the "Big 3" machine learning tasks: Regression, Classification, and Clustering.

In Part 2, we will look at algorithms for Dimensionality Reduction, including Feature Selection and Feature Extraction.

However, we want to leave you with a few words of advice based on our experience:

1. **First... practice, practice, practice.** Reading about algorithms can help you find your footing at the start, but true mastery comes with practice. As you work through projects and/or competitions, you'll develop practical intuition, which unlocks the ability to pick up almost any algorithm and apply it effectively.
2. **Second... master the fundamentals.** There are dozens of algorithms we couldn't list here, and some of them can be quite effective in specific situations. However, almost all of them are some adaptation of the algorithms on this list, which will provide you a strong foundation for applied machine learning.
3. **Finally, remember that better data beats fancier algorithms.** In applied

machine learning, algorithms are commodities because you can easily switch them in and out depending on the problem. However, effective exploratory analysis, data cleaning, and feature engineering can significantly boost your results.

# Dimensionality Reduction Algorithms: Strengths and Weaknesses

elitedatascience.com/dimensionality-reduction-algorithms

May 23, 2017

Welcome to Part 2 of our tour through modern machine learning algorithms. In this part, we'll cover methods for Dimensionality Reduction, further broken into Feature Selection and Feature Extraction. In general, these tasks are rarely performed in isolation. Instead, they're often preprocessing steps to support other tasks.

If you missed Part 1, you can check it out here. It explains our methodology for categorization algorithms, and it covers the "Big 3" machine learning tasks:

1. Regression
2. Classification
3. Clustering

In this part, we'll cover:

4. Feature Selection
5. Feature Extraction

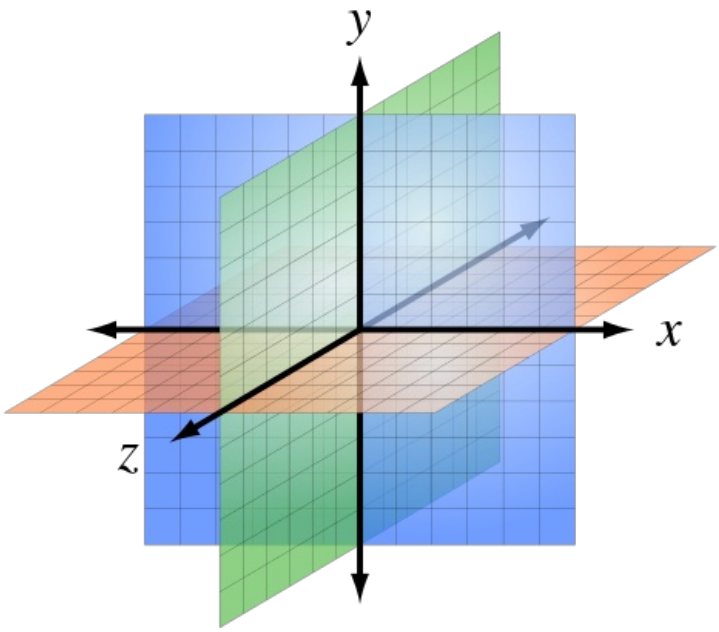
We will also cover other tasks, such as Density Estimation and Anomaly Detection, in dedicated guides in the future.

## The Curse of Dimensionality

In machine learning, "dimensionality" simply refers to the number of features (i.e. input variables) in your dataset.

When the number of features is very large relative to the number of observations in your dataset, *certain* algorithms struggle to train effective models. This is called the "Curse of Dimensionality," and it's especially relevant for clustering algorithms that rely on distance calculations.

A Quora user has provided an excellent analogy for the Curse of Dimensionality, which we'll borrow here:




> Let's say you have a straight line 100 yards long and you dropped a penny somewhere on it. It wouldn't be too hard to find. You walk along the line and it takes two minutes.
>
> Now let's say you have a square 100 yards on each side and you dropped a penny somewhere on it. It would be pretty hard, like searching across two football fields stuck together. It could take days.
>
> Now a cube 100 yards across. That's like searching a 30-story building the size of a football stadium. Ugh.
>
> The difficulty of searching through the space gets a *lot* harder as you have more dimensions.

In this guide, we'll look at the 2 primary methods for reducing dimensionality: Feature Selection and Feature Extraction.

# 4. Feature Selection

Feature selection is for filtering irrelevant or redundant features from your dataset. The key difference between feature selection and extraction is that feature selection keeps a subset of the original features while feature extraction creates brand new ones.

To be clear, some supervised algorithms already have **built-in feature selection**, such as Regularized Regression and Random Forests. Typically, we recommend starting with these algorithms if they fit your task. They're covered in Part 1.

As a stand-alone task, feature selection can be unsupervised (e.g. Variance Thresholds) or supervised (e.g. Genetic Algorithms). You can also combine multiple methods if needed.

## 4.1. Variance Thresholds

Variance thresholds remove features whose values don't change much from observation to observation (i.e. their variance falls below a threshold). These features provide little value.

For example, if you had a public health dataset where 96% of observations were for 35-year-old men, then the 'Age' and 'Gender' features can be eliminated without a major loss in information.

Because variance is dependent on scale, you should always normalize your features first.

- **Strengths:** Applying variance thresholds is based on solid intuition: features that don't change much also don't add much information. This is an easy and relatively safe way to reduce dimensionality at the start of your modeling process.
- **Weaknesses:** If your problem does require dimensionality reduction, applying variance thresholds is rarely sufficient. Furthermore, you must manually set or tune a variance threshold, which could be tricky. We recommend starting with a conservative (i.e. lower) threshold.
- **Implementations:** Python / R

## 4.2. Correlation Thresholds

Correlation thresholds remove features that are highly correlated with others (i.e. its values change very similarly to another's). These features provide redundant information.

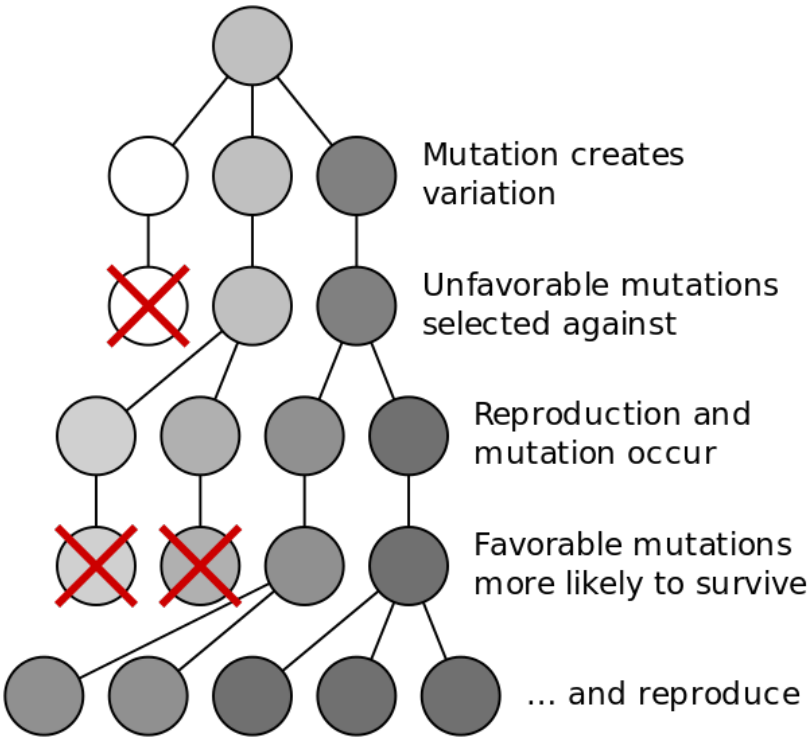
For example, if you had a real-estate dataset with 'Floor Area (Sq. Ft.)' and 'Floor Area (Sq. Meters)' as separate features, you can safely remove one of them.

Which one should you remove? Well, you'd first calculate all pair-wise correlations. Then, if the correlation between a pair of features is above a given threshold, you'd remove the one that has larger mean absolute correlation with other features.

- **Strengths:** Applying correlation thresholds is also based on solid intuition: similar features provide redundant information. Some algorithms are not robust to correlated features, so removing them can boost performance.
- **Weaknesses:** Again, you must manually set or tune a correlation threshold, which can be tricky to do. Plus, if you set your threshold too low, you risk dropping useful information. Whenever possible, we prefer algorithms with built-in feature selection over correlation thresholds. Even for algorithms without built-in feature selection, Principal Component Analysis (PCA) is often a better alternative.
- **Implementations:** Python / R

## 4.3. Genetic Algorithms (GA)

Genetic algorithms (GA) are a broad class of algorithms that can be adapted to different purposes. They are *search algorithms* that are inspired by evolutionary biology and natural selection, combining mutation and cross-over to efficiently traverse large solution spaces. Here's a great intro to the intuition behind GA's.




In machine learning, GA's have two main uses. The first is for *optimization*, such as finding the best weights for a neural network.

The second is for supervised feature selection. In this use case, "genes" represent individual features and the "organism" represents a candidate set of features. Each organism in the "population" is graded on a fitness score such as model performance on a hold-out set. The fittest organisms survive and reproduce, repeating until the population converges on a solution some generations later.

- **Strengths:** Genetic algorithms can efficiently select features from very high dimensional datasets, where exhaustive search is unfeasible. When you need to preprocess data for an algorithm that doesn't have built-in feature selection (e.g. nearest neighbors) and when you must preserve the original features (i.e. no PCA allowed), GA's are likely your best bet. These situations can arise in business/client settings that require a transparent and interpretable solution.
- **Weaknesses:** GA's add a higher level of complexity to your implementation, and they aren't worth the hassle in most cases. If possible, it's faster and simpler to use PCA or to directly use an algorithm with built-in feature selection.
- **Implementations:** Python / R

## 4.4. Honorable Mention: Stepwise Search

Stepwise search is a supervised feature selection method based on sequential search, and it has two flavors: forward and backward. For forward stepwise search, you start without any features. Then, you'd train a 1-feature model using each of your candidate features and keep the version with the best performance. You'd continue adding features, one at a time, until your performance improvements stall.

Backward stepwise search is the same process, just reversed: start with all features in your model and then remove one at a time until performance starts to drop substantially.

We note this algorithm purely for historical reasons. Despite many textbooks listing stepwise search as a valid option, it almost always underperforms other supervised methods such as regularization. Stepwise search has many documented flaws, one of the most fatal being that it's a *greedy* algorithm that can't account for future effects of each change. We don't recommend this method.

# 5. Feature Extraction

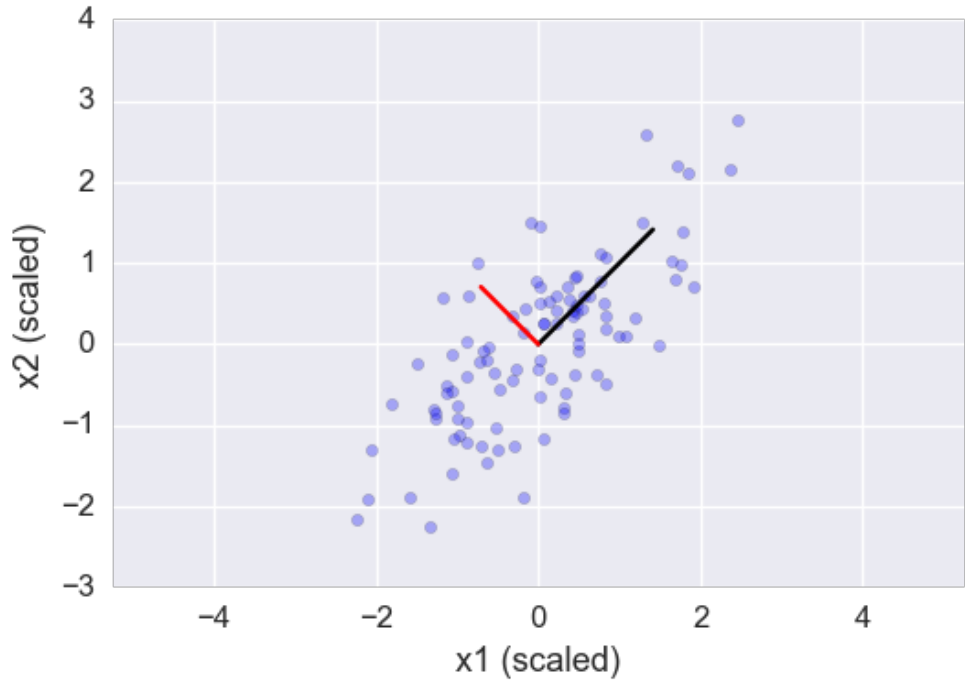
Feature extraction is for creating a new, smaller set of features that stills captures most of the useful information. Again, feature selection keeps a subset of the original features while feature extraction creates new ones.

As with feature selection, some algorithms already have **built-in feature extraction**. The best example is Deep Learning, which extracts increasingly useful representations of the raw input data through each hidden neural layer. We covered this in Part 1.

As a stand-alone task, feature extraction can be unsupervised (i.e. PCA) or supervised (i.e. LDA).

## 4.1. Principal Component Analysis (PCA)

Principal component analysis (PCA) is an unsupervised algorithm that creates linear combinations of the original features. The new features are orthogonal, which means that they are uncorrelated. Furthermore, they are ranked in order of their "explained variance." The *first principal component* (PC1) explains the most variance in your dataset, PC2 explains the second-most variance, and so on.




Therefore, you can reduce dimensionality by limiting the number of principal components to keep based on cumulative explained variance. For example, you might decide to keep only as many principal components as needed to reach a cumulative explained variance of 90%.

You should always normalize your dataset before performing PCA because the transformation is dependent on scale. If you don't, the features that are on the largest scale would dominate your new principal components.

- **Strengths:** PCA is a versatile technique that works well in practice. It's fast and simple to implement, which means you can easily test algorithms with and without PCA to compare performance. In addition, PCA offers several variations and extensions (i.e. kernel PCA, sparse PCA, etc.) to tackle specific roadblocks.
- **Weaknesses:** The new principal components are not interpretable, which may be a deal-breaker in some settings. In addition, you must still manually set or tune a threshold for cumulative explained variance.
- **Implementations:** Python / R

## 4.2. Linear Discriminant Analysis (LDA)

Linear discriminant analysis (LDA) - not to be confused with latent Dirichlet allocation - also creates linear combinations of your original features. However, unlike PCA, LDA doesn't maximize explained variance. Instead, it maximizes the *separability* between classes.

Therefore, LDA is a supervised method that can only be used with labeled data. So which is better: LDA and PCA? Well, results will vary from problem to problem, and the same "No Free Lunch" theorem from Part 1 applies.

The LDA transformation is also dependent on scale, so you should normalize your

dataset first.

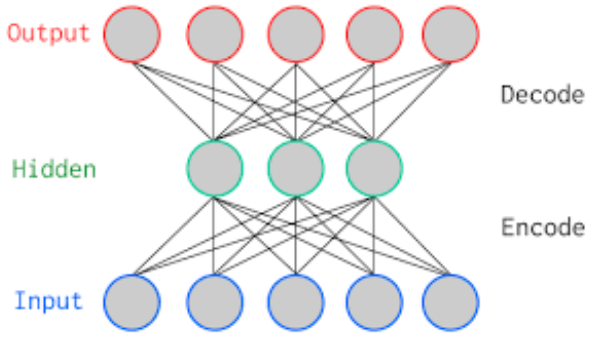
- **Strengths:** LDA is supervised, which *can* (but doesn't always) improve the predictive performance of the extracted features. Furthermore, LDA offers variations (i.e. quadratic LDA) to tackle specific roadblocks.
- **Weaknesses:** As with PCA, the new features are not easily interpretable, and you must still manually set or tune the number of components to keep. LDA also requires labeled data, which makes it more situational.
- **Implementations:** Python / R

### 4.3. Autoencoders

Autoencoders are neural networks that are trained to reconstruct their original inputs. For example, image autoencoders are trained to reproduce the original images instead of classifying the image as a dog or a cat.

So how is this helpful? Well, the key is to structure the hidden layer to have *fewer neurons* than the input/output layers. Thus, that hidden layer will learn to produce a smaller representation of the original image.

Because you use the input image as the target output, autoencoders are considered unsupervised. They can be used directly (e.g. image compression) or stacked in sequence (e.g. deep learning).




- **Strengths:** Autoencoders are neural networks, which means they perform well for certain types of data, such as image and audio data.
- **Weaknesses:** Autoencoders are neural networks, which means they require more data to train. They are not used as general-purpose dimensionality reduction algorithms.
- **Implementations:** Python / R

## Parting Words

We've just taken a whirlwind tour through modern algorithms for Dimensionality Reduction, broken into Feature Selection and Feature Extraction.

We'll leave you with the same parting advice from Part 1.

1. **Practice, practice, practice.** Grab a dataset and strike while the iron is hot.
2. **Master the fundamentals.** For example, it's more fruitful to first understand the differences between PCA and LDA than to dive into the nuances of LDA versus quadratic-LDA.

3. **Remember, better data beats fancier algorithms.** We repeat this a lot, but it's the honest darn truth!

# Best Practices for Feature Engineering

elitedatascience.com/feature-engineering-best-practices

July 26, 2017

Feature engineering, the process creating new input features for machine learning, is one of the most effective ways to improve predictive models.

> Coming up with features is difficult, time-consuming, requires expert knowledge. "Applied machine learning" is basically feature engineering. ~ Andrew Ng

Through feature engineering, you can isolate key information, highlight patterns, and bring in domain expertise.

Unsurprisingly, it can be easy to get stuck because feature engineering is so open-ended.

In this guide, we'll discuss 20 best practices and heuristics that will help you navigate feature engineering.

**Free: Feature Engineering Checklist**

Get all of these heuristics in a **handy PDF checklist** + plenty of other free cheatsheets, checklists, worksheets, and resource lists in our ***Subscriber Vault.***


## What is Feature Engineering?

Feature engineering is an informal topic, and there are many possible definitions. The machine learning workflow is fluid and iterative, so there's no one "right answer."

In a nutshell, we define feature engineering as **creating new features from your existing ones to improve model performance.**

A typical data science process might look like this:

1. Project Scoping / Data Collection
2. Exploratory Analysis
3. Data Cleaning
4. **Feature Engineering**
5. Model Training (including cross-validation to tune hyper-parameters)
6. Project Delivery / Insights

## What is Not Feature Engineering?

That means there are certain steps we do not consider to be feature engineering:

- We do not consider **initial data collection** to be feature engineering.
- Similarly, we do not consider **creating the target variable** to be feature engineering.
- We do not consider removing duplicates, handling missing values, or fixing mislabeled classes to be feature engineering. We put these under **data cleaning**.
- We do not consider **scaling or normalization** to be feature engineering because these steps belong inside the cross-validation loop (i.e. after you've already built your analytical base table).
- Finally, we do not consider **feature selection or PCA** to be feature engineering. These steps also belong inside your cross-validation loop.

Again, this is simply *our* categorization. Reasonable data scientists may disagree, and that's perfectly fine.

With those disclaimers out of the way, let's dive into the best practices and heuristics!

## Indicator Variables

The first type of feature engineering involves using indicator variables to isolate key information.

Now, some of you may be wondering, "shouldn't a good algorithm learn the key information on its own?"

Well, not always. It depends on the amount of data you have and the strength of competing signals. You can help your algorithm "focus" on what's important by highlighting it beforehand.

- **Indicator variable from thresholds:** Let's say you're studying alcohol preferences by U.S. consumers and your dataset has an age feature. You can create an indicator variable for age >= 21 to distinguish subjects who were over the legal drinking age.
- **Indicator variable from multiple features:** You're predicting real-estate prices and you have the features n_bedrooms and n_bathrooms. If houses with 2 beds and 2 baths command a premium as rental properties, you can create an indicator

variable to flag them.

- **Indicator variable for special events:** You're modeling weekly sales for an e-commerce site. You can create two indicator variables for the weeks of Black Friday and Christmas.
- **Indicator variable for groups of classes:** You're analyzing website conversions and your dataset has the categorical feature traffic_source. You could create an indicator variable for paid_traffic by flagging observations with traffic source values of  or .

## Interaction Features

The next type of feature engineering involves highlighting interactions between two or more features.

Have you ever heard the phrase, "the sum is greater than the parts?" Well, some features can be combined to provide more information than they would as individuals.

Specifically, look for opportunities to take the sum, difference, product, or quotient of multiple features.

**Note: We don't recommend using an automated loop to create interactions for all your features. This leads to "feature explosion."*

- **Sum of two features:** Let's say you wish to predict revenue based on preliminary sales data. You have the features sales_blue_pens and sales_black_pens. You could sum those features if you only care about overall sales_pens.
- **Difference between two features:** You have the features house_built_date and house_purchase_date. You can take their difference to create the feature house_age_at_purchase.
- **Product of two features:** You're running a pricing test, and you have the feature price and an indicator variable conversion. You can take their product to create the feature earnings.
- **Quotient of two features:** You have a dataset of marketing campaigns with the features n_clicks and n_impressions. You can divide clicks by impressions to create click_through_rate, allowing you to compare across campaigns of different volume.

## Feature Representation

This next type of feature engineering is simple yet impactful. It's called feature representation.

Your data won't always come in the ideal format. You should consider if you'd gain information by representing the same feature in a different way.

- **Date and time features:** Let's say you have the feature purchase_datetime. It might be more useful to extract purchase_day_of_week and purchase_hour_of_day.

You can also aggregate observations to create features such as purchases_over_last_30_days.

- **Numeric to categorical mappings:** You have the feature years_in_school. You might create a new feature grade with classes such as "Elementary School", "Middle School", and "High School".
- **Grouping sparse classes:** You have a feature with many classes that have low sample counts. You can try grouping similar classes and then grouping the remaining ones into a single "Other" class.
- **Creating dummy variables:** Depending on your machine learning implementation, you may need to manually transform categorical features into dummy variables. You should always do this *after* grouping sparse classes.

## External Data

An underused type of feature engineering is bringing in external data. This can lead to some of the biggest breakthroughs in performance.

For example, one way quantitative hedge funds perform research is by layering together different streams of financial data.

Many machine learning problems can benefit from bringing in external data. Here are some examples:

- **Time series data:** The nice thing about time series data is that you only need one feature, some form of date, to layer in features from another dataset.
- **External API's:** There are plenty of API's that can help you create features. For example, the Microsoft Computer Vision API can return the number of faces from an image.
- **Geocoding:** Let's say have you street_address, city, and state. Well, you can geocode them into latitude and longitude. This will allow you to calculate features such as local demographics (e.g. median_income_within_2_miles) with the help of another dataset.
- **Other sources of the same data:** How many ways could you track a Facebook ad campaign? You might have Facebook's own tracking pixel, Google Analytics, and possibly another third-party software. Each source can provide information that the others don't track. Plus, any differences between the datasets could be informative (e.g. bot traffic that one source ignores while another source keeps).

## Error Analysis (Post-Modeling)

The final type of feature engineering we'll cover falls under a process called error analysis. This is performed *after* training your first model.

Error analysis is a broad term that refers to analyzing the misclassified or high error observations from your model and deciding on your next steps for improvement.

Possible next steps include collecting more data, splitting the problem apart, or engineering new features that address the errors. To use error analysis for feature engineering, you'll need to understand *why* your model missed its mark.

Here's how:

- **Start with larger errors**: Error analysis is typically a manual process. You won't have time to scrutinize every observation. We recommend starting with those that had higher error scores. Look for patterns that you can formalize into new features.
- **Segment by classes:** Another technique is to segment your observations and compare the average error within each segment. You can try creating indicator variables for the segments with the highest errors.
- **Unsupervised clustering:** If you have trouble spotting patterns, you can run an unsupervised clustering algorithm on the misclassified observations. We don't recommend blindly using those clusters as a new feature, but they can make it easier to spot patterns. Remember, the goal is to understand *why* observations were misclassified.
- **Ask colleagues or domain experts:** This is a great complement to any of the other three techniques. Asking a domain expert is especially useful if you've identified a pattern of poor performance (e.g. through segmentations) but don't yet understand why.

## Conclusion

As you see, there are many possibilities for feature engineering. We've covered 20 best practices and heuristics, but they are by no means exhaustive!

Remember these general guidelines as you start to experiment on your own:

**Good features to engineer…**

- Can be computed for future observations.
- Are usually intuitive to explain.
- Are informed by domain knowledge or exploratory analysis.
- Must have the potential to be predictive. Don't just create features for the sake of it.
- **Never touch the target variable.** This a trap that beginners sometimes fall into. Whether you're creating indicator variables or interaction features, never use your target variable. That's like "cheating" and it would give you very misleading results.

**Finally, don't worry if this feels overwhelming right now!** You'll naturally get better at feature engineering through practice and experience.

In fact, if this is your first exposure to some of these tactics, we highly recommend picking up a dataset and solidifying what you've learned. Check out some ideas for Fun Machine Learning Projects for Beginners.

# How to Handle Imbalanced Classes in Machine Learning

elitedatascience.com/imbalanced-classes

July 5, 2017

Imbalanced classes put "accuracy" out of business. This is a surprisingly common problem in machine learning (specifically in classification), occurring in datasets with a disproportionate ratio of observations in each class.

Standard accuracy no longer reliably measures performance, which makes model training much trickier.

Imbalanced classes appear in many domains, including:

- Fraud detection
- Spam filtering
- Disease screening
- SaaS subscription churn
- Advertising click-throughs

In this guide, we'll explore 5 effective ways to handle imbalanced classes.

**Free: Code Cheatsheet**

Get all of this tutorial's code in a **handy PDF cheatsheet** + plenty of other free cheatsheets, checklists, worksheets, and resource lists in our ***Subscriber Vault.***


## Intuition: Disease Screening Example

Let's say your client is a leading research hospital, and they've asked you to train a model for detecting a disease based on biological inputs collected from patients.

But here's the catch... the disease is relatively rare; it occurs in only 8% of patients who are screened.

Now, before you even start, do you see how the problem might break? Imagine if you didn't bother training a model at all. Instead, what if you just wrote a single line of code that always predicts 'No Disease?'

A crappy, but accurate, solution
Python

```
def disease_screen(patient_data):
    # Ignore patient_data
    return 'No Disease.'
```

Well, guess what? Your "solution" would have 92% accuracy!

Unfortunately, that accuracy is misleading.

- For patients who *do not* have the disease, you'd have 100% accuracy.
- For patients who *do* have the disease, you'd have 0% accuracy.
- Your overall accuracy would be high simply because most patients do not have the disease (not because your model is any good).

This is clearly a problem because many machine learning algorithms are designed to maximize overall accuracy. The rest of this guide will illustrate different tactics for handling imbalanced classes.

### Important notes before we begin:

First, please note that we're not going to split out a separate test set, tune hyperparameters, or implement cross-validation. In other words, we're not necessarily going to follow best practices.

Instead, this tutorial is focused purely on addressing imbalanced classes.

In addition, not every technique below will work for every problem. However, 9 times out of 10, at least one of these techniques should do the trick.

## Balance Scale Dataset

For this guide, we'll use a synthetic dataset called Balance Scale Data, which you can download from the UCI Machine Learning Repository here.

This dataset was originally generated to model psychological experiment results, but it's useful for us because it's a manageable size and has imbalanced classes.

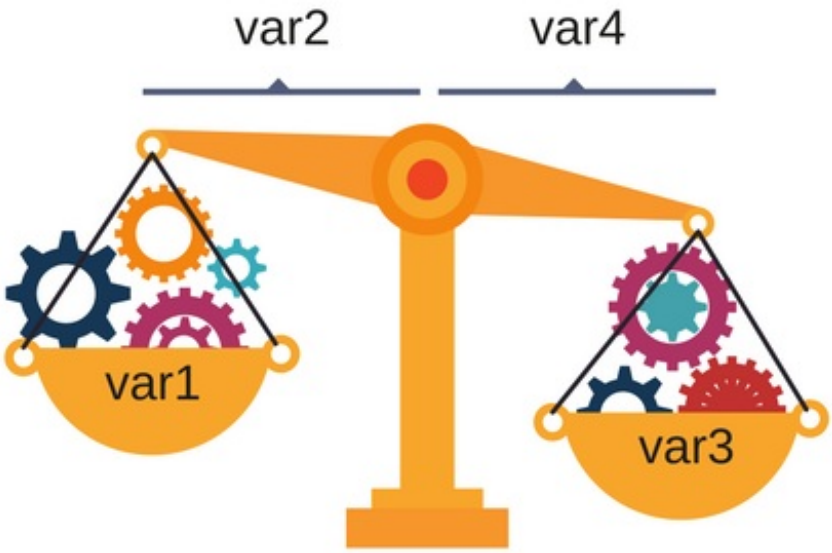
Import libraries and read dataset
Python

```
import pandas as pd
import numpy as np
# Read dataset
df = pd.read_csv('balance-scale.data',
                 names=['balance', 'var1', 'var2', 'var3', 'var4'])
# Display example observations
df.head()
```

The dataset contains information about whether a scale is balanced or not, based on weights and distances of the two arms.

- It has 1 target variable, which we've labeled balance .
- It has 4 input features, which we've labeled var1 through var4 .

| | balance | var1 | var2 | var3 | var4 |
|---|---|---|---|---|---|
| **0** | B | 1 | 1 | 1 | 1 |
| **1** | R | 1 | 1 | 1 | 2 |
| **2** | R | 1 | 1 | 1 | 3 |
| **3** | R | 1 | 1 | 1 | 4 |
| **4** | R | 1 | 1 | 1 | 5 |




The target variable has 3 classes.

- **R** for right-heavy, i.e. when var3 * var4 > var1 * var2
- **L** for left-heavy, i.e. when var3 * var4 < var1 * var2
- **B** for balanced, i.e. when var3 * var4 = var1 * var2

Count of each class
Python

```
df['balance'].value_counts()
# R    288
# L    288
# B     49
# Name: balance, dtype: int64
```

However, for this tutorial, we're going to turn this into a **binary classification** problem.

We're going to label each observation as **1** (positive class) if the scale is balanced or **0** (negative class) if the scale is not balanced:

Transform into binary classification
Python

```
# Transform into binary classification
df['balance'] = [1 if b=='B' else 0 for b in df.balance]
df['balance'].value_counts()
# 0    576
# 1     49
# Name: balance, dtype: int64
# About 8% were balanced

```

As you can see, only about 8% of the observations were balanced. Therefore, if we were to always predict **0**, we'd achieve an accuracy of 92%.

## The Danger of Imbalanced Classes

Now that we have a dataset, we can really show the dangers of imbalanced classes.

First, let's import the Logistic Regression algorithm and the accuracy metric from Scikit-Learn.

Import algorithm and accuracy metric
Python

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

Next, we'll fit a very simple model using default settings for everything.

Train model on imbalanced data
Python

```
# Separate input features (X) and target variable (y)
y = df.balance
X = df.drop('balance', axis=1)
# Train model
clf_0 = LogisticRegression().fit(X, y)
# Predict on training set
pred_y_0 = clf_0.predict(X)


```

As mentioned above, many machine learning algorithms are designed to maximize overall accuracy by default.

We can confirm this:

Python

```
# How's the accuracy?
print( accuracy_score(pred_y_0, y) )
# 0.9216
```

So our model has 92% overall accuracy, but is it because it's predicting only 1 class?

Python

```
# Should we be excited?
print( np.unique( pred_y_0 ) )
# [0]
```

As you can see, this model is only predicting **0**, which means it's completely ignoring the minority class in favor of the majority class.

Next, we'll look at the first technique for handling imbalanced classes: up-sampling the minority class.

## 1. Up-sample Minority Class

Up-sampling is the process of randomly duplicating observations from the minority class in order to reinforce its signal.

There are several heuristics for doing so, but the most common way is to simply resample with replacement.

First, we'll import the resampling module from Scikit-Learn:

Module for resampling
Python

```
from sklearn.utils import resample
```

Next, we'll create a new DataFrame with an up-sampled minority class. Here are the steps:

1. First, we'll separate observations from each class into different DataFrames.
2. Next, we'll resample the minority class **with replacement**, setting the number of samples to match that of the majority class.
3. Finally, we'll combine the up-sampled minority class DataFrame with the original majority class DataFrame.

Here's the code:

Upsample minority class
Python

```
# Separate majority and minority classes
df_majority = df[df.balance==0]
df_minority = df[df.balance==1]
# Upsample minority class
df_minority_upsampled = resample(df_minority,
                                 replace=True,     # sample with replacement
                                 n_samples=576,    # to match majority class
                                 random_state=123) # reproducible results
# Combine majority class with upsampled minority class
df_upsampled = pd.concat([df_majority, df_minority_upsampled])
# Display new class counts
df_upsampled.balance.value_counts()
# 1    576
# 0    576
# Name: balance, dtype: int64
```

As you can see, the new DataFrame has more observations than the original, and the ratio of the two classes is now 1:1.

Let's train another model using Logistic Regression, this time on the balanced dataset:

Train model on upsampled dataset
Python

```
# Separate input features (X) and target variable (y)
y = df_upsampled.balance
X = df_upsampled.drop('balance', axis=1)
# Train model
clf_1 = LogisticRegression().fit(X, y)
# Predict on training set
pred_y_1 = clf_1.predict(X)
# Is our model still predicting just one class?
print( np.unique( pred_y_1 ) )
# [0 1]
# How's our accuracy?
print( accuracy_score(y, pred_y_1) )
# 0.513888888889
```

Great, now the model is no longer predicting just one class. While the accuracy also took a nosedive, it's now more meaningful as a performance metric.

## 2. Down-sample Majority Class

Down-sampling involves randomly removing observations from the majority class to prevent its signal from dominating the learning algorithm.

The most common heuristic for doing so is resampling without replacement.

The process is similar to that of up-sampling. Here are the steps:

1. First, we'll separate observations from each class into different DataFrames.
2. Next, we'll resample the majority class **without replacement**, setting the number of samples to match that of the minority class.
3. Finally, we'll combine the down-sampled majority class DataFrame with the original minority class DataFrame.

Here's the code:

Downsample majority class
Python

```
# Separate majority and minority classes
df_majority = df[df.balance==0]
df_minority = df[df.balance==1]
# Downsample majority class
df_majority_downsampled = resample(df_majority,
                                   replace=False,    # sample without replacement
                                   n_samples=49,     # to match minority class
                                   random_state=123) # reproducible results
# Combine minority class with downsampled majority class
df_downsampled = pd.concat([df_majority_downsampled, df_minority])
# Display new class counts
df_downsampled.balance.value_counts()
# 1    49
# 0    49
# Name: balance, dtype: int64


```

This time, the new DataFrame has fewer observations than the original, and the ratio of the two classes is now 1:1.

Again, let's train a model using Logistic Regression:

Train model on downsampled dataset
Python

```
# Separate input features (X) and target variable (y)
y = df_downsampled.balance
X = df_downsampled.drop('balance', axis=1)
# Train model
clf_2 = LogisticRegression().fit(X, y)
# Predict on training set
pred_y_2 = clf_2.predict(X)
# Is our model still predicting just one class?
print( np.unique( pred_y_2 ) )
# [0 1]
# How's our accuracy?
print( accuracy_score(y, pred_y_2) )
# 0.581632653061


```

The model isn't predicting just one class, and the accuracy seems higher.

We'd still want to validate the model on an unseen test dataset, but the results are more encouraging.

## 3. Change Your Performance Metric

So far, we've looked at two ways of addressing imbalanced classes by resampling the dataset. Next, we'll look at using other performance metrics for evaluating the models.

Albert Einstein once said, "if you judge a fish on its ability to climb a tree, it will live its whole life believing that it is stupid." This quote really highlights the importance of choosing the right evaluation metric.

For a general-purpose metric for classification, we recommend **Area Under ROC Curve** (AUROC).

- We won't dive into its details in this guide, but you can read more about it here.
- Intuitively, AUROC represents the likelihood of your model distinguishing observations from two classes.
- In other words, if you randomly select one observation from each class, what's the probability that your model will be able to "rank" them correctly?

We can import this metric from Scikit-Learn:

Area Under ROC Curve
Python

```
from sklearn.metrics import roc_auc_score
```

To calculate AUROC, you'll need predicted class probabilities instead of just the predicted classes. You can get them using the .predict_proba() function like so:

Get class probabilities
Python

```
# Predict class probabilities
prob_y_2 = clf_2.predict_proba(X)
# Keep only the positive class
prob_y_2 = [p[1] for p in prob_y_2]
prob_y_2[:5] # Example
# [0.45419197226479618,
#  0.48205962213283882,
#  0.46862327066392456,
#  0.47868378832689096,
#  0.58143856820159667]


```

So how did this model (trained on the down-sampled dataset) do in terms of AUROC?

AUROC of model trained on downsampled dataset
Python

```
print( roc_auc_score(y, prob_y_2) )
# 0.568096626406
```

Ok... and how does this compare to the original model trained on the imbalanced dataset?

AUROC of model trained on imbalanced dataset
Python

```
prob_y_0 = clf_0.predict_proba(X)
prob_y_0 = [p[1] for p in prob_y_0]
print( roc_auc_score(y, prob_y_0) )
# 0.530718537415

```

Remember, our original model trained on the imbalanced dataset had an accuracy of 92%, which is much higher than the 58% accuracy of the model trained on the down-sampled dataset.

However, the latter model has an AUROC of 57%, which is higher than the 53% of the original model (but not by much).

**Note:** if you got an AUROC of 0.47, it just means you need to invert the predictions because Scikit-Learn is misinterpreting the positive class. AUROC should be >= 0.5.

## 4. Penalize Algorithms (Cost-Sensitive Training)

The next tactic is to use penalized learning algorithms that increase the cost of classification mistakes on the minority class.

A popular algorithm for this technique is Penalized-SVM:

Support Vector Machine
Python

```
from sklearn.svm import SVC
```

During training, we can use the argument class_weight='balanced'  to penalize mistakes on the minority class by an amount proportional to how under-represented it is.

We also want to include the argument probability=True  if we want to enable probability estimates for SVM algorithms.

Let's train a model using Penalized-SVM on the original imbalanced dataset:

Train Penalized SVM on imbalanced dataset
Python

```
# Separate input features (X) and target variable (y)
y = df.balance
X = df.drop('balance', axis=1)
# Train model
clf_3 = SVC(kernel='linear',
            class_weight='balanced', # penalize
            probability=True)
clf_3.fit(X, y)
# Predict on training set
pred_y_3 = clf_3.predict(X)
# Is our model still predicting just one class?
print( np.unique( pred_y_3 ) )
# [0 1]
# How's our accuracy?
print( accuracy_score(y, pred_y_3) )
# 0.688
# What about AUROC?
prob_y_3 = clf_3.predict_proba(X)
prob_y_3 = [p[1] for p in prob_y_3]
print( roc_auc_score(y, prob_y_3) )
# 0.5305236678
```

Again, our purpose here is only to illustrate this technique. To really determine which of these tactics works best *for this problem*, you'd want to evaluate the models on a hold-out test set.

## 5. Use Tree-Based Algorithms

The final tactic we'll consider is using tree-based algorithms. Decision trees often perform well on imbalanced datasets because their hierarchical structure allows them to learn signals from both classes.

In modern applied machine learning, tree ensembles (Random Forests, Gradient Boosted Trees, etc.) almost always outperform singular decision trees, so we'll jump right into those:

Random Forest
Python

```
from sklearn.ensemble import RandomForestClassifier
```

Now, let's train a model using a Random Forest on the original imbalanced dataset.

Train Random Forest on imbalanced dataset
Python

```
# Separate input features (X) and target variable (y)
y = df.balance
X = df.drop('balance', axis=1)
# Train model
clf_4 = RandomForestClassifier()
clf_4.fit(X, y)
# Predict on training set
pred_y_4 = clf_4.predict(X)
# Is our model still predicting just one class?
print( np.unique( pred_y_4 ) )
# [0 1]
# How's our accuracy?
print( accuracy_score(y, pred_y_4) )
# 0.9744
# What about AUROC?
prob_y_4 = clf_4.predict_proba(X)
prob_y_4 = [p[1] for p in prob_y_4]
print( roc_auc_score(y, prob_y_4) )
# 0.999078798186
```

Wow! 97% accuracy and nearly 100% AUROC? Is this magic? A sleight of hand? Cheating? Too good to be true?

Well, tree ensembles have become very popular because they perform extremely well on many real-world problems. We certainly recommend them wholeheartedly.

**However:**

While these results are encouraging, the model *could* be overfit, so you should still evaluate your model on an unseen test set before making the final decision.

*Note: your numbers may differ slightly due to the randomness in the algorithm. You can set a random seed for reproducible results.*

## Honorable Mentions

There were a few tactics that didn't make it into this tutorial:

### Create Synthetic Samples (Data Augmentation)

Creating synthetic samples is a close cousin of up-sampling, and some people might categorize them together. For example, the SMOTE algorithm is a method of resampling from the minority class while slightly perturbing feature values, thereby creating "new" samples.

You can find an implementation of SMOTE in the imblearn library.

### Combine Minority Classes

Combining minority classes of your target variable may be appropriate for some multi-class problems.

For example, let's say you wished to predict credit card fraud. In your dataset, each method of fraud may be labeled separately, but you might not care about distinguishing them. You could combine them all into a single 'Fraud' class and treat the problem as binary classification.

### Reframe as Anomaly Detection

Anomaly detection, a.k.a. outlier detection, is for detecting outliers and rare events. Instead of building a classification model, you'd have a "profile" of a normal observation. If a new observation strays too far from that "normal profile," it would be flagged as an anomaly.

## Conclusion & Next Steps

In this guide, we covered 5 tactics for handling imbalanced classes in machine learning:

1. Up-sample the minority class
2. Down-sample the majority class
3. Change your performance metric
4. Penalize algorithms (cost-sensitive training)
5. Use tree-based algorithms

These tactics are subject to the No Free Lunch theorem, and you should try several of them and use the results from the test set to decide on the best solution for your problem.

# Overfitting in Machine Learning: What It Is and How to Prevent It

elitedatascience.com/overfitting-in-machine-learning

September 7, 2017

Did you know that there's one mistake...

...that thousands of data science beginners unknowingly commit?

And that this mistake can single-handedly ruin your machine learning model?

No, that's not an exaggeration. We're talking about one of the trickiest obstacles in applied machine learning: *overfitting*.

But don't worry:

In this guide, we'll walk you through exactly what overfitting means, how to spot it in your models, and what to do if your model is overfit.

By the end, you'll know how to deal with this tricky problem once and for all.

## Table of Contents

## Examples of Overfitting

Let's say we want to predict if a student will land a job interview based on her resume.

Now, assume we train a model from a dataset of 10,000 resumes and their outcomes.

Next, we try the model out on the original dataset, and it predicts outcomes with 99% accuracy... wow!

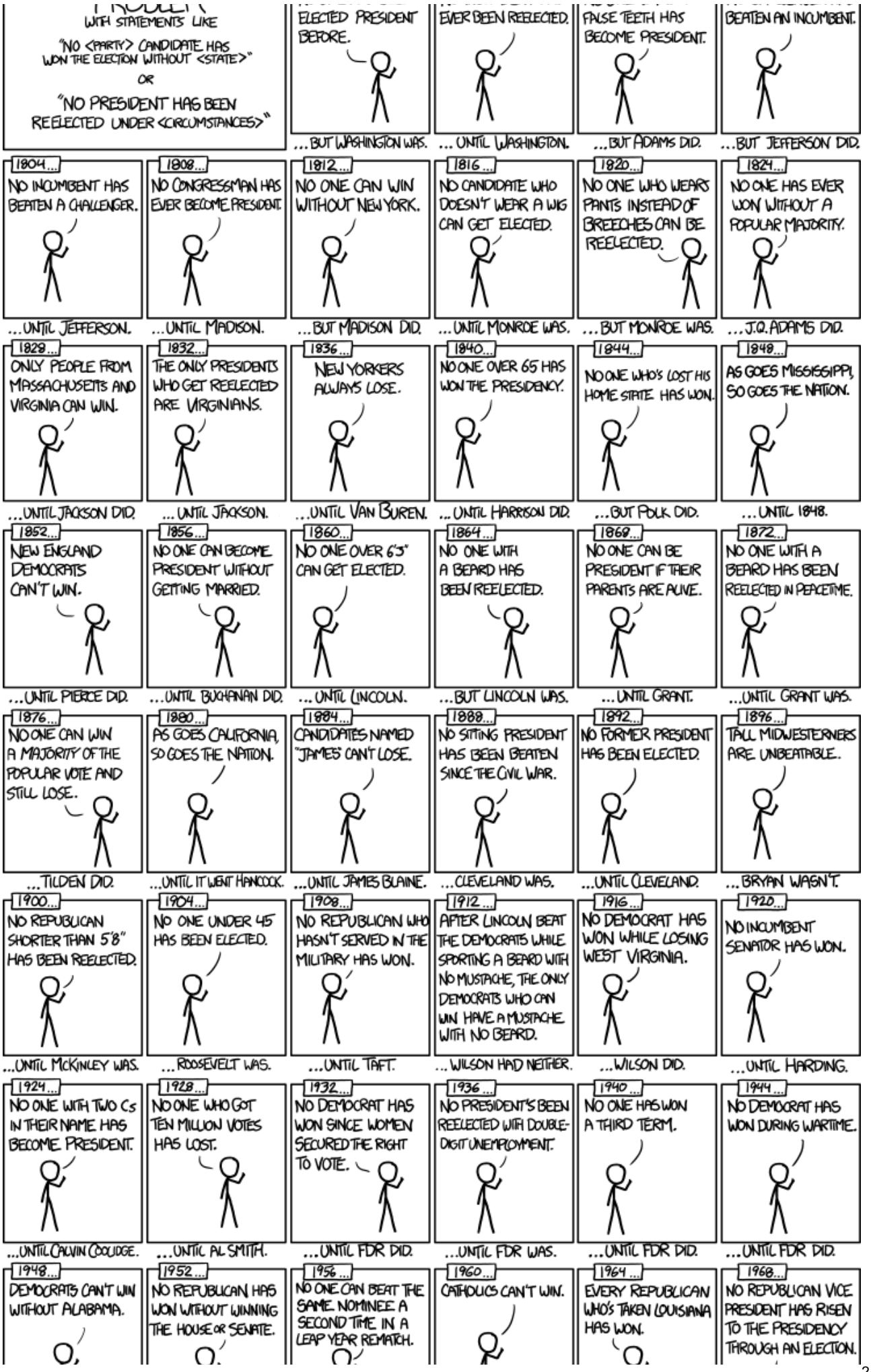
But now comes the bad news.

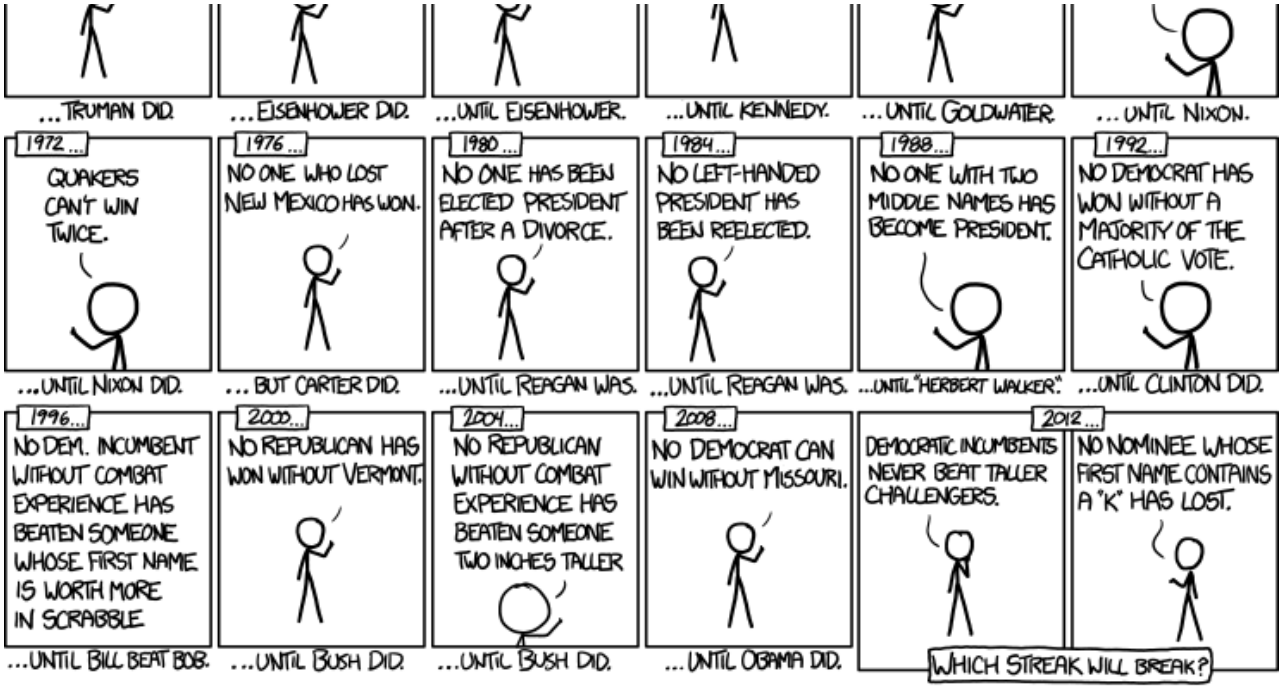
When we run the model on a new ("unseen") dataset of resumes, we only get 50% accuracy... uh-oh!

**Our model doesn't *generalize* well from our training data to unseen data.**

This is known as overfitting, and it's a common problem in machine learning and data science.

In fact, overfitting occurs in the real world all the time. You only need to turn on the news channel to hear examples:


WITH STATEMENTS LIKE
"NO <PARTY> CANDIDATE HAS WON THE ELECTION WITHOUT <STATE>"
OR
"NO PRESIDENT HAS BEEN REELECTED UNDER <CIRCUMSTANCES>"
ELECTED PRESIDENT BEFORE.
...BUT WASHINGTON WAS.
EVER BEEN REELECTED.
...UNTIL WASHINGTON.
FALSE TEETH HAS BECOME PRESIDENT.
...BUT ADAMS DID.
BEATEN AN INCUMBENT.
...BUT JEFFERSON DID.
1804...
NO INCUMBENT HAS BEATEN A CHALLENGER.
...UNTIL JEFFERSON.
1808...
NO CONGRESSMAN HAS EVER BECOME PRESIDENT.
...UNTIL MADISON.
1812...
NO ONE CAN WIN WITHOUT NEW YORK.
...BUT MADISON DID.
1816...
NO CANDIDATE WHO DOESN'T WEAR A WIG CAN GET ELECTED.
...UNTIL MONROE WAS.
1820...
NO ONE WHO WEARS PANTS INSTEAD OF BREECHES CAN BE REELECTED.
...BUT MONROE WAS.
1824...
NO ONE HAS EVER WON WITHOUT A POPULAR MAJORITY.
...J.Q. ADAMS DID.
1828...
ONLY PEOPLE FROM MASSACHUSETTS AND VIRGINIA CAN WIN.
...UNTIL JACKSON DID.
1832...
THE ONLY PRESIDENTS WHO GET REELECTED ARE VIRGINIANS.
...UNTIL JACKSON.
1836...
NEW YORKERS ALWAYS LOSE.
...UNTIL VAN BUREN.
1840...
NO ONE OVER 65 HAS WON THE PRESIDENCY.
...UNTIL HARRISON DID.
1844...
NO ONE WHO'S LOST HIS HOME STATE HAS WON.
...BUT POLK DID.
1848...
AS GOES MISSISSIPPI, SO GOES THE NATION.
...UNTIL 1848.
1852...
NEW ENGLAND DEMOCRATS CAN'T WIN.
...UNTIL PIERCE DID.
1856...
NO ONE CAN BECOME PRESIDENT WITHOUT GETTING MARRIED.
...UNTIL BUCHANAN DID.
1860...
NO ONE OVER 6'3" CAN GET ELECTED.
...UNTIL LINCOLN.
1864...
NO ONE WITH A BEARD HAS BEEN REELECTED.
...BUT LINCOLN WAS.
1868...
NO ONE CAN BE PRESIDENT IF THEIR PARENTS ARE ALIVE.
...UNTIL GRANT.
1872...
NO ONE WITH A BEARD HAS BEEN REELECTED IN PEACETIME.
...UNTIL GRANT WAS.
1876...
NO ONE CAN WIN A MAJORITY OF THE POPULAR VOTE AND STILL LOSE.
...TILDEN DID.
1880...
AS GOES CALIFORNIA, SO GOES THE NATION.
...UNTIL IT WENT HANCOCK.
1884...
CANDIDATES NAMED "JAMES" CAN'T LOSE.
...UNTIL JAMES BLAINE.
1888...
NO SITTING PRESIDENT HAS BEEN BEATEN SINCE THE CIVIL WAR.
...CLEVELAND WAS.
1892...
NO FORMER PRESIDENT HAS BEEN ELECTED.
...UNTIL CLEVELAND.
1896...
TALL MIDWESTERNERS ARE UNBEATABLE.
...BRYAN WASN'T.
1900...
NO REPUBLICAN SHORTER THAN 5'8" HAS BEEN REELECTED.
...UNTIL MCKINLEY WAS.
1904...
NO ONE UNDER 45 HAS BEEN ELECTED.
...ROOSEVELT WAS.
1908...
NO REPUBLICAN WHO HASN'T SERVED IN THE MILITARY HAS WON.
...UNTIL TAFT.
1912...
AFTER LINCOLN BEAT THE DEMOCRATS WHILE SPORTING A BEARD WITH NO MUSTACHE, THE ONLY DEMOCRATS WHO CAN WIN HAVE A MUSTACHE WITH NO BEARD.
...WILSON HAD NEITHER.
1916...
NO DEMOCRAT HAS WON WHILE LOSING WEST VIRGINIA.
...WILSON DID.
1920...
NO INCUMBENT SENATOR HAS WON.
...UNTIL HARDING.
1924...
NO ONE WITH TWO Cs IN THEIR NAME HAS BECOME PRESIDENT.
...UNTIL CALVIN COOLIDGE.
1928...
NO ONE WHO GOT TEN MILLION VOTES HAS LOST.
...UNTIL AL SMITH.
1932...
NO DEMOCRAT HAS WON SINCE WOMEN SECURED THE RIGHT TO VOTE.
...UNTIL FDR DID.
1936...
NO PRESIDENT'S BEEN REELECTED WITH DOUBLE-DIGIT UNEMPLOYMENT.
...UNTIL FDR WAS.
1940...
NO ONE HAS WON A THIRD TERM.
...UNTIL FDR DID.
1944...
NO DEMOCRAT HAS WON DURING WARTIME.
...UNTIL FDR DID.
1948...
DEMOCRATS CAN'T WIN WITHOUT ALABAMA.
1952...
NO REPUBLICAN HAS WON WITHOUT WINNING THE HOUSE OR SENATE.
1956...
NO ONE CAN BEAT THE SAME NOMINEE A SECOND TIME IN A LEAP YEAR REMATCH.
1960...
CATHOLICS CAN'T WIN.
1964...
EVERY REPUBLICAN WHO'S TAKEN LOUISIANA HAS WON.
1968...
NO REPUBLICAN VICE PRESIDENT HAS RISEN TO THE PRESIDENCY THROUGH AN ELECTION.




Overfitting Electoral Precedence (source: XKCD)

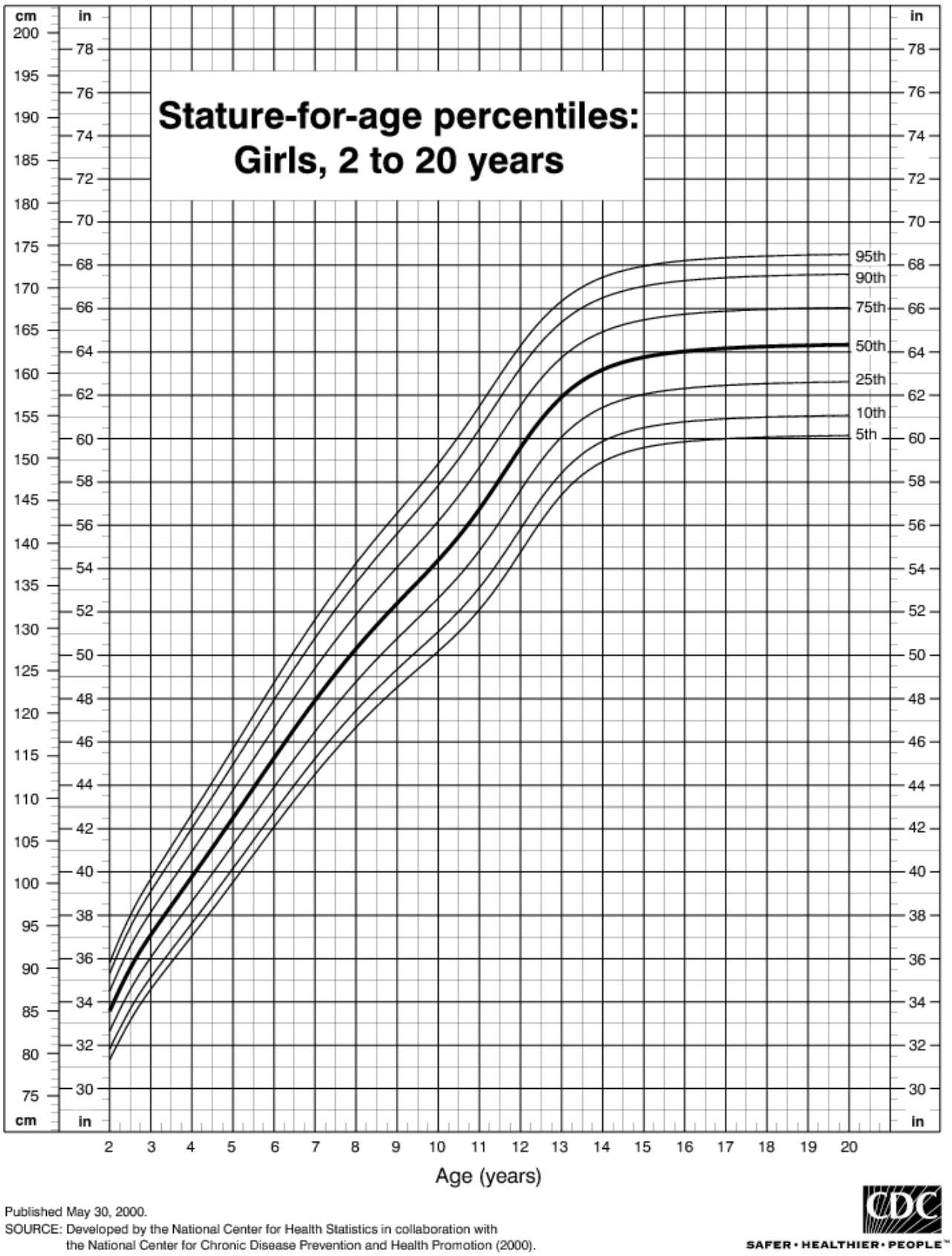
## Signal vs. Noise

You may have heard of the famous book The Signal and the Noise by Nate Silver.

In predictive modeling, you can think of the “signal” as the true underlying pattern that you wish to learn from the data.

“Noise,” on the other hand, refers to the irrelevant information or randomness in a dataset.

For example, let’s say you’re modeling height vs. age in children. If you sample a large portion of the population, you’d find a pretty clear relationship:




Height vs. Age (source: CDC)

This is the signal.

However, if you could only sample one local school, the relationship might be muddier. It would be affected by outliers (e.g. kid whose dad is an NBA player) and randomness (e.g. kids who hit puberty at different ages).

**Noise interferes with signal.**

Here's where machine learning comes in. A well functioning ML algorithm will separate the signal from the noise.

If the algorithm is too complex or flexible (e.g. it has too many input features or it's not properly regularized), it can end up "memorizing the noise" instead of finding the signal.

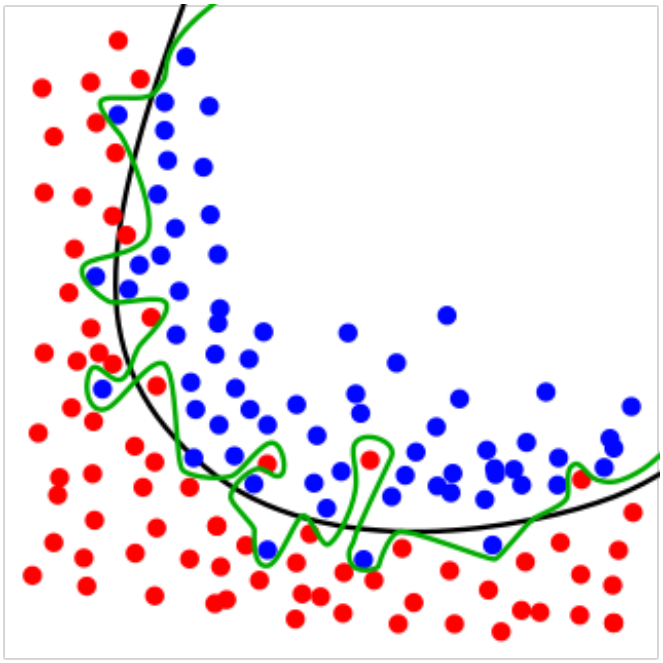
This overfit model will then make predictions based on that noise. It will perform unusually well on its training data… yet very poorly on new, unseen data.

## Goodness of Fit

In statistics, *goodness of fit* refers to how closely a model's predicted values match the observed (true) values.

A model that has learned the noise instead of the signal is considered "overfit" because it fits the training dataset but has poor fit with new datasets.

*While the black line fits the data well, the green line is overfit.*



## Overfitting vs. Underfitting

We can understand overfitting better by looking at the opposite problem, underfitting.

Underfitting occurs when a model is too simple – informed by too few features or regularized too much – which makes it inflexible in learning from the dataset.

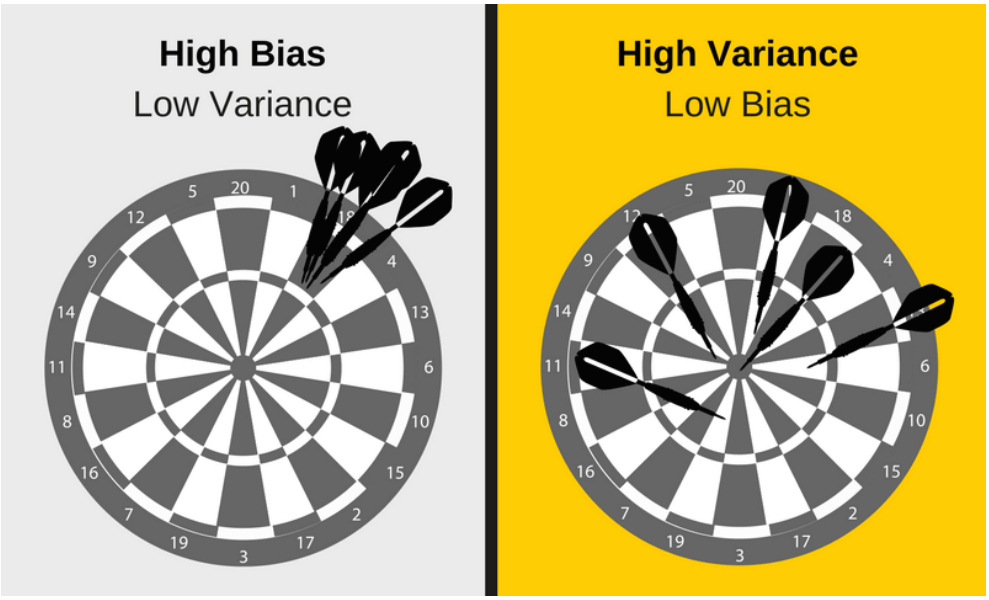
Simple learners tend to have less variance in their predictions but more bias towards wrong outcomes (see: The Bias-Variance Tradeoff).

On the other hand, complex learners tend to have more variance in their predictions.

**Both bias and variance are forms of prediction error in machine learning.**

Typically, we can reduce error from bias but might increase error from variance as a result, or vice versa.

This trade-off between too simple (high bias) vs. too complex (high variance) is a key concept in statistics and machine learning, and one that affects all supervised learning algorithms.

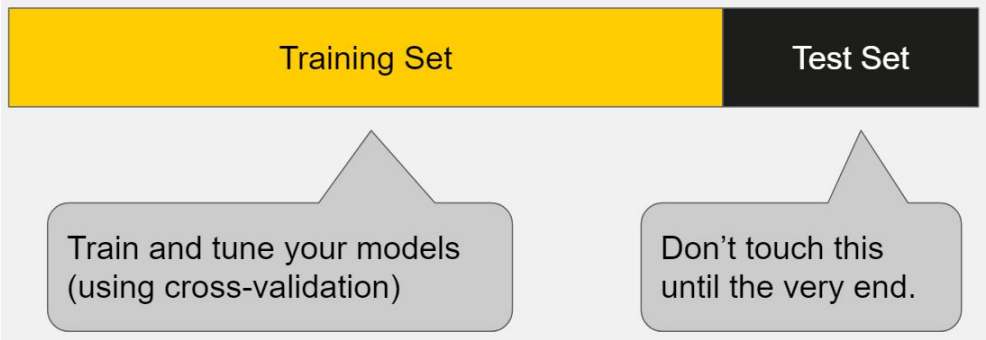



Bias vs. Variance (source: EDS)

## How to Detect Overfitting

A key challenge with overfitting, and with machine learning in general, is that we can't know how well our model will perform on new data until we actually test it.

To address this, we can split our initial dataset into separate *training* and *test* subsets.




Train-Test Split

This method can approximate of how well our model will perform on new data.

**If our model does much better on the training set than on the test set, then we're likely overfitting.**

For example, it would be a big red flag if our model saw 99% accuracy on the training set but only 55% accuracy on the test set.

If you'd like to see how this works in Python, we have a full tutorial for machine learning using Scikit-Learn.

**Another tip is to start with a very simple model to serve as a benchmark.**

Then, as you try more complex algorithms, you'll have a reference point to see if the additional complexity is worth it.

This is the Occam's razor test. If two models have comparable performance, then you should usually pick the simpler one.

## How to Prevent Overfitting

Detecting overfitting is useful, but it doesn't solve the problem. Fortunately, you have several options to try.

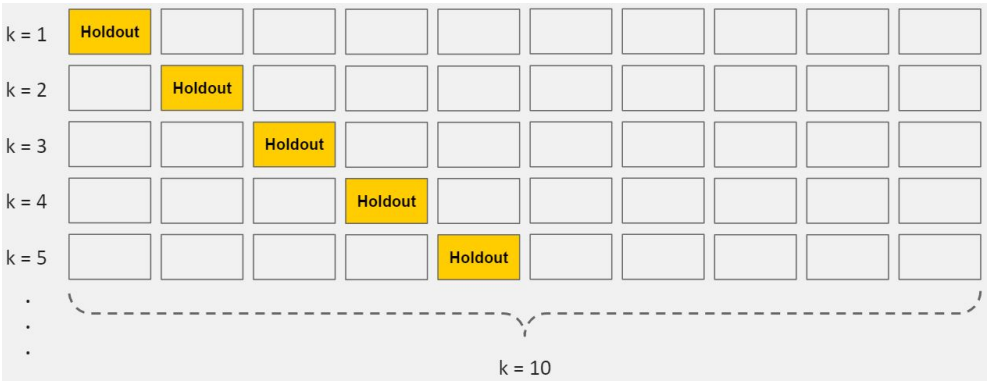
Here are a few of the most popular solutions for overfitting:

### Cross-validation

Cross-validation is a powerful preventative measure against overfitting.

The idea is clever: Use your initial training data to generate multiple mini train-test splits. Use these splits to tune your model.

In standard k-fold cross-validation, we partition the data into k subsets, called folds. Then, we iteratively train the algorithm on k-1 folds while using the remaining fold as the test set (called the "holdout fold").




K-Fold Cross-Validation

Cross-validation allows you to tune hyperparameters with only your original training set.

This allows you to keep your test set as a truly unseen dataset for selecting your final model.

We have another article with a more detailed breakdown of cross-validation.

## Train with more data

It won't work every time, but training with more data can help algorithms detect the signal better. In the earlier example of modeling height vs. age in children, it's clear how sampling more schools will help your model.

Of course, that's not always the case. If we just add more noisy data, this technique won't help. That's why you should always ensure your data is clean and relevant.

## Remove features

Some algorithms have built-in feature selection.

For those that don't, you can manually improve their generalizability by removing irrelevant input features.

An interesting way to do so is to tell a story about how each feature fits into the model. This is like the data scientist's spin on software engineer's rubber duck debugging technique, where they debug their code by explaining it, line-by-line, to a rubber duck.

If anything doesn't make sense, or if it's hard to justify certain features, this is a good way to identify them.
In addition, there are several feature selection heuristics you can use for a good starting point.

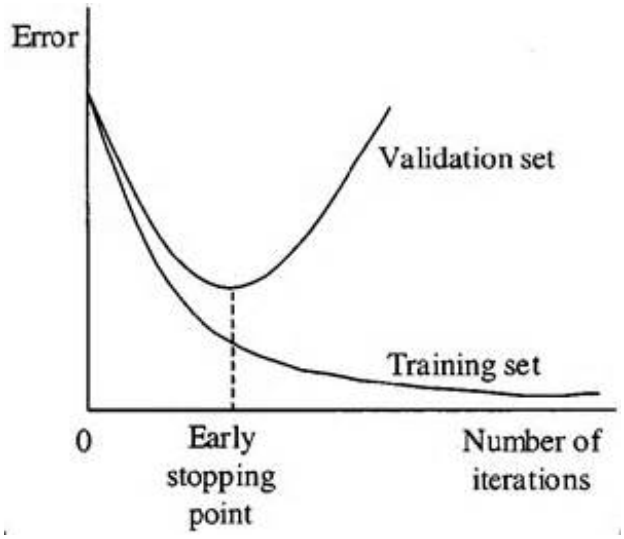
## Early stopping

When you're training a learning algorithm iteratively, you can measure how well each iteration of the model performs.

Up until a certain number of iterations, new iterations improve the model. After that point, however, the model's ability to generalize can weaken as it begins to overfit the training data.

Early stopping refers stopping the training process before the learner passes that point.

Today, this technique is mostly used in deep learning while other techniques (e.g. regularization) are preferred for classical machine learning.




## Regularization

Regularization refers to a broad range of techniques for artificially forcing your model to be simpler.

The method will depend on the type of learner you're using. For example, you could prune a decision tree, use dropout on a neural network, or add a penalty parameter to the cost function in regression.

Oftentimes, the regularization method is a hyperparameter as well, which means it can be tuned through cross-validation.

We have a more detailed discussion here on algorithms and regularization methods.

## Ensembling

Ensembles are machine learning methods for combining predictions from multiple separate models. There are a few different methods for ensembling, but the two most common are:

*Bagging* attempts to reduce the chance overfitting complex models.

- It trains a large number of "strong" learners in parallel.
- A strong learner is a model that's relatively unconstrained.
- Bagging then combines all the strong learners together in order to "smooth out" their predictions.

*Boosting* attempts to improve the predictive flexibility of simple models.

- It trains a large number of "weak" learners in sequence.
- A weak learner is a constrained model (i.e. you could limit the max depth of each decision tree).
- Each one in the sequence focuses on learning from the mistakes of the one before it.

- Boosting then combines all the weak learners into a single strong learner.

While bagging and boosting are both ensemble methods, they approach the problem from opposite directions.

Bagging uses complex base models and tries to "smooth out" their predictions, while boosting uses simple base models and tries to "boost" their aggregate complexity.

## Next Steps

Whew! We just covered quite a few concepts:

- Signal, noise, and how they relate to overfitting.
- Goodness of fit from statistics
- Underfitting vs. overfitting
- The bias-variance tradeoff
- How to detect overfitting using train-test splits
- How to prevent overfitting using cross-validation, feature selection, regularization, etc.

Hopefully seeing all of these concepts linked together helped clarify some of them.

**To truly master this topic, we recommend getting hands-on practice.**

While these concepts may feel overwhelming at first, they will ‘click into place’ once you start seeing them in the context of real-world code and problems.

So here are some additional resources to help you get started:

Now, go forth and learn! (Or have your code do it for you!)

# Python Machine Learning Tutorial, Scikit-Learn: Wine Snob Edition

elitedatascience.com/python-machine-learning-tutorial-scikit-learn

February 10, 2017

In this end-to-end Python machine learning tutorial, you'll learn how to use Scikit-Learn to build and tune a supervised learning model!

We'll be training and tuning a random forest for wine quality (as judged by wine ~~snobs~~ experts) based on traits like acidity, residual sugar, and alcohol concentration.

Before we start, we should state that this guide is meant for beginners who are interested in *applied* machine learning.

Our goal is introduce you to one of the most flexible and useful libraries for machine learning in Python. We'll skip the theory and math in this tutorial, but we'll still recommend great resources for learning those.

**Free: Scikit-Learn Cheatsheet**

Get all of this tutorial's code in a **handy PDF cheatsheet** + plenty of other free cheatsheets, checklists, worksheets, and resource lists in our ***Subscriber Vault.***


## Before we start...

### Recommended Prerequisites

The recommended prerequisites for this guide are at least basic Python programming skills. To move quickly, we'll assume you have this background.

### Why Scikit-Learn for machine learning?

Scikit-Learn, also known as **sklearn**, is Python's premier general-purpose machine learning library. While you'll find other packages that do better at certain tasks, Scikit-Learn's versatility makes it the best starting place for most ML problems.

It's also a fantastic library for beginners because it offers a high-level interface for many tasks (e.g. preprocessing data, cross-validation, etc.). This allows you to better practice the entire machine learning workflow and understand the big picture.

**WTF is machine learning?**

> Ahem… maybe this is a better place to start instead.

**What this guide is not:**

This is not a complete course on machine learning. Machine learning requires the practitioner to make dozens of decisions throughout the entire modeling process, and we won't cover all of those nuances.

Instead, this is a tutorial that will take you from zero to your first Python machine learning model with as little headache as possible!

If you're interested in mastering the theory behind machine learning, then we recommend our free guide:

> How to Learn Machine Learning, The Self-Starter Way

In addition, we also won't be covering *exploratory data analysis* in much detail, which is a vital part of real-world machine learning. We'll leave that for a separate guide.

**A quick tip before we begin:**

This tutorial is designed to be streamlined, and it won't cover any one topic in too much detail. It may be helpful to have the Scikit-Learn documentation open beside you as a supplemental reference.

## Python Machine Learning Tutorial Contents

**Here are the steps for building your first random forest model using Scikit-Learn:**

---

## Step 1: Set up your environment.

First, grab a nice glass of wine.

Drinking wine makes predicting wine easier (probably).

Next, make sure the following are installed on your computer:

- Python 2.7+ or Python 3

- NumPy
- Pandas
- Scikit-Learn (a.k.a. sklearn)

We strongly recommend installing Python through Anaconda (installation guide). It comes with all of the above packages already installed.

If you need to update any of the packages, it's as easy as typing $ conda update <package> from your command line program (Terminal in Mac).

You can confirm Scikit-Learn was installed properly:

Shell

```
1  $ python -c "import sklearn; print sklearn.__version__"
2  0.18.1
```

Great, now let's start a new file and name it **sklearn_ml_example.py**.

## Step 2: Import libraries and modules.

To begin, let's import numpy, which provides support for more efficient numerical computation:

NumPy
Python

```
1  import numpy as np
```

Next, we'll import Pandas, a convenient library that supports dataframes . Pandas is technically optional because Scikit-Learn can handle numerical matrices directly, but it'll make our lives easier:

Pandas
Python

```
1  import pandas as pd
```

Now it's time to start importing functions for machine learning. The first one will be the train_test_split() function from the model_selection module. As its name implies, this module contains many utilities that will help us choose between models.

Import sampling helper
Python

```
1 from sklearn.model_selection import train_test_split
```

Next, we'll import the entire preprocessing module. This contains utilities for scaling, transforming, and wrangling data.

Import preprocessing modules
Python

```
1 from sklearn import preprocessing
```

Next, let's import the *families* of models we'll need... wait, did you just say "families?"

**What's the difference between model "families" and actual models?**

A "family" of models are broad types of models, such as random forests, SVM's, linear regression models, etc. Within each family of models, you'll get an actual model after you fit and tune its parameters to the data.

**Tip: Don't worry too much about this for now... It will make more sense once we get to Step 7.*

We can import the random forest *family* like so:

Import random forest model
Python

```
1 from sklearn.ensemble import RandomForestRegressor
```

For the scope of this tutorial, we'll only focus on training a random forest and tuning its parameters. We'll have another detailed tutorial for how to choose between model families.

For now, let's move on to importing the tools to help us perform *cross-validation*.

Import cross-validation pipeline
Python

```
1 from sklearn.pipeline import make_pipeline
2 from sklearn.model_selection import GridSearchCV
```

Next, let's import some metrics we can use to evaluate our model performance later.

Import evaluation metrics
Python

```
from sklearn.metrics import mean_squared_error, r2_score
```

And finally, we'll import a way to persist our model for future use.

Import module for saving scikit-learn models
Python

```
from sklearn.externals import joblib
```

Joblib is an alternative to Python's pickle package, and we'll use it because it's more efficient for storing large numpy arrays.

**Phew!** That was a lot. Don't worry, we'll cover each function in detail once we get to it. Let's first take a quick sip of wine and toast to our progress... cheers!

## Step 3: Load red wine data.

Alright, now we're ready to load our data set. The Pandas library that we imported is loaded with a whole suite of helpful import/output tools.

You can read data from CSV, Excel, SQL, SAS, and many other data formats. Here's a list of all the Pandas IO tools.

The convenient tool we'll use today is the **read_csv()** function. Using this function, we can load any CSV file, even from a remote URL!

Load wine data from remote URL
Python

```
dataset_url = 'http://mlr.cs.umass.edu/ml/machine-learning-databases/wine-
quality/winequality-red.csv'
data = pd.read_csv(dataset_url)
```

Now let's take a look at the first 5 rows of data:

Output the first 5 rows of data
Python

```
print data.head()
# fixed acidity;"volatile acidity";"citric acid"...
# 0   7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56...
# 1   7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68...
# 2   7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0...
# 3   11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;...
# 4   7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56...
```

Crap... that looks really messy. Upon further inspection, it looks like the CSV file is actually using semicolons to separate the data. That's annoying, but easy to fix:

Read CSV with semicolon separator
Python

```
data = pd.read_csv(dataset_url, sep=';')
print data.head()
#    fixed acidity  volatile acidity  citric acid...
# 0            7.4              0.70         0.00...
# 1            7.8              0.88         0.00...
# 2            7.8              0.76         0.04...
# 3           11.2              0.28         0.56...
# 4            7.4              0.70         0.00...

```

Great, that's much nicer. Now, let's take a look at the data.

Python

```
print data.shape
# (1599, 12)
```

We have 1,599 samples and 12 features, including our target feature. We can easily print some summary statistics.

Summary statistics
Python

```
print data.describe()
#        fixed acidity  volatile acidity  citric acid...
# count    1599.000000       1599.000000  1599.000000...
# mean        8.319637          0.527821     0.270976...
# std         1.741096          0.179060     0.194801...
# min         4.600000          0.120000     0.000000...
# 25%         7.100000          0.390000     0.090000...
# 50%         7.900000          0.520000     0.260000...
# 75%         9.200000          0.640000     0.420000...
# max        15.900000          1.580000     1.000000...
```

Here's the list of all the features:

- **quality** (target)
- fixed acidity
- volatile acidity
- citric acid
- residual sugar
- chlorides

- free sulfur dioxide
- total sulfur dioxide
- density
- pH
- sulphates
- alcohol

All of the features are numeric, which is convenient. However, they have some very different scales, so let's make a mental note to **standardize** the data later.

As a reminder, for this tutorial, we're cutting out a lot of exploratory data analysis we'd typically recommend.

For now, let's move on to splitting the data.

## Step 4: Split data into training and test sets.

Splitting the data into training and test sets **at the beginning of your modeling workflow** is crucial for getting a realistic estimate of your model's performance.

First, let's separate our target (y) features from our input (X) features:

Separate target from training features
Python

```
y = data.quality
X = data.drop('quality', axis=1)
```

This allows us to take advantage of Scikit-Learn's useful **train_test_split** function:

Split data into train and test sets
Python

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                    test_size=0.2,
                                    random_state=123,
                                    stratify=y)
```

As you can see, we'll set aside 20% of the data as a test set for evaluating our model. We also set an arbitrary "random state" (a.k.a. seed) so that we can reproduce our results.

Finally, it's good practice to **stratify your sample** by the target variable. This will ensure your training set looks similar to your test set, making your evaluation metrics more reliable.

## Step 5: Declare data preprocessing steps.

Remember, in Step 3, we made the mental note to *standardize* our features because they were on different scales.

**WTF is standardization?**

Standardization is the process of subtracting the means from each feature and then dividing by the feature standard deviations.

Standardization is a common requirement for machine learning tasks. Many algorithms assume that all features are centered around zero and have approximately the same variance.

**First, here's some code that we *won't use...***

Scikit-Learn makes data preprocessing a breeze. For example, it's pretty easy to simply scale a dataset:

Lazy way of scaling data
Python

```
1  X_train_scaled = preprocessing.scale(X_train)
2  print X_trained_scaled
3  # array([[ 0.51358886,  2.19680282, -0.164433  , ...,  1.08415147,
4  #         -0.69866131, -0.58608178],
5  #        [-1.73698885, -0.31792985, -0.82867679, ...,  1.46964764,
6  #          1.2491516 ,  2.97009781],
7  #        [-0.35201795,  0.46443143, -0.47100705, ..., -0.13658641,
8  # ...
```

You can confirm that the scaled dataset is indeed centered at zero, with unit variance:

Python

```
1  print X_train_scaled.mean(axis=0)
2  # [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
3  print X_train_scaled.std(axis=0)
4  # [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
5
```

Great, but why did we say that we *won't* use this code?

The reason is that we won't be able to perform the exact same transformation on the test set.

Sure, we can still *scale* the test set separately, but we won't be using the same means and standard deviations as we used to transform the training set.

In other words, that means it wouldn't be a fair representation of how the model pipeline, include the preprocessing steps, would perform on brand new data.

**Now, here's the preprocessing code we *will* use...**

So instead of directly invoking the scale function, we'll be using a feature in Scikit-Learn called the **Transformer API.** The Transformer API allows you to "fit" a preprocessing step using the training data the same way you'd fit a model...

...and then use the same transformation on future data sets!

Here's what that process looks like:

1. Fit the transformer on the training set (saving the means and standard deviations)
2. Apply the transformer to the training set (scaling the training data)
3. Apply the transformer to the test set (using the same means and standard deviations)

This makes your final estimate of model performance more realistic, and it allows to insert your preprocessing steps into a **cross-validation pipeline** (more on this in Step 7).

Here's how you do it:

Fitting the Transformer API
Python

```
1  scaler = preprocessing.StandardScaler().fit(X_train)
```

Now, the **scaler** object has the saved means and standard deviations for each feature in the training set.

Let's confirm that worked:

Applying transformer to training data
Python

```
1  X_train_scaled = scaler.transform(X_train)
2  print X_train_scaled.mean(axis=0)
3  # [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
4  print X_train_scaled.std(axis=0)
5  # [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
6
7
```

Note how we're taking the scaler object and using it to **transform** the training set. Later, we can transform the test set using the exact same means and standard deviations used to transform the training set:

Applying transformer to test data
Python

```
X_test_scaled = scaler.transform(X_test)
print X_test_scaled.mean(axis=0)
# [ 0.02776704  0.02592492 -0.03078587 -0.03137977 -0.00471876 -0.04413827
#  -0.02414174 -0.00293273 -0.00467444 -0.10894663  0.01043391]
print X_test_scaled.std(axis=0)
# [ 1.02160495  1.00135689  0.97456598  0.91099054  0.86716698  0.94193125
#  1.03673213  1.03145119  0.95734849  0.83829505  1.0286218 ]


```

Notice how the scaled features in the test set are not perfectly centered at zero with unit variance! This is exactly what we'd expect, as we're transforming the test set using the means from the training set, not from the test set itself.

In practice, when we set up the cross-validation pipeline, we won't even need to manually fit the Transformer API. Instead, we'll simply declare the class object, like so:

Pipeline with preprocessing and model
Python

```
pipeline = make_pipeline(preprocessing.StandardScaler(),
                         RandomForestRegressor(n_estimators=100))
```

This is exactly what it looks like: a **modeling pipeline** that first transforms the data using StandardScaler() and then fits a model using a random forest regressor.

## Step 6: Declare hyperparameters to tune.

Now it's time to consider the hyperparameters that we'll want to tune for our model.

**WTF are hyperparameters?**

There are two types of parameters we need to worry about: *model parameters* and *hyperparameters*. Models parameters can be learned directly from the data (i.e. regression coefficients), while hyperparameters cannot.

Hyperparameters express "higher-level" structural information about the model, and they are typically set before training the model.

**Example: random forest hyperparameters.**

As an example, let's take our random forest for regression:

Within each decision tree, the computer can empirically decide where to create branches based on *either* mean-squared-error (MSE) or mean-absolute-error (MAE). Therefore, the actual branch locations are **model parameters**.

However, the algorithm does not know *which* of the two criteria, MSE or MAE, that it should use. The algorithm also cannot decide how many trees to include in the forest. These are examples of **hyperparameters** that the user must set.

We can list the tunable hyperparameters like so:

List tunable hyperparameters
Python

```
print pipeline.get_params()
# ...
# 'randomforestregressor__criterion': 'mse',
# 'randomforestregressor__max_depth': None,
# 'randomforestregressor__max_features': 'auto',
# 'randomforestregressor__max_leaf_nodes': None,
# ...
```

You can also find a list of all the parameters on the RandomForestRegressor documentation page. Just note that when it's tuned through a *pipeline*, you'll need to prepend  randomforestregressor__ before the parameter name, like in the code above.

Now, let's declare the hyperparameters we want to tune through cross-validation.

Declare hyperparameters to tune
Python

```
hyperparameters = { 'randomforestregressor__max_features' : ['auto', 'sqrt', 'log2'],
                    'randomforestregressor__max_depth': [None, 5, 3, 1]}
```

As you can see, the format should be a Python dictionary (data structure for key-value pairs) where keys are the hyperparameter names and values are lists of settings to try. The options for parameter values can be found on the documentation page.

## Step 7: Tune model using a cross-validation pipeline.

Now we're almost ready to dive into fitting our models. But first, we need to spend some time talking about cross-validation.

This is one of the most important skills in all of machine learning because it helps you maximize model performance while *reducing the chance of overfitting*.

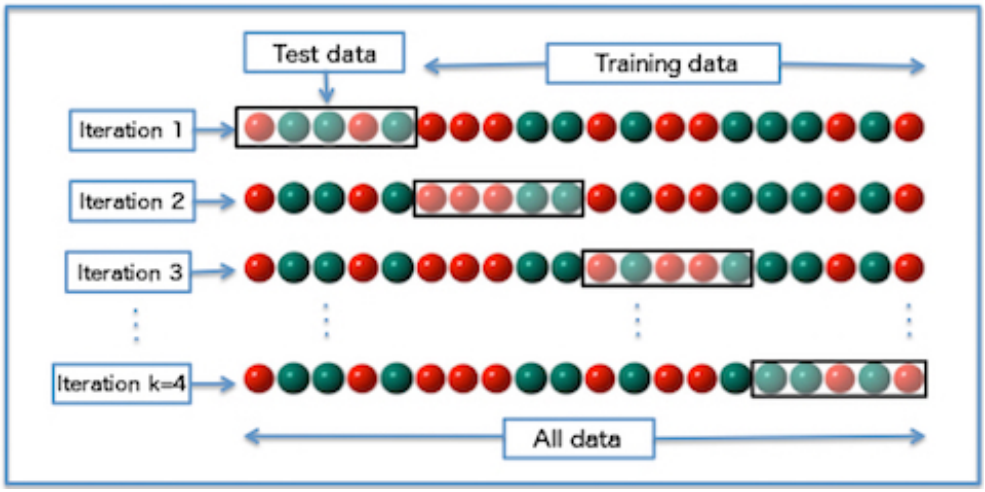
**WTF is cross-validation (CV)?**

Cross-validation is a process for reliably estimating the performance of **a method** for building a model by training and evaluating your model multiple times using the same method.

Practically, that "method" is simply a set of hyperparameters in this context.

These are the steps for CV:

1. Split your data into *k* equal parts, or "folds" (typically *k*=10).
2. Train your model on *k*-1 folds (e.g. the first 9 folds).
3. Evaluate it on the remaining "hold-out" fold (e.g. the 10th fold).
4. Perform steps (2) and (3) *k* times, each time holding out a different fold.
5. Aggregate the performance across all *k* folds. This is your performance metric.




K-Fold Cross-Validation diagram, courtesy of Wikipedia

**Why is cross-validation important in machine learning?**

Let's say you want to train a random forest regressor. One of the hyperparameters you must tune is the maximum depth allowed for each decision tree in your forest.

How can you decide?

That's where cross-validation comes in. Using only your training set, you can use CV to evaluate different hyperparameters and estimate their effectiveness.

This allows you to keep your test set "untainted" and save it for a true hold-out evaluation when you're finally ready to select a model.

For example, you can use CV to tune a random forest model, a linear regression model, and a k-nearest neighbors model, *using only the training set*. Then, you still have the untainted test set to make your final selection between the model families!

**So WTF is a cross-validation "pipeline?"**

The best practice when performing CV is to include your data preprocessing steps *inside* the cross-validation loop. This prevents accidentally tainting your training folds with influential data from your test fold.

Here's how the CV pipeline looks after including preprocessing steps:

1. Split your data into *k* equal parts, or "folds" (typically *k*=10).
2. **Preprocess *k*-1 training folds.**
3. Train your model on the same *k*-1 folds.
4. **Preprocess the hold-out fold using the same transformations from step (2).**
5. Evaluate your model on the same hold-out fold.
6. Perform steps **(2) - (5)** *k* times, each time holding out a different fold.
7. Aggregate the performance across all *k* folds. This is your performance metric.

Fortunately, Scikit-Learn makes it stupidly simple to set this up:

Sklearn cross-validation with pipeline
Python

```
clf = GridSearchCV(pipeline, hyperparameters, cv=10)
# Fit and tune model
clf.fit(X_train, y_train)

```

Yes, it's really that easy. **GridSearchCV** essentially performs cross-validation across the entire "grid" (all possible permutations) of hyperparameters.

It takes in your model (in this case, we're using a model *pipeline*), the hyperparameters you want to tune, and the number of folds to create.

Obviously, there's a lot going on under the hood. We've included the pseudo-code above, and we'll cover writing cross-validation from scratch in a separate guide.

Now, you can see the best set of parameters found using CV:

Python

```
print clf.best_params_
# {'randomforestregressor__max_depth': None, 'randomforestregressor__max_features': 'auto'}
```

Interestingly, it looks like the default parameters win out for this data set.

**Tip: It turns out that in practice, random forests don't actually require a lot of tuning. They tend to work pretty well out-of-the-box with a reasonable number of trees. Even so, these same steps can be used when building any type of supervised learning model.*

## Step 8: Refit on the entire training set.

After you've tuned your hyperparameters appropriately using cross-validation, you can generally get a small performance improvement by refitting the model on the entire training set.

Conveniently, GridSearchCV from sklearn will automatically refit the model with the best set of hyperparameters using the entire training set.

This functionality is ON by default, but you can confirm it:

Confirm model will be retrained
Python

```
print clf.refit
# True
```

Now, you can simply use the clf object as your model when applying it to other sets of data. That's what we'll be doing in the next step.

## Step 9: Evaluate model pipeline on test data.

Alright, we're in the home stretch!

This step is really straightforward once you understand that the clf object you used to tune the hyperparameters can also be used directly like a model object.

Here's how to predict a new set of data:

Predict a new set of data
Python

```
y_pred = clf.predict(X_test)
```

Now we can use the metrics we imported earlier to evaluate our model performance.

Python

```
print r2_score(y_test, y_pred)
# 0.45044082571584243
print mean_squared_error(y_test, y_pred)
# 0.35461593750000003

```

Great, so now the question is... **is this performance good enough?**

Well, the rule of thumb is that your very first model probably won't be the best possible model. However, we recommend a combination of three strategies to decide if you're satisfied with your model performance.

1. Start with the goal of the model. If the model is tied to a business problem, have you successfully solved the problem?
2. Look in academic literature to get a sense of the current performance benchmarks for specific types of data.
3. Try to find low-hanging fruit in terms of ways to improve your model.

There are various ways to improve a model. We'll have more guides that go into detail about how to improve model performance, but here are a few quick things to try:

1. Try other regression model families (e.g. regularized regression, boosted trees, etc.).
2. Collect more data if it's cheap to do so.
3. Engineer smarter features after spending more time on exploratory analysis.
4. Speak to a domain expert to get more context (...this is a good excuse to go wine tasting!).

As a final note, when you try other families of models, we recommend using the same training and test set as you used to fit the random forest model. That's the best way to get a true apples-to-apples comparison between your models.

## Step 10: Save model for future use.

Great job completing this tutorial!

You've done the hard part, and deserve another glass of wine. Maybe this time you can use your shiny new predictive model to select the bottle.

But before you go, let's save your hard work so you can use the model in the future. It's really easy to do so:

Save model to a .pkl file
Python

```
1  joblib.dump(clf, 'rf_regressor.pkl')
```

And that's it. When you want to load the model again, simply use this function:

Load model from .pkl file
Python

```
clf2 = joblib.load('rf_regressor.pkl')
# Predict data set using loaded model
clf2.predict(X_test)

```

Congratulations, you've reached the end of this tutorial!

We've just completed a whirlwind tour of Scikit-Learn's core functionality, but we've only really scratched the surface. Hopefully you've gained some guideposts to further explore all that sklearn has to offer.

For continued learning, we recommend studying other examples in sklearn.

## The complete code, from start to finish.

Here's all the code in one place, in a single script.

Python

```
# 2. Import libraries and modules
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.externals import joblib
# 3. Load red wine data.
dataset_url = 'http://mlr.cs.umass.edu/ml/machine-learning-databases/wine-
quality/winequality-red.csv'
data = pd.read_csv(dataset_url, sep=';')
# 4. Split data into training and test sets
y = data.quality
X = data.drop('quality', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                    test_size=0.2,
                                    random_state=123,
                                    stratify=y)
# 5. Declare data preprocessing steps
pipeline = make_pipeline(preprocessing.StandardScaler(),
                  RandomForestRegressor(n_estimators=100))
# 6. Declare hyperparameters to tune
hyperparameters = { 'randomforestregressor__max_features' : ['auto', 'sqrt', 'log2'],
              'randomforestregressor__max_depth': [None, 5, 3, 1]}
# 7. Tune model using cross-validation pipeline
clf = GridSearchCV(pipeline, hyperparameters, cv=10)
clf.fit(X_train, y_train)
# 8. Refit on the entire training set
# No additional code needed if clf.refit == True (default is True)
# 9. Evaluate model pipeline on test data
pred = clf.predict(X_test)
print r2_score(y_test, pred)
print mean_squared_error(y_test, pred)
# 10. Save model for future use
joblib.dump(clf, 'rf_regressor.pkl')
# To load: clf2 = joblib.load('rf_regressor.pkl')
```

# The 5 Levels of Machine Learning Iteration

elitedatascience.com/machine-learning-iteration

February 15, 2017

Can you guess the answer to this riddle?

- If you've studied machine learning, you've seen this everywhere…
- If you're a programmer, you've done this a thousand times…
- If you've practiced any skill, this is already second-nature for you…

Nope, it's not overdosing on coffee… It's… **iteration**!

Yes, iteration as in *repeating a set of tasks to achieve a result.*

Wait, isn't that just… the dictionary definition? Well, yes it is. And yes, that's all we really mean. And no, we're not about to reveal some mind-blowing secret about it.

Yep, this is what we're making a fuss about.

But we do hope to frame this simple concept in a way that might be new to you. Our goal is to walk through a **tour of several essential concepts** in ML, but to do so from a different perspective than the common approach taught in textbooks.

You see, most books focus on the **sequential** process for machine learning: load data, then preprocess it, then fit models, then make predictions, etc.

This sequential approach is certainly reasonable and helpful to see, but real-world machine learning is rarely so linear. Instead, practical machine learning has a distinct **cyclical** nature that demands constant iteration, tuning, and improvement.

Therefore, we hope to showcase how the simple technique of iteration is actually very beautiful and profound in the context of machine learning. This post is intended for beginners, but more experienced users can enjoy it as well.

## Why make a fuss about iteration?

Iteration is a central concept of machine learning, and it's vital on many levels. Knowing *exactly* where this simple concept appears in the ML workflow has many practical benefits:

1. You'll better understand the algorithms you work with.
2. You'll anticipate more realistic timelines for your projects.
3. You'll spot low hanging fruit for model improvement.

4. You'll find it easier to stay motivated after poor initial results.
5. You'll be able to solve bigger problems with machine learning.

From our experience, seeing the ML workflow from the perspective of iteration can really help beginners see the big picture concepts behind machine learning.

So without further ado, let's begin our tour of the 5 levels of machine learning iteration.

### Table of Contents

## The Model Level: Fitting Parameters

The first level where iteration plays a big role is at the model level. Any model, whether it be a regression model, a decision tree, or a neural network, is defined by many (sometimes even millions) of model **parameters**.

For example, a regression model is defined by its feature coefficients, a decision tree is defined by its branch locations, and a neural network is defined by the weights connecting its layers.

But how does the machine *learn* the right values for all of the model parameters? Here's where iterative algorithms come into play!

### Fitting Parameters with Gradient Descent

One of the shining successes in machine learning is the **gradient descent** algorithm (and its modified counterpart, stochastic gradient descent).

Gradient descent is an iterative method for finding the minimum of a function. In machine learning, that function is typically the **loss (or cost) function.** "Loss" is simply some metric that quantifies the cost of wrong predictions.

Gradient descent calculates the loss achieved by a model with a given set of parameters, and then alters those parameters to reduce the loss. It repeats this process until that loss can't substantially be reduced further.

The final set of parameters that minimize the loss now define your **fitted model**.

### Gradient Descent Intuition

We won't derive the math behind gradient descent here, but we'll paint a picture of the intuition:

1. Imagine a mountain range with hills and valleys (**loss function**).
2. Each location (**parameter set**) on the mountain has an altitude (**loss**).
3. Now drop a ball somewhere on a mountain (**initialization**).
4. At any moment, the ball rolls in the steepest direction (**the gradient**).

5. It continue to roll (**iteration**) until it gets stuck in a valley (**local minimum**).
6. Ideally, you want to find the lowest possible valley (**global minimum**).
7. There are clever ways to prevent the ball from being stuck in local minima (e.g. initializing multiple balls, giving it more momentum so it can traverse small hills, etc.)
8. Oh yeah, and if the mountain terrain is shaped like a bowl (**convex function**), then the ball is guaranteed to reach the lowest point.

Here's a great short video from Andrew Ng further explaining the intuition behind Gradient Descent.

To learn more about the math behind gradient descent, we recommend these resources:

- Lecture slides from UCLA on mathematical optimization
- Notes on mathematical optimization from scipy-lectures.org

In practice, especially when using existing ML implementations like Scikit-Learn, you won't need to implement gradient descent from scratch.

## The Micro Level: Tuning Hyperparameters

The next level where iteration plays a huge role is at what we named the "micro" level, more commonly known as the *general* model or model *family*.

You can think of a **model family** as broad category of models with customizable structures. Logistic regressions, decision trees, SVMs, and neural networks are actually all different *families* of models. Each model family has a set of structural choices you must make before actually fitting the model parameters.

For example, within the logistic regression *family*, you can build separate *models* using either L1 or L2 regularization penalties. Within the decision tree *family*, you can have different *models* with different structural choices such as the depth of the tree, pruning thresholds, or even the splitting criteria.

These structural choices are called *hyperparameters.*

### Why Hyperparameters are Special

**Hyperparameters** are "higher-level" parameters that cannot be learned directly from the data using gradient descent or other optimization algorithms. They describe structural information about a model that must be decided before fitting model parameters.

So when people say they are going to "train a logistic regression model," what they really mean is a two-stage process.

1. First, decide *hyperparameters* for the model family: e.g. Should the model have an

L1 or L2 penalty to prevent overfitting?
2. Then, fit the model *parameters* to the data: e.g. What are the model coefficients that minimize the loss function?

We discussed earlier how gradient descent can help perform Step 2. But in order fit model parameters using gradient descent, the user must first set the hyperparameters from model family.

So how can we tackle the Step 1, finding the best hyperparameters for the model family?

## Tuning Hyperparameters with Cross-Validation.

**Cross-validation** is one of those techniques that works in so many scenarios that you'll almost feel like you're cheating when you use it.

In this context, cross-validation is an iterative method for evaluating the performance of models built with a given set of hyperparameters. It's a clever way to reuse your training data by dividing it into parts and cycling through them (pseudocode below).

With cross-validation, you can fit and evaluate models with various sets of hyperparameters using only your training data. That means you can save the test set as a true untainted hold-out set for your final model selection (more on this in the next section).

Here's a short and sweet video explaining the idea behind the most popular form of cross-validation, **k-fold cross-validation**.

## Cross-Validation Step-by-Step

These are the steps for selecting hyperparameters using 10-fold cross-validation:

1. Split your training data into 10 equal parts, or "folds."
2. From all sets of hyperparameters you wish to consider, choose a *set of hyperparameters.*
3. Train your model with that *set of hyperparameters*on the first 9 folds.
4. Evaluate it on the 10th fold, or the"hold-out" fold.
5. Repeat steps (3) and (4) 10 times with the same *set of hyperparameters*, each time holding out a different fold.
6. Aggregate the performance across all 10 folds. This is your performance metric for the *set of hyperparameters*.
7. Repeat steps (2) to (6) for all sets of hyperparameters you wish to consider.

Here's how that looks in pseudocode:

Pseudocode for cross-validation

```
all_folds = split_into_k_parts(all_training_data)
for set_p in hyperparameter_sets:
    model = InstanceFromModelFamily()
    for fold_k in all_folds:
        training_folds = all_folds besides fold_k
        fit model on training_folds using set_p
        fold_k_performance = evaluate model on fold_k
    set_p_performance = average all k fold_k_performances for set_p
select set from hyperparameter_sets with best set_p_performance
```

## The Macro Level: Solving Your Problem

Now we're going to step away from individual models and even model families. We're going to discuss iteration at the **problem-solving level**.

Often, the first model you build for a problem won't be the best possible, even if you tune it perfectly with cross-validation. That's because fitting model parameters and tuning hyperparameters are only two parts of the entire machine learning problem-solving workflow.

There are several other iterative techniques that you can leverage to find the best performing solution. We consider these next 2 techniques to be low-hanging-fruit for improving your predictive performance.

### Trying Different Model Families

There's a concept in machine learning called the **No Free Lunch theorem.** There are different interpretations of the NFL theorem (not to be confused with National Football League), but the one we care about states: *There is no one model family that works best for every problem.*

Depending on a variety of factors such as the type of data, problem domain, sparsity of data, and even the amount of data you've collected, different model families will perform better.

Therefore, one of the easiest ways to improve your solution for a given problem is to try several different model families. This level of iteration sits nicely above the previous level.

Here's how that looks in pseudocode:

Pseudocode for selecting model family

```
training_data, test_data = randomly_split(all_data)
list_of_families = logistic regression,
                   decision tree,
                   SVM,
                   neural network, etc...
for model_family in list_of_families:
    best_model = tuned with cross-validation on training_data
evaluate best_model from each model_family on test_data
select final model
```

Note that the cross-validation step is the same as the one in the previous section. This beautiful form of **nested iteration** is an effective way of solving problems with machine learning.

## Ensembling Models

The next way to improve your solution is by combining multiple models into an **ensemble.** This is a direct extension from the iterative process needed to fit those models.

We'll save a detailed discussion of ensemble methods for a different post, but a common form of ensembling is simply averaging the predictions from your multiple models. Of course, there are more advanced ways to combine your models, but the iteration needed to fit multiple models is the same.

This combined prediction will often see a small performance increase over any of the individual models. Here's the pseudocode for building a simple ensemble model:

Pseudocode for simple ensembling

```
training_data, test_data = randomly_split(all_data)
list_of_families = logistic regression,
                   decision tree,
                   SVM,
                   neural network, etc...
for model_family in list_of_families:
    best_model = tuned with cross-validation on training_data
average predictions by best_model from each model_family
... profit! (often)
```

Note how most of that process is exactly the same as the previous technique!

Practically, that means you can easily double-up on these two techniques. First, you can build the best model from a variety of different model families. Then you can ensemble them. Finally, you can evaluate the individual models *and* the ensemble model on the same test set.

As a final word of caution: You should always keep an untainted test set to select your final model. We recommend splitting your data into train and test sets at the very beginning of your modeling process. Don't touch the test set until the very end.

## The Meta Level: Improving Your Data

**Better data beats better algorithms.** That doesn't always mean *more* data beats better algorithms. Yes, better data often implies more data, but it also implies cleaner data, more relevant data, and better features engineered from the data.

Improving your data is also an iterative process. As you tackle larger challenges with machine learning, you'll realize that it's pretty damn hard to get your data completely right from the start.

Maybe there's some key feature that you didn't think to collect. Maybe you didn't collect enough data. Maybe you misunderstood one of the columns in the dataset and need to circle back with a colleague to explain it.

A great machine learning practitioner always keeps an open mind toward continuously improving the dataset.

### Collecting Better Data

The ability to collect better data is a skill that develops with time, experience, and more domain expertise. For example, if you're building a real estate pricing model, you should collect every bit of information about the house itself, the nearby neighborhood, and even past property tax payments that are publicly available.

Another element of better data is the overall **cleanliness** of the data. That means having less missing data, lower measurement error, and doing your best to replace proxy metrics with primary metrics.

Here are a few questions to ask yourself that can spark ideas for improving your dataset:

- Are you collecting all the features that you need?
- Can you clean the data better?
- Can you reduce measurement error?
- Are there outliers that can be removed?
- Is it cheap to collect more data?

### Engineering Better Features

**Feature engineering**, or creating new features from the data by leveraging domain knowledge, is one of the most valuable activities you can do to improve your models.

It's often difficult and time-consuming, but it's considered essential in applied machine learning. Therefore, as a machine learning practitioner, it is your duty to continue learning about your chosen domain.

That's because as you learn more about the domain, you'll develop better intuition around the types of features that are most impactful. You should treat this is as an iterative process that improves alongside your growth in personal expertise.

## The Human Level: Improving Yourself

Now we've arrived at the **most important** level of iteration in machine learning: the human level. Even if you forget everything else from this post, we hope you take away the lesson from this section.

Here's the truth: machine learning and data science are big and hairy topics. Especially if you're a beginner, you may be feeling overwhelmed with all there is to learn. There are so many moving pieces, and new developments are happening every day.

And you know what? Parts of ML are still very tough and confusing for us. But that's OK, because we strongly believe the most important level of iteration is at the human level, **the machine learning practitioner.**

So we want to conclude this lengthy post with a few parting suggestions. We hope that this last section can help you keep things in perspective and feel less overwhelmed by the information overload in this field.

### #1. Never stop learning.

As you can see, iteration is built into every layer of the machine learning process. Your personal skills are no exception. Machine learning is a deep and rich field, and everything will become easier the more you practice.

### #2. Don't expect perfection from the start.

You don't need to win your very first Kaggle competition. And it's fine if you build a model and find out it completely sucks. The most valuable treasure is your personal growth and improvement, and that should be your main focus.

### #3. It's OK to not know everything.

In fact, it's almost impossible to know *everything* about ML. The key is to build a foundation that will help you pick up new algorithms and techniques as you need them. And you guessed it... understanding **iteration** is part of that foundation.

### #4. Try everything at least twice.

Struggling with an algorithm or task? Spending much longer than you thought it would take? No problem, just remember to try it at least one more time. Everything is easier and faster on the second try, and this is the best way to *see* your progress.

### #5. Cycle between theory, practice, and projects.

We believe the most effective way to learn machine learning is by cycling between theory, targeted practice, and larger projects. This is the fastest way to master the theory while developing practical, real-world skills. You can learn more about this approach from our free guide: How to Learn Machine Learning, The Self-Starter Way

## Summary of Iteration in Machine Learning

Iteration is a simple concept, yet beautiful in its application. It glues machine learning together on every level.

Summary of iteration in real-world machine learning

```
1  Human Level: Repeatedly practice to improve your skills.
2    Meta Level: Continue to improve your data and features.
3      Macro Level: Explore different model families and ensembles.
4        Micro Level: Cross-validation to tune model hyperparameters.
5          Model Level: Gradient descent to fit model parameters.
```

# The Ultimate Python Seaborn Tutorial: Gotta Catch 'Em All

elitedatascience.com/python-seaborn-tutorial

May 2, 2017

In this step-by-step Seaborn tutorial, you'll learn how to use one of Python's most convenient libraries for data visualization.

For those who've tinkered with Matplotlib before, you may have wondered, "why does it take me 10 lines of code just to make a decent-looking histogram?"

Well, if you're looking for a simpler way to plot attractive charts, then you'll love Seaborn. We'll walk you through everything you need to know to get started, and we'll use a fun Pokémon dataset (which you can download below).

**Free: Seaborn Cheatsheet**

Get all of this tutorial's code in a **handy PDF cheatsheet** + plenty of other free cheatsheets, checklists, worksheets, and resource lists in our ***Subscriber Vault.***


## Introduction to Seaborn

Seaborn provides a high-level interface to Matplotlib, a powerful but sometimes unwieldy Python visualization library.

On Seaborn's official website, they state:

> If matplotlib "tries to make easy things easy and hard things possible", seaborn tries to make a well-defined set of hard things easy too.

We've found this to be a pretty good summary of Seaborn's strengths. In practice, the "well-defined set of hard things" includes:

- Using default themes that are aesthetically pleasing.

- Setting custom color palettes.
- Making attractive statistical plots.
- Easily and flexibly displaying distributions.
- Visualizing information from matrices and DataFrames.

Those last three points are why **Seaborn is our tool of choice for Exploratory Analysis.** It makes it very easy to “get to know” your data quickly and efficiently.

However, Seaborn is a complement, not a substitute, for Matplotlib. There are some tweaks that still require Matplotlib, and we’ll cover how to do that as well.

### How to Learn Seaborn, the Self-Starter Way:

While Seaborn simplifies data visualization in Python, it still has many features. Therefore, the best way to learn Seaborn is to *learn by doing*.

- 1
  Each library approaches data visualization differently, so it's important to understand how Seaborn "thinks about" the problem.
- 2
  Learning in context is the best way to master a new skill quickly.
- 3
  Since you've already learned the library's paradigms and had some hands-on practice, you'll easily find what you need.

This process will give you intuition about what you can do with Seaborn, leaving documentation to serve as further guidance. This is the fastest way to go from zero to proficient.

**A quick tip before we begin:**

We tried to make this tutorial as streamlined as possible, which means we won’t go into too much detail for any one topic. It’s helpful to have the Seaborn documentation open beside you, in case you want to learn more about a feature.

## Seaborn Tutorial Contents

Instead of just showing you how to make a bunch of plots, we’re going to walk through the most important paradigms of the Seaborn library. Along the way, we’ll illustrate each concept with examples.

**Here are the steps we’ll cover in this tutorial:**

---

## Step 1: Installing Seaborn.

First, things first: **Let's. Get. Pumped. Up!**

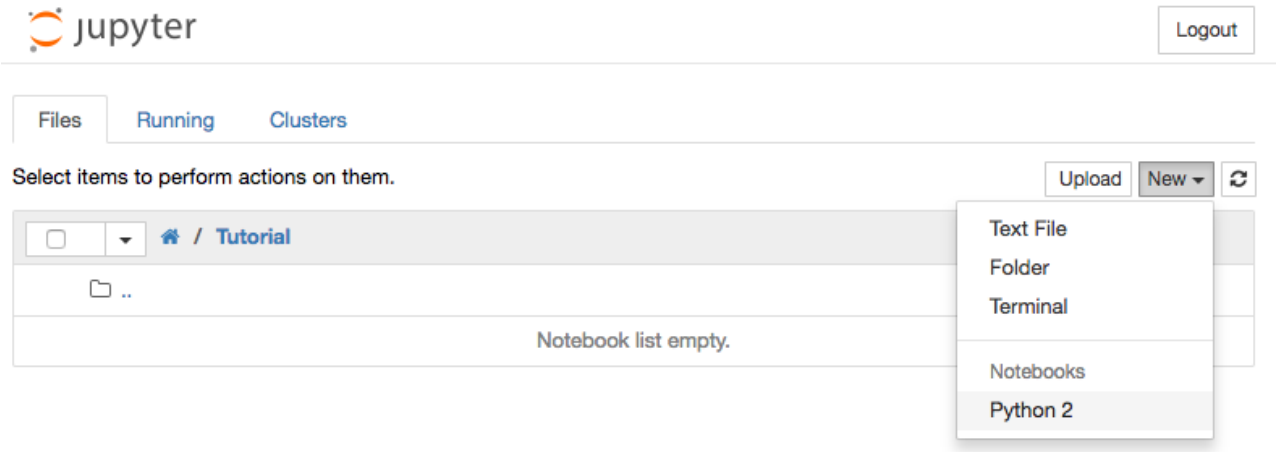
(*Yes... We totally looped that while writing this tutorial...*)

Next, make sure you have the following installed on your computer:

- Python 2.7+ or Python 3
- Pandas
- Matplotlib
- Seaborn
- Jupyter Notebook (optional, but recommended)

We strongly recommend installing the Anaconda Distribution, which comes with all of those packages. Simply follow the instructions on that download page.

Once you have Anaconda installed, simply start Jupyter (either through the command line or the Navigator app) and open a new notebook:




## Step 2: Importing libraries and dataset.

Let's start by importing Pandas, which is a great library for managing relational (i.e. table-format) datasets:

Pandas
Python

```
1  # Pandas for managing datasets
2  import pandas as pd
```

Next, we'll import Matplotlib, which will help us customize our plots further.

> **Tip:** In Jupyter Notebook, you can also include %matplotlib inline to display your plots inside your notebook.

Matplotlib
Python

```
# Matplotlib for additional customization
from matplotlib import pyplot as plt
%matplotlib inline
```

Then, we'll import the Seaborn library, which is the star of today's show.

Seaborn
Python

```
# Seaborn for plotting and styling
import seaborn as sns
```

Now we're ready to import our dataset.

> **Tip:** we gave each of our imported libraries an **alias**. Later, we can invoke Pandas with pd, Matplotlib with plt, and Seaborn with sns.

Today, we'll be using a cool Pokémon dataset (first generation). Here's the free download:

Pokemon.csv
Dataset for this tutorial.

Once you've downloaded the CSV file, you can import it with Pandas.

> **Tip:** The argument  index_col=0 simply means we'll treat the first column of the dataset as the ID column.

Import dataset
Python

```
# Read dataset
df = pd.read_csv('Pokemon.csv', index_col=0)
```

Here's what the dataset looks like:

Example observations
Python

```
# Display first 5 observations
df.head()
```

| # | Name | Type 1 | Type 2 | Total | HP | Attack | Defense | Sp. Atk | Sp. Def | Speed | Stage | Legendary |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Bulbasaur | Grass | Poison | 318 | 45 | 49 | 49 | 65 | 65 | 45 | 1 | False |
| 2 | Ivysaur | Grass | Poison | 405 | 60 | 62 | 63 | 80 | 80 | 60 | 2 | False |
| 3 | Venusaur | Grass | Poison | 525 | 80 | 82 | 83 | 100 | 100 | 80 | 3 | False |
| 4 | Charmander | Fire | NaN | 309 | 39 | 52 | 43 | 60 | 50 | 65 | 1 | False |
| 5 | Charmeleon | Fire | NaN | 405 | 58 | 64 | 58 | 80 | 65 | 80 | 2 | False |

As you can see, we have combat stats data for the original 151 (a.k.a best 151) Pokémon.

## Step 3: Seaborn's plotting functions.

One of Seaborn's greatest strengths is its diversity of plotting functions. For instance, making a **scatter plot** is just one line of code using the lmplot() function.

There are two ways you can do so.

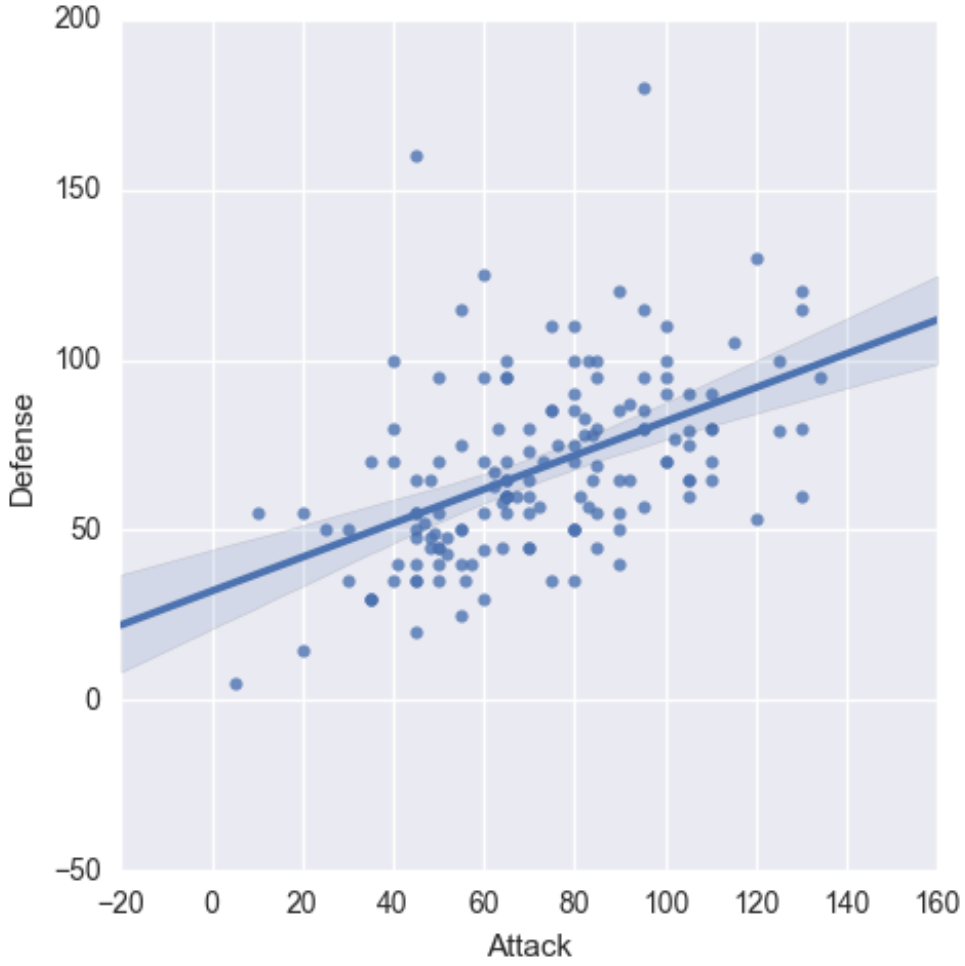
- The first way (recommended) is to pass your DataFrame to the data= argument, while passing column names to the axes arguments, x= and y=.
- The second way is to directly pass in Series of data to the axes arguments.

For example, let's compare the Attack and Defense stats for our Pokémon:

Default Scatterplot
Python

```
# Recommended way
sns.lmplot(x='Attack', y='Defense', data=df)
# Alternative way
# sns.lmplot(x=df.Attack, y=df.Defense)

```



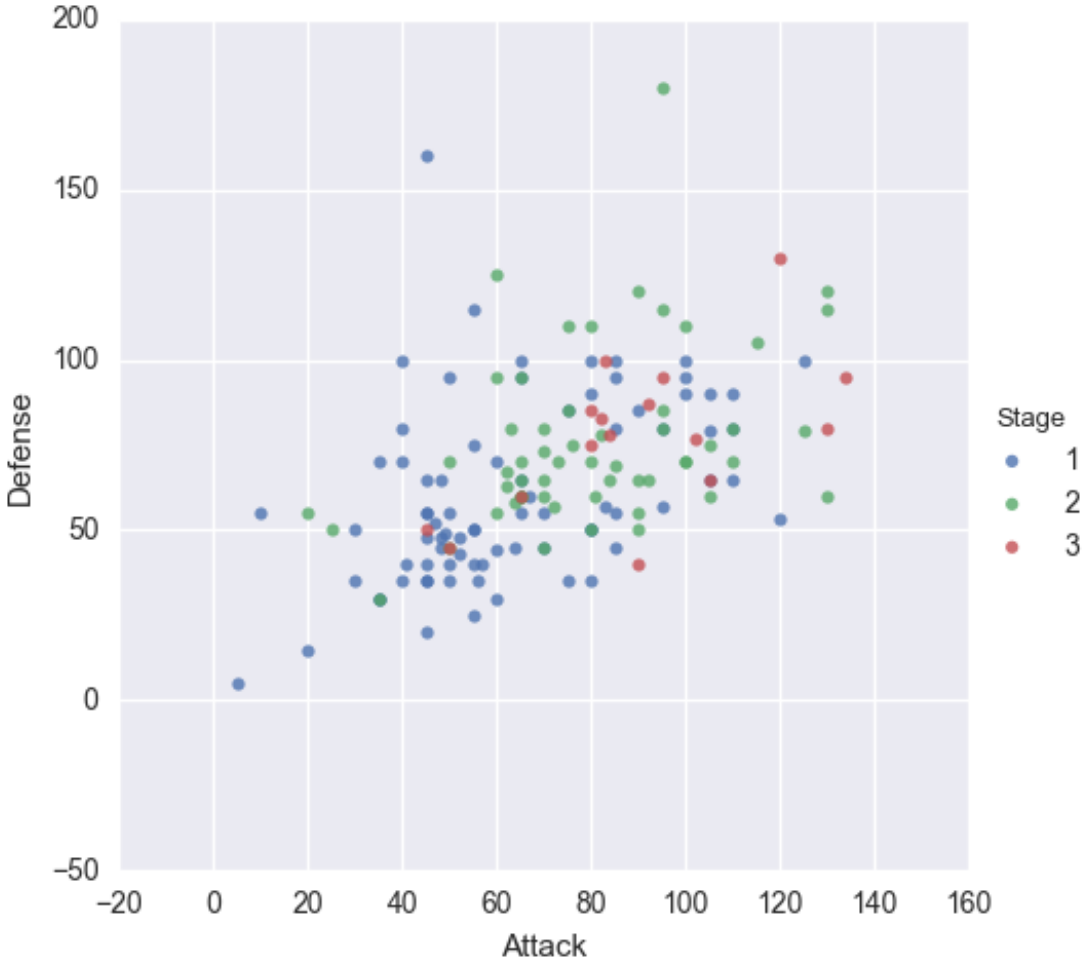

By the way, Seaborn doesn't have a dedicated scatter plot function, which is why you see a diagonal line. We actually used Seaborn's function for fitting and plotting a **regression line**.

Thankfully, each plotting function has several useful options that you can set. Here's how we can tweak the lmplot():

- First, we'll set fit_reg=False to remove the regression line, since we only want a scatter plot.
- Then, we'll set hue='Stage' to color our points by the Pokémon's evolution stage. This **hue** argument is very useful because it allows you to express a third dimension of information using color.

Scatterplot parameters
Python

```
# Scatterplot arguments
sns.lmplot(x='Attack', y='Defense', data=df,
           fit_reg=False, # No regression line
           hue='Stage')   # Color by evolution stage
```




Looking better, but we can improve this scatter plot further. For example, all of our Pokémon have positive Attack and Defense values, yet our **axes limits** fall below zero. Let's see how we can fix that...

## Step 4: Customizing with Matplotlib.

Remember, Seaborn is a high-level interface to Matplotlib. From our experience, Seaborn will get you *most* of the way there, but you'll sometimes need to bring in Matplotlib.

Setting your axes limits is one of those times, but the process is pretty simple:

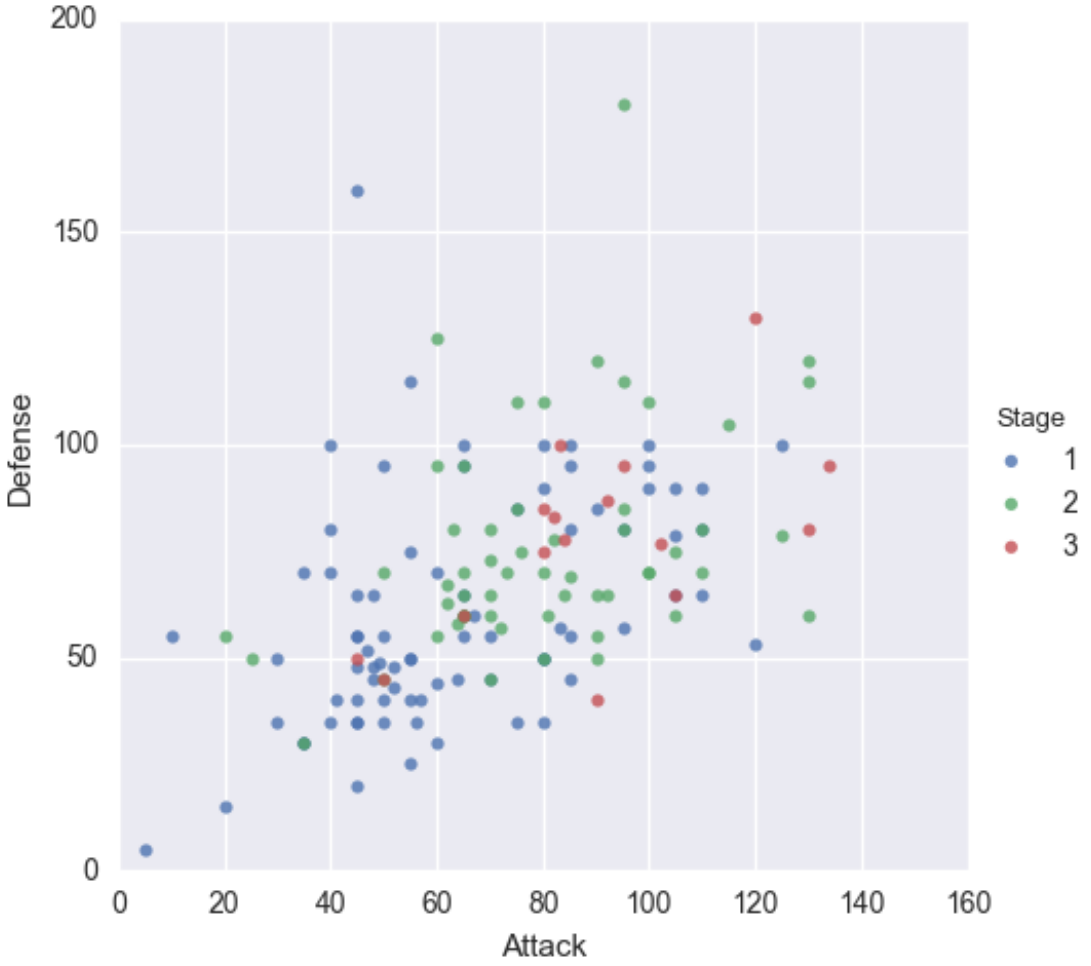
1. First, invoke your Seaborn plotting function as normal.
2. Then, invoke Matplotlib's customization functions. In this case, we'll use its ylim() and xlim() functions.

Here's our new scatter plot with sensible axes limits:

Customizing with Matplotlib
Python

```
# Plot using Seaborn
sns.lmplot(x='Attack', y='Defense', data=df,
           fit_reg=False,
           hue='Stage')
# Tweak using Matplotlib
plt.ylim(0, None)
plt.xlim(0, None)

```




For more information on Matplotlib's customization functions, check out its documentation.

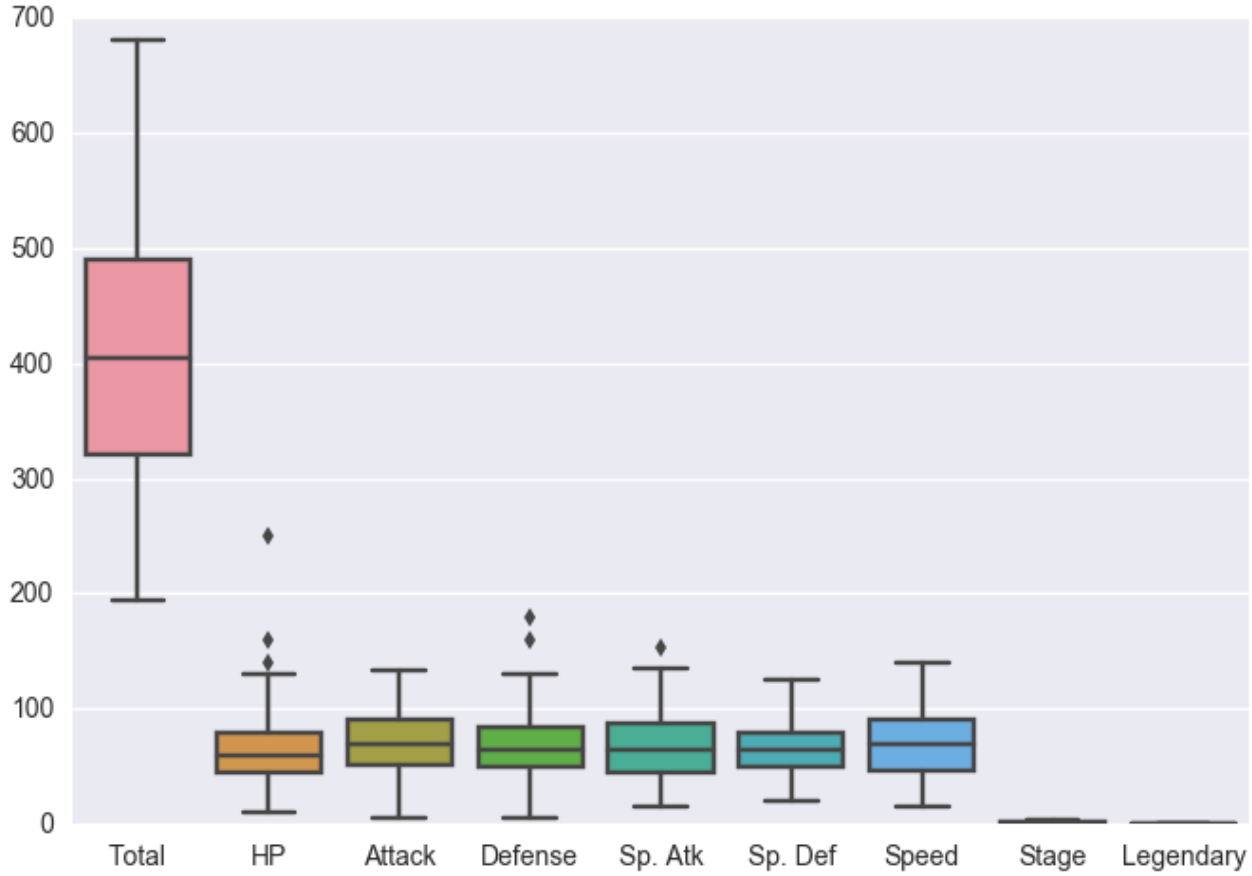
## Step 5: The role of Pandas.

Even though this is a Seaborn tutorial, Pandas actually plays a very important role. You see, Seaborn's plotting functions benefit from a base DataFrame that's reasonably formatted.

For example, let's say we wanted to make a **box plot** for our Pokémon's combat stats:

Default boxplot
Python

```
# Boxplot
sns.boxplot(data=df)
```




Well, that's a reasonable start, but there are some columns we'd probably like to remove:

- We can remove the Total since we have individual stats.
- We can remove the Stage and Legendary columns because they aren't combat stats.

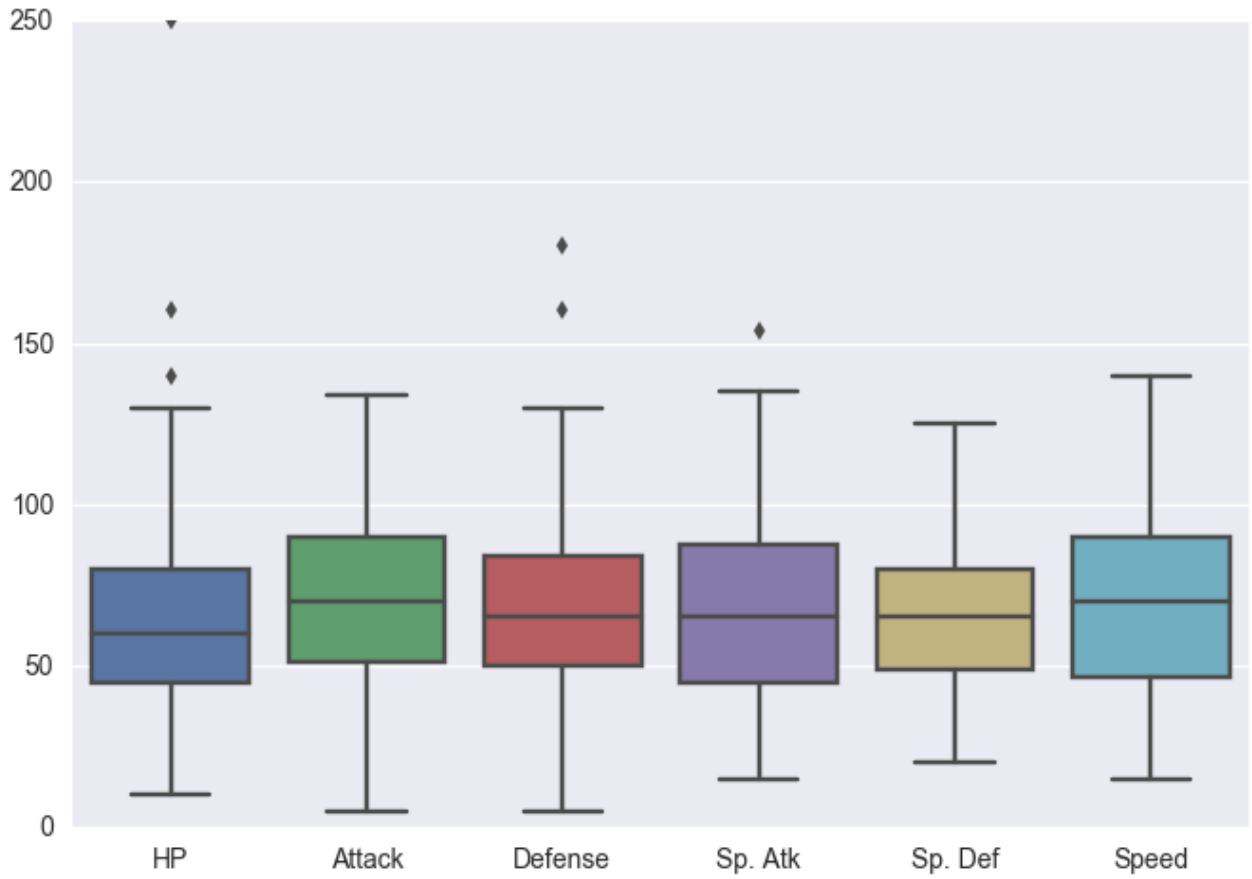
In turns out that this isn't easy to do within Seaborn alone. Instead, it's much simpler to **pre-format** your DataFrame.

Let's create a new DataFrame called stats_df that only keeps the stats columns:

Pre-format DataFrame
Python

```
# Pre-format DataFrame
stats_df = df.drop(['Total', 'Stage', 'Legendary'], axis=1)
# New boxplot using stats_df
sns.boxplot(data=stats_df)

```




It's outside the scope of this tutorial to dive into Pandas, but here's a handy cheat sheet.

## Step 6: Seaborn themes.

Another advantage of Seaborn is that it comes with decent style themes right out of the box. The default theme is called *'darkgrid'*.

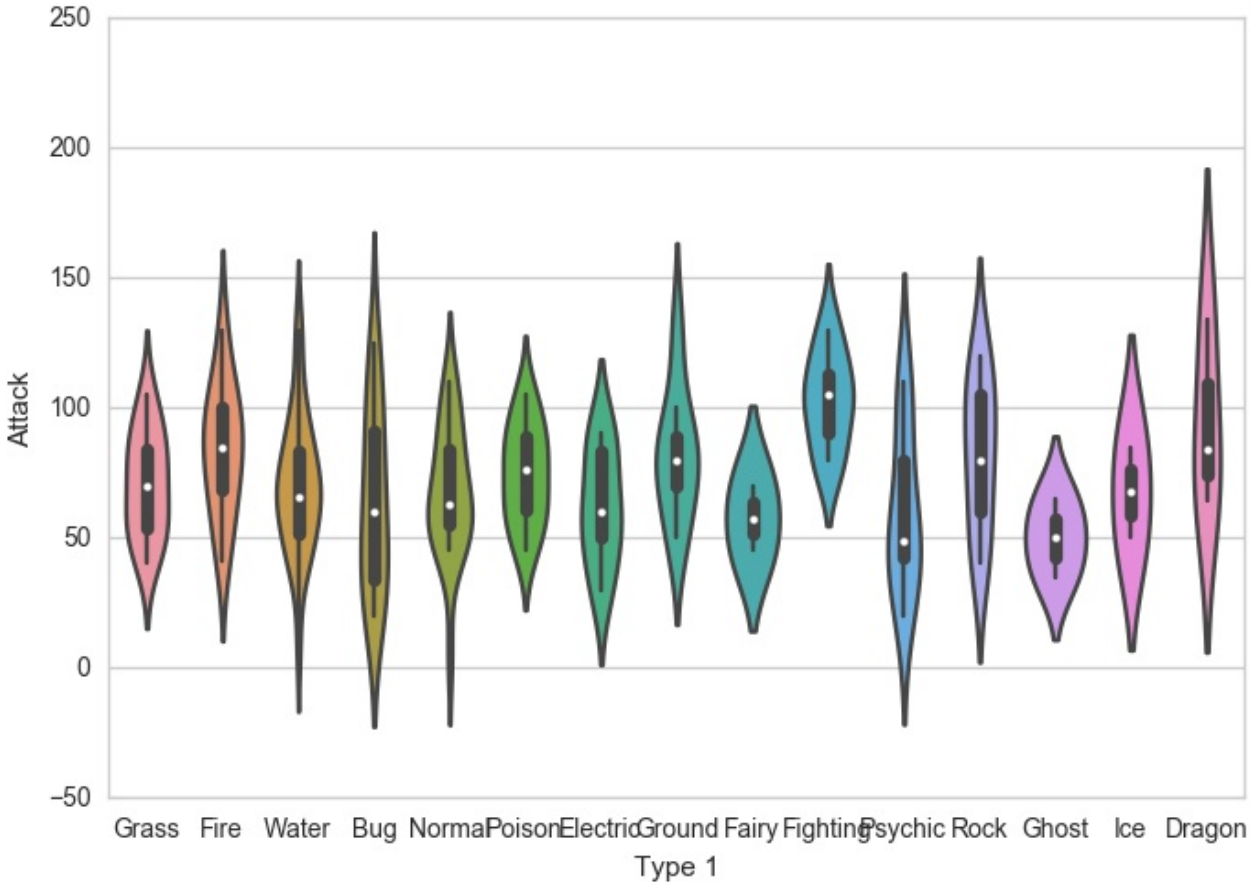
Next, we'll change the theme to *'whitegrid'* while making a **violin plot**.

- Violin plots are useful alternatives to box plots.
- They show the distribution (through the thickness of the violin) instead of only the summary statistics.

For example, we can visualize the distribution of Attack by Pokémon's primary type:

Set theme, then plot violin plot
Python

```
1  # Set theme
2  sns.set_style('whitegrid')
3  # Violin plot
4  sns.violinplot(x='Type 1', y='Attack', data=df)
5
```




As you can see, Dragon types tend to have higher Attack stats than Ghost types, but they also have greater variance.

Now, Pokémon fans might find something quite jarring about that plot: *The colors are nonsensical.* Why is the Grass type colored pink or the Water type colored orange? We must fix this!

## Step 7: Color palettes.

Fortunately, Seaborn allows us to set custom color palettes. We can simply create an ordered **Python list** of color hex values.

Let's use Bulbapedia to help us create a new color palette:

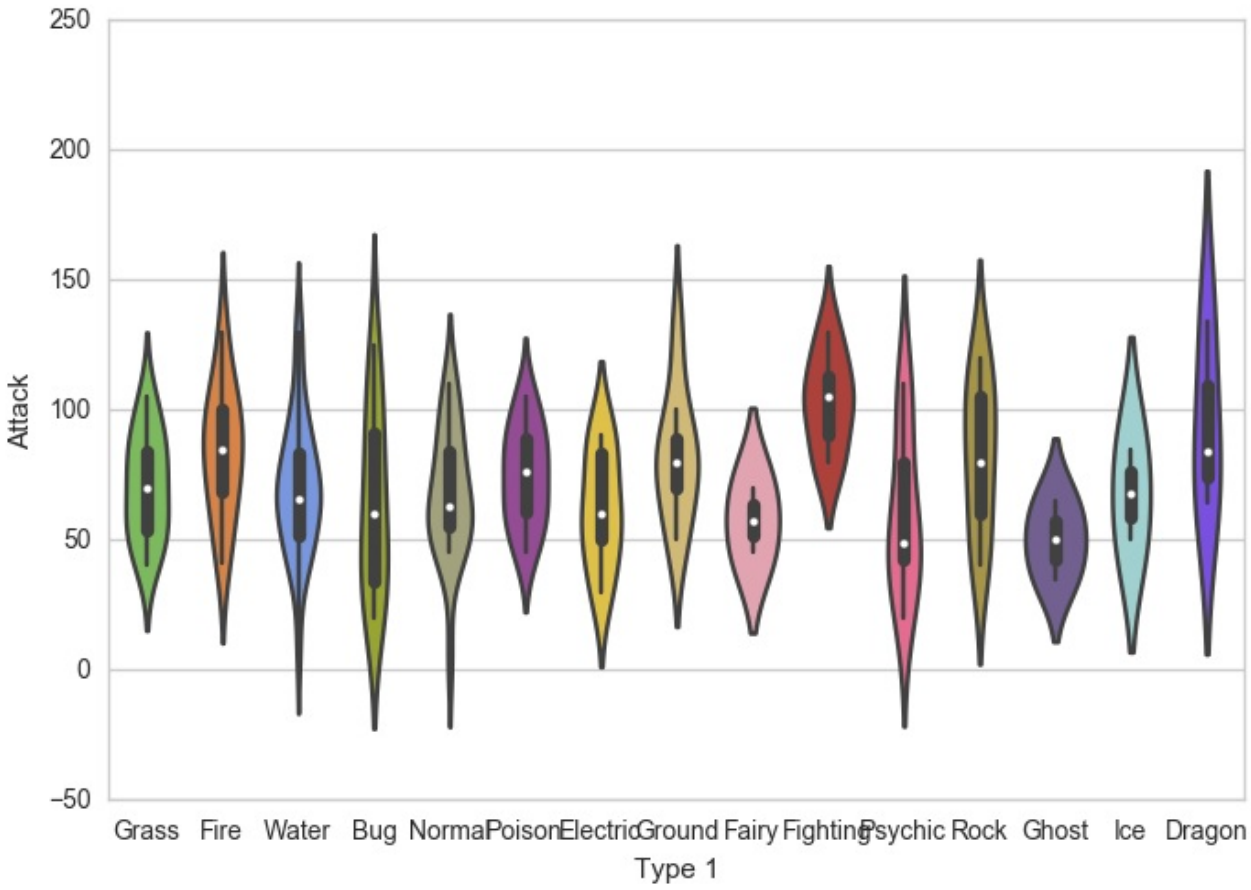
Pokemon color palette
Python

```
pkmn_type_colors = ['#78C850',  # Grass
                    '#F08030',  # Fire
                    '#6890F0',  # Water
                    '#A8B820',  # Bug
                    '#A8A878',  # Normal
                    '#A040A0',  # Poison
                    '#F8D030',  # Electric
                    '#E0C068',  # Ground
                    '#EE99AC',  # Fairy
                    '#C03028',  # Fighting
                    '#F85888',  # Psychic
                    '#B8A038',  # Rock
                    '#705898',  # Ghost
                    '#98D8D8',  # Ice
                    '#7038F8',  # Dragon
                   ]
```

Wonderful. Now we can simply use the palette= argument to recolor our chart.

Custom color palette
Python

```
# Violin plot with Pokemon color palette
sns.violinplot(x='Type 1', y='Attack', data=df,
               palette=pkmn_type_colors) # Set color palette
```



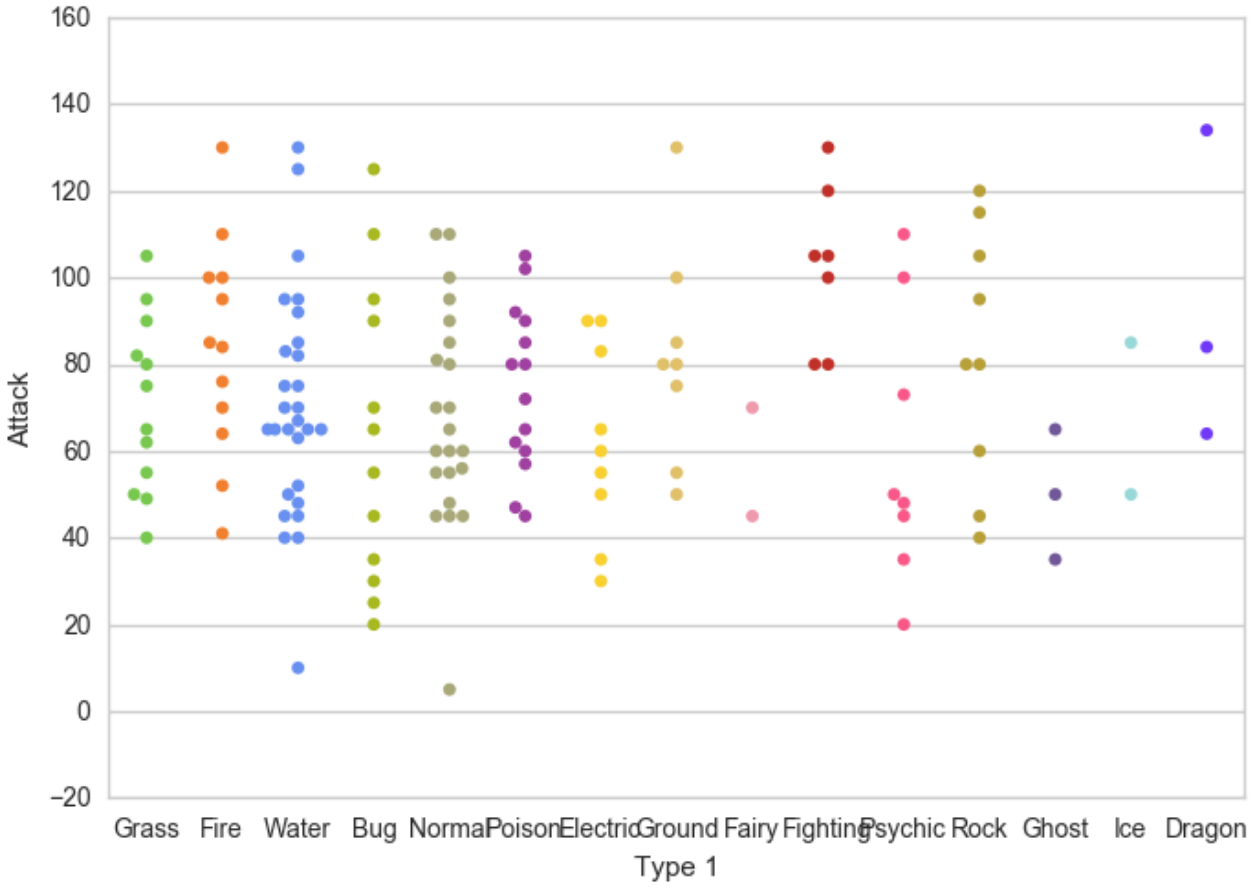

Much better!

Violin plots are great for visualizing distributions. However, since we only have 151 Pokémon in our dataset, we may want to simply display each point.

That's where the **swarm plot** comes in. This visualization will show each point, while "stacking" those with similar values:

Swarm plot
Python

```
# Swarm plot with Pokemon color palette
sns.swarmplot(x='Type 1', y='Attack', data=df,
              palette=pkmn_type_colors)
```




That's handy, but can't we combine our swarm plot and the violin plot? After all, they display similar information, right?

## Step 8: Overlaying plots.

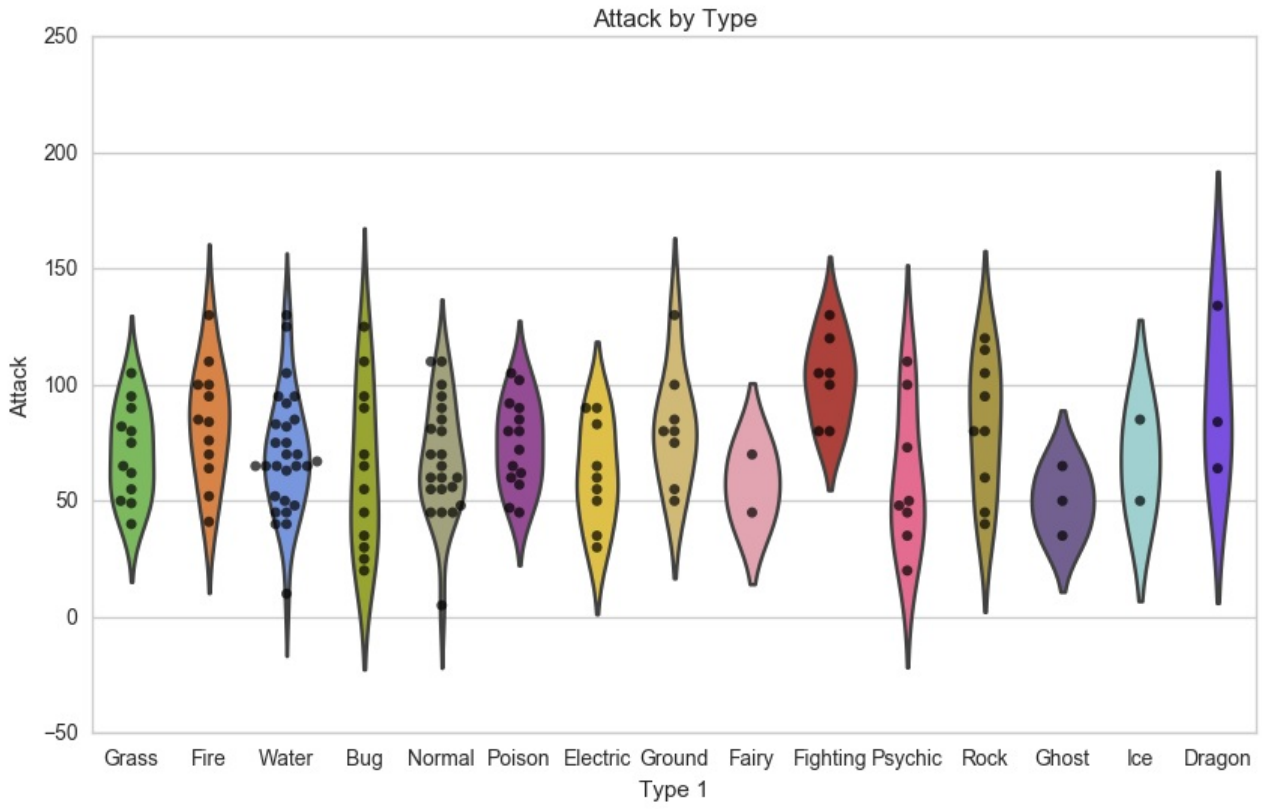
The answer is yes.

It's pretty straightforward to overlay plots using Seaborn, and it works the same way as with Matplotlib. Here's what we'll do:

1. First, we'll make our figure larger using Matplotlib.
2. Then, we'll plot the violin plot. However, we'll set inner=None to remove the bars inside the violins.
3. Next, we'll plot the swarm plot. This time, we'll make the points black so they pop out more.
4. Finally, we'll set a title using Matplotlib.

Overlaying swarm and violin plots
Python

```
# Set figure size with matplotlib
plt.figure(figsize=(10,6))
# Create plot
sns.violinplot(x='Type 1',
               y='Attack',
               data=df,
               inner=None, # Remove the bars inside the violins
               palette=pkmn_type_colors)
sns.swarmplot(x='Type 1',
              y='Attack',
              data=df,
              color='k', # Make points black
              alpha=0.7) # and slightly transparent
# Set title with matplotlib
plt.title('Attack by Type')
```




Awesome, now we have a pretty chart that tells us how Attack values are distributed across different Pokémon types. But what it we want to see all of the other stats as well?

## Step 9: Putting it all together.

Well, we could certainly repeat that chart for each stat. But we can also combine the information into one chart... we just have to do some **data wrangling** with Pandas beforehand.

First, here's a reminder of our data format:

First 5 rows of stats_df
Python

```
1  stats_df.head()
```

| | Name | Type 1 | Type 2 | HP | Attack | Defense | Sp. Atk | Sp. Def | Speed |
|---|---|---|---|---|---|---|---|---|---|
| # | | | | | | | | | |
| 1 | Bulbasaur | Grass | Poison | 45 | 49 | 49 | 65 | 65 | 45 |
| 2 | Ivysaur | Grass | Poison | 60 | 62 | 63 | 80 | 80 | 60 |
| 3 | Venusaur | Grass | Poison | 80 | 82 | 83 | 100 | 100 | 80 |
| 4 | Charmander | Fire | NaN | 39 | 52 | 43 | 60 | 50 | 65 |
| 5 | Charmeleon | Fire | NaN | 58 | 64 | 58 | 80 | 65 | 80 |

As you can see, all of our stats are in separate columns. Instead, we want to "melt" them into one column.

To do so, we'll use Pandas's melt() function. It takes 3 arguments:

- First, the DataFrame to melt.
- Second, ID variables to keep (Pandas will melt all of the other ones).
- Finally, a name for the new, melted variable.

Here's the output:

Melt DataFrame
Python

```
1  # Melt DataFrame
2  melted_df = pd.melt(stats_df,
3                      id_vars=["Name", "Type 1", "Type 2"], # Variables to keep
4                      var_name="Stat") # Name of melted variable
5  melted_df.head()
```

| | Name | Type 1 | Type 2 | Stat | value |
|---|---|---|---|---|---|
| 0 | Bulbasaur | Grass | Poison | HP | 45 |
| 1 | Ivysaur | Grass | Poison | HP | 60 |
| 2 | Venusaur | Grass | Poison | HP | 80 |
| 3 | Charmander | Fire | NaN | HP | 39 |
| 4 | Charmeleon | Fire | NaN | HP | 58 |

All 6 of the stat columns have been "melted" into one, and the new Stat column indicates the original stat (HP, Attack, Defense, Sp. Attack, Sp. Defense, or Speed). For example, it's hard to see here, but Bulbasaur now has 6 rows of data.

In fact, if you print the shape of these two DataFrames...

Shape comparison
Python

```
print( stats_df.shape )
print( melted_df.shape )
# (151, 9)
# (906, 5)
```

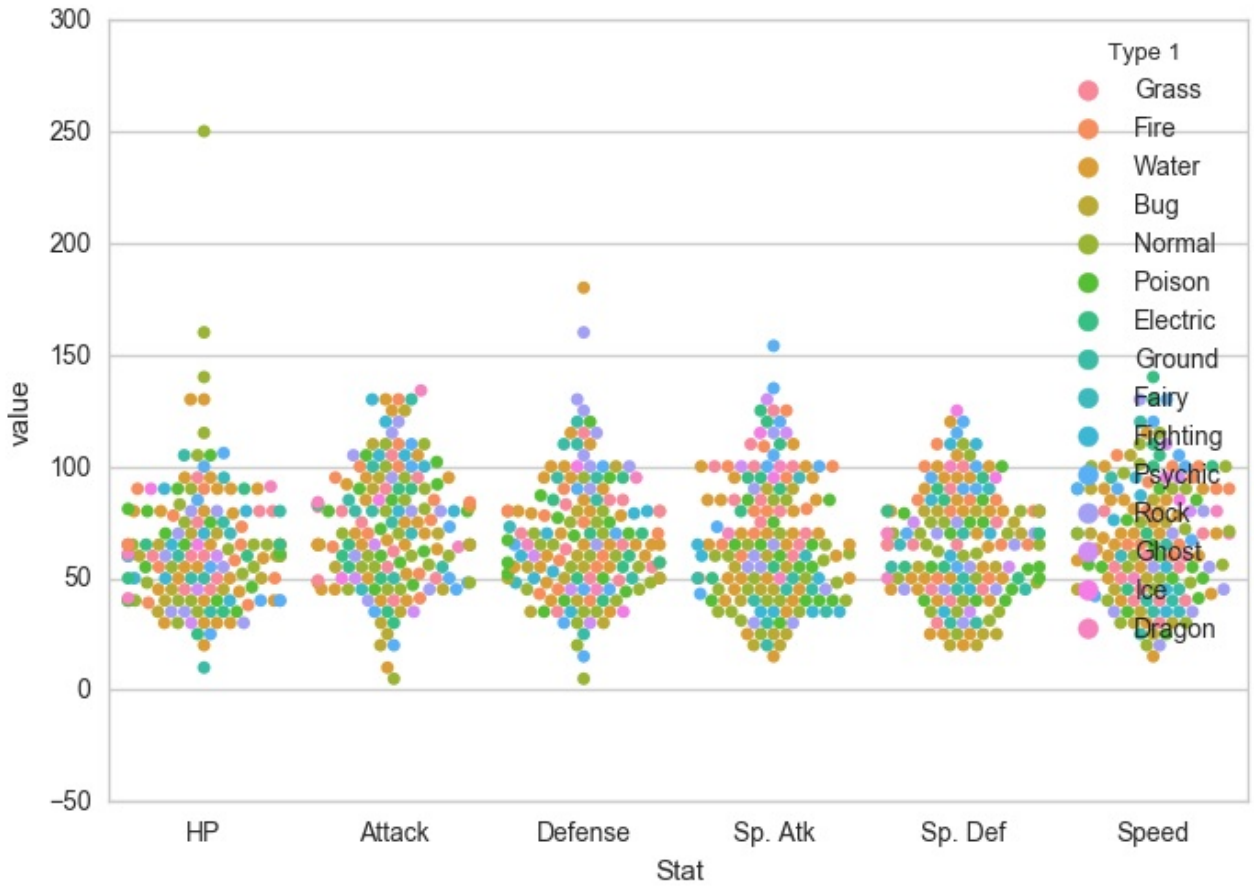
...you'll find that melted_df has 6 times the number of rows as stats_df.

Now we can make a swarm plot with melted_df.

- But this time, we're going to set x='Stat' and y='value' so our swarms are separated by stat.
- Then, we'll set hue='Type 1' to color our points by the Pokémon type.

Swarmplot with melted_df
Python

```
# Swarmplot with melted_df
sns.swarmplot(x='Stat', y='value', data=melted_df,
              hue='Type 1')
```

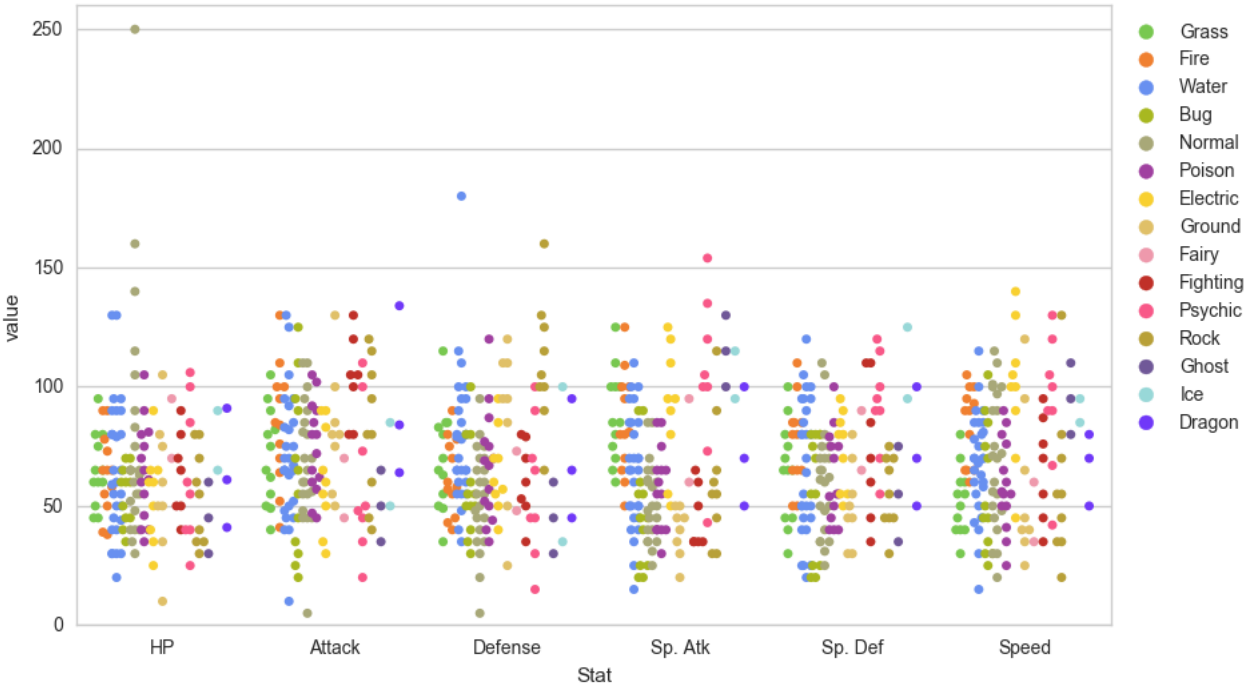



Finally, let's make a few final tweaks for a more readable chart:

1. Enlarge the plot.
2. Separate points by hue using the argument split=True .
3. Use our custom Pokemon color palette.
4. Adjust the y-axis limits to end at 0.
5. Place the legend to the right.

Customizations
Python

```
# 1. Enlarge the plot
plt.figure(figsize=(10,6))
sns.swarmplot(x='Stat',
              y='value',
              data=melted_df,
              hue='Type 1',
              split=True, # 2. Separate points by hue
              palette=pkmn_type_colors) # 3. Use Pokemon palette
# 4. Adjust the y-axis
plt.ylim(0, 260)
# 5. Place legend to the right
plt.legend(bbox_to_anchor=(1, 1), loc=2)
```




There we go!

## Step 10: Pokédex (mini-gallery).

We're going to conclude this tutorial with a few quick-fire data visualizations, just to give you a sense of what's possible with Seaborn.

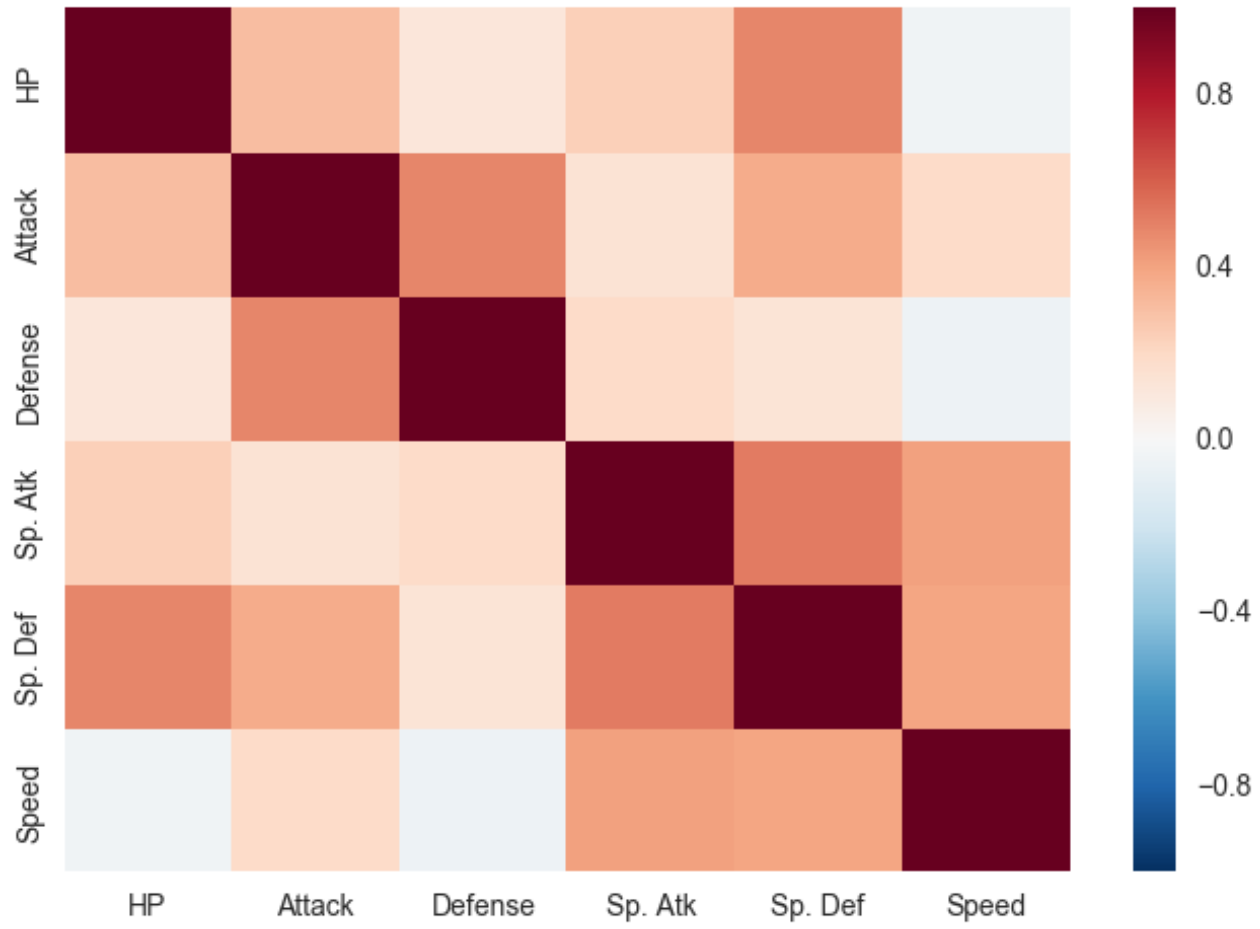
### 10.1 - Heatmap

Heatmaps help you visualize matrix-like data.

Heatmap
Python

```
# Calculate correlations
corr = stats_df.corr()
# Heatmap
sns.heatmap(corr)

```

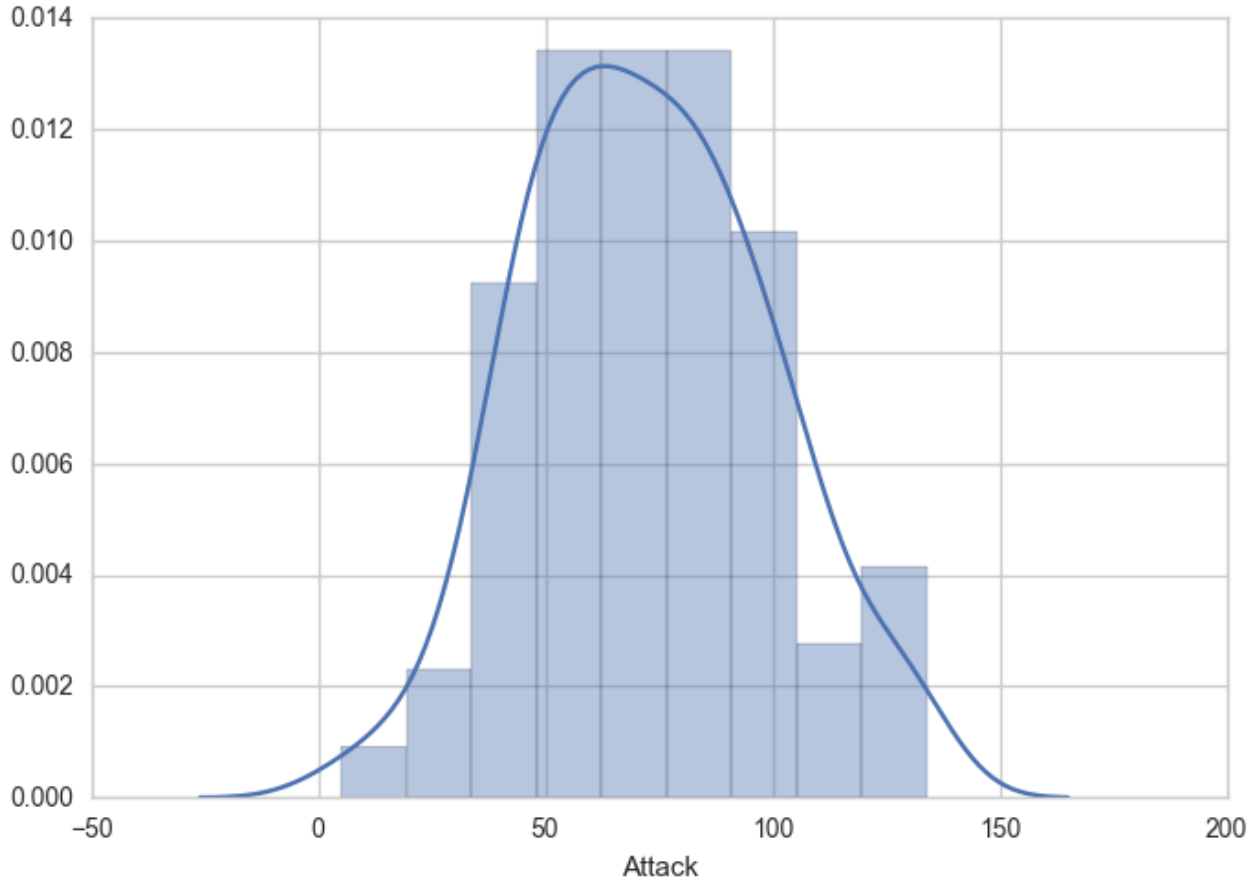



## 10.2 - Histogram

Histograms allow you to plot the distributions of numeric variables.

Histogram
Python

```
# Distribution Plot (a.k.a. Histogram)
sns.distplot(df.Attack)
```

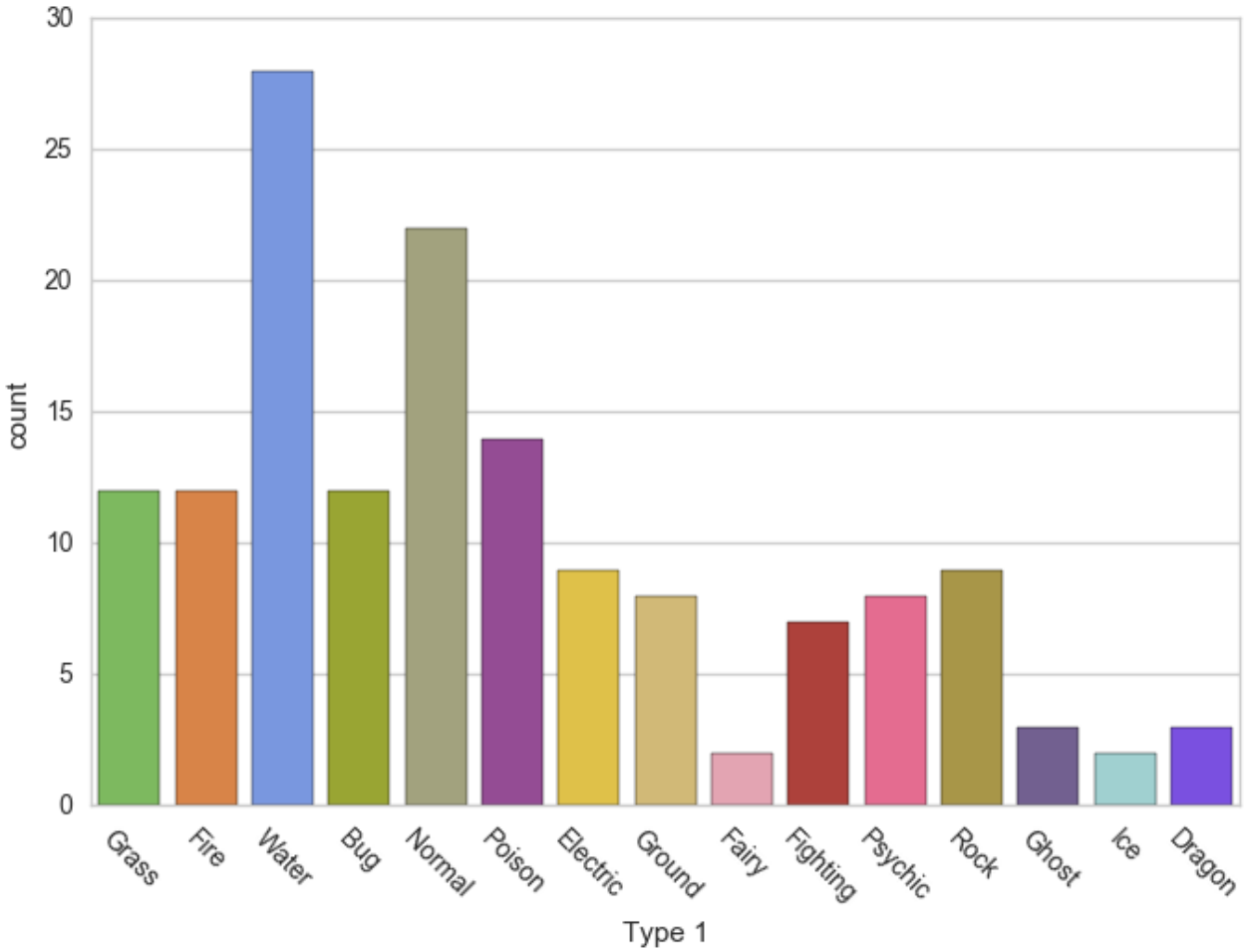



## 10.3 - Bar Plot

Bar plots help you visualize the distributions of categorical variables.

Bar Plot
Python

```
1  # Count Plot (a.k.a. Bar Plot)
2  sns.countplot(x='Type 1', data=df, palette=pkmn_type_colors)
3  # Rotate x-labels
4  plt.xticks(rotation=-45)
5
```



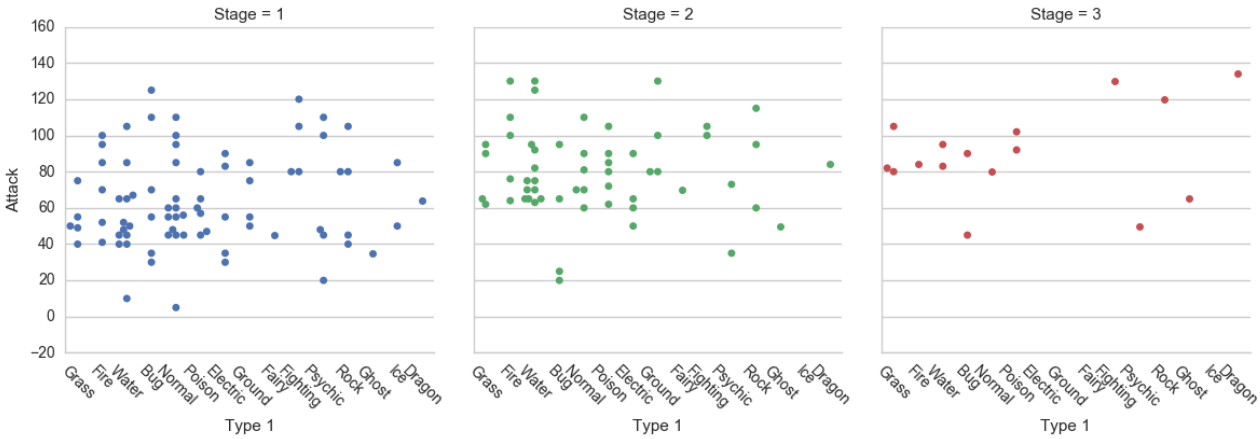

## 10.4 - Factor Plot

Factor plots make it easy to separate plots by categorical classes.

Factor Plot
Python

```
# Factor Plot
g = sns.factorplot(x='Type 1',
                   y='Attack',
                   data=df,
                   hue='Stage',  # Color by stage
                   col='Stage',  # Separate by stage
                   kind='swarm') # Swarmplot
# Rotate x-axis labels
g.set_xticklabels(rotation=-45)
# Doesn't work because only rotates last plot
# plt.xticks(rotation=-45)


```



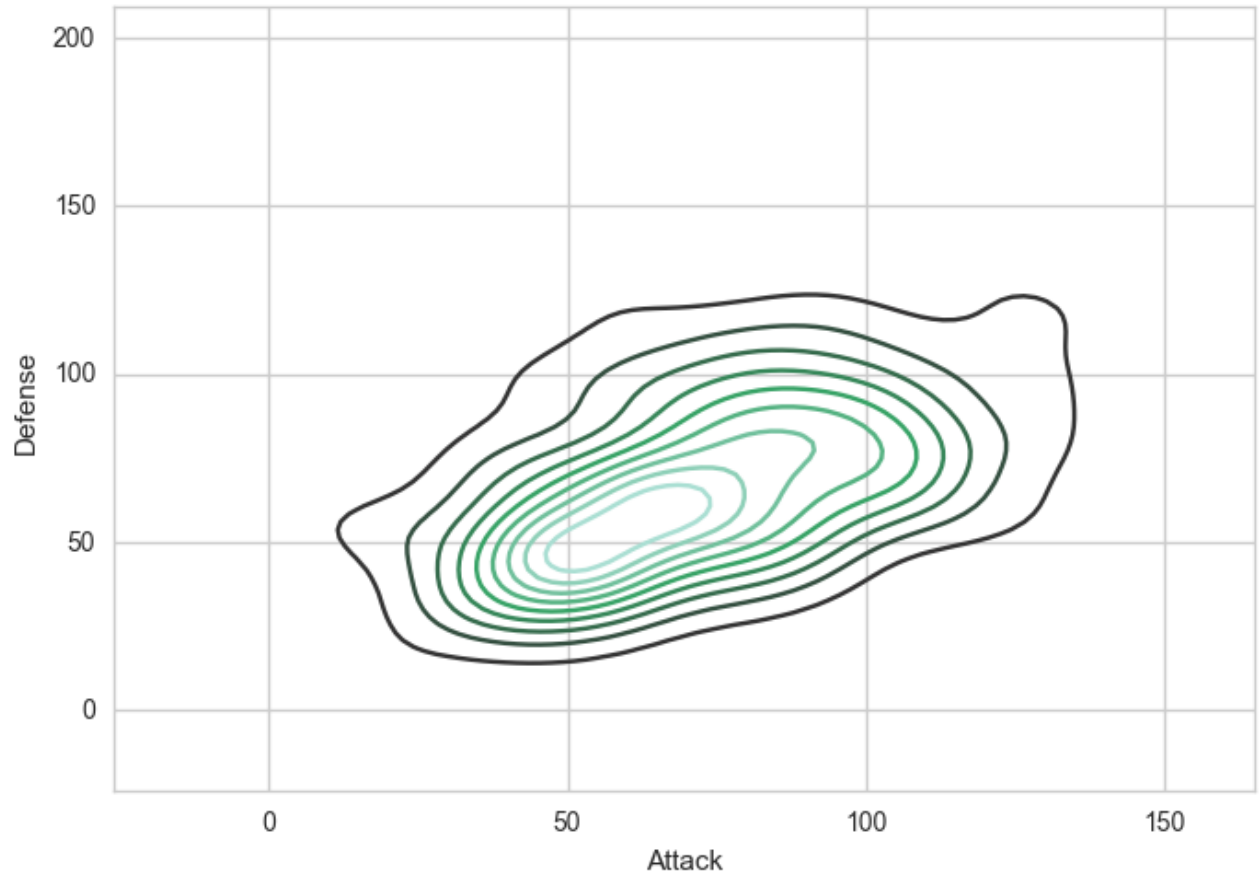

## 10.5 - Density Plot

Density plots display the distribution between two variables.

> **Tip:** Consider overlaying this with a scatter plot.

Density Plot
Python

```
1  # Density Plot
2  sns.kdeplot(df.Attack, df.Defense)
```

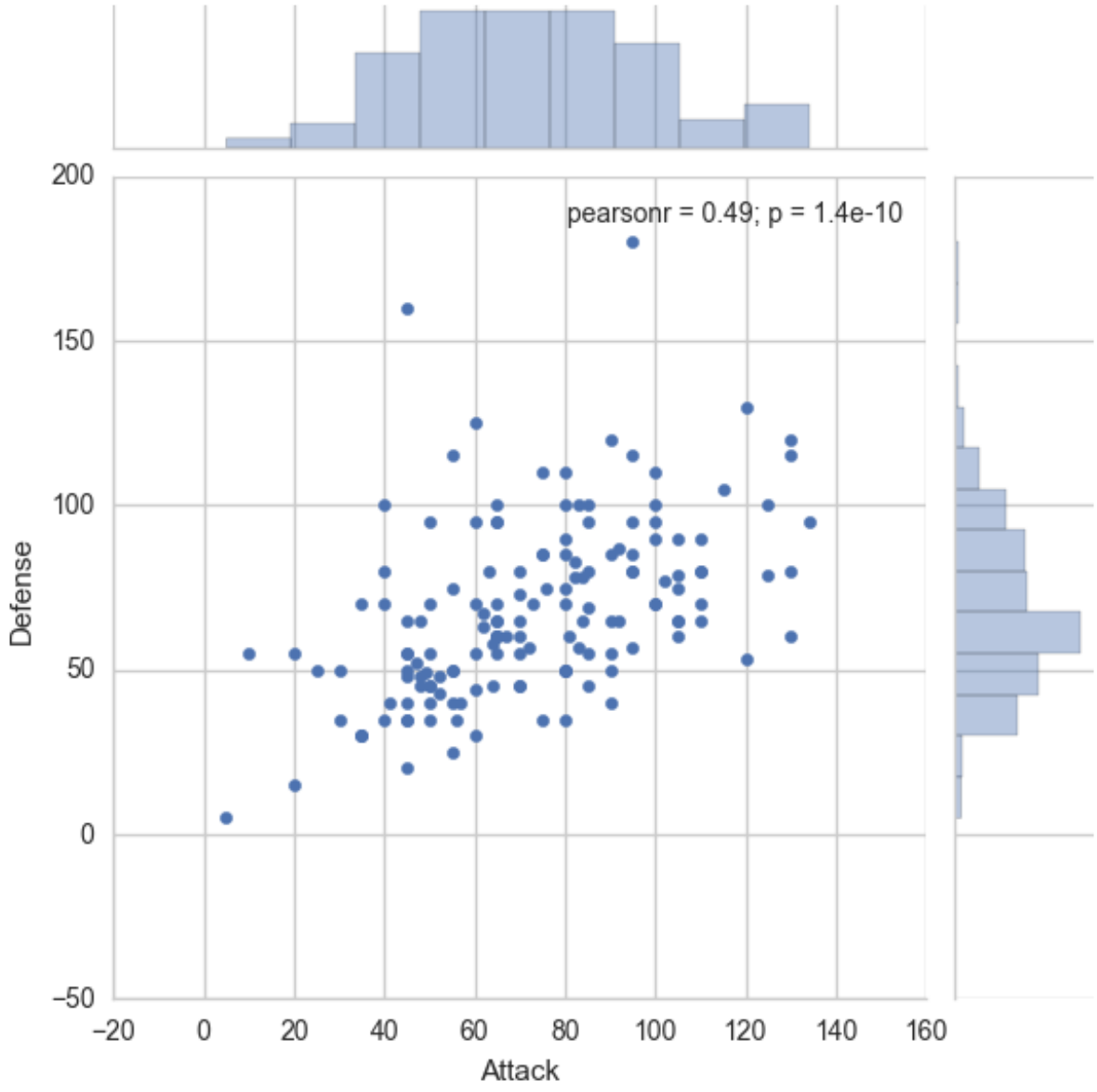



## 10.6 - Joint Distribution Plot

Joint distribution plots combine information from scatter plots and histograms to give you detailed information for bi-variate distributions.

Joint Distribution Plot
Python

```
1  # Joint Distribution Plot
2  sns.jointplot(x='Attack', y='Defense', data=df)
```




Congratulations... you've made it to the end of this Python Seaborn tutorial!

We've just concluded a tour of key Seaborn paradigms and showed you many examples along the way. Feel free to use this page along with the official Seaborn gallery as references for your projects going forward.