

The Machine Learning Masterclass

Companion E-Book



EliteDataScience.com

COPYRIGHT NOTICE

Copyright © 2017+ EliteDataScience.com

ALL RIGHTS RESERVED.

This book or parts thereof may not be reproduced in any form, stored in any retrieval system, or transmitted in any form by any means—electronic, mechanical, photocopy, recording, or otherwise—without prior written permission of the publisher, except as provided by United States of America copyright law.

Contents

1	The Big Picture	8
1.1	Model Complexity	9
1.1.1	Models vs. machine learning	9
1.1.2	Mapping functions	9
1.1.3	Model complexity	10
1.1.4	Signal vs. noise	10
1.1.5	Overfitting vs. underfitting	11
1.2	Noisy Sine Wave	12
1.2.1	Methodology	12
1.2.2	Synthetic dataset	12
1.2.3	Plot and interpretation	13
1.3	Mean Model	14
1.3.1	Bare bones simplicity	14
1.3.2	Analysis	14
1.4	Linear Regression	16
1.4.1	Adding complexity	16
1.4.2	Analysis	17
1.5	Polynomial Regression	18
1.5.1	Even more complexity	18
1.5.2	Analysis	18
1.6	Decision Trees	20
1.6.1	Complexity overload	20
1.6.2	Analysis	20
1.6.3	Tree ensembles	21

2	Exploratory Analysis	22
2.1	Basic Information	23
2.1.1	Shape	23
2.1.2	Datatypes	23
2.1.3	Example observations	23
2.2	Distributions of numeric features	25
2.2.1	Histogram grid	25
2.2.2	Summary statistics	26
2.3	Distributions of categorical features	27
2.3.1	Bar plots	27
2.3.2	Sparse classes	27
2.4	Segmentations	29
2.4.1	Box plots	29
2.4.2	Groupby	29
2.5	Correlations	31
2.5.1	Intuition	31
2.5.2	DataFrame correlations	31
2.5.3	Heatmaps	31
3	Data Cleaning	34
3.1	Unwanted Observations	35
3.1.1	Duplicate	35
3.1.2	Irrelevant	35
3.2	Structural Errors	36
3.2.1	Wannabe indicator variables	36
3.2.2	Typos and capitalization	36
3.2.3	Mislabeled classes	37
3.3	Unwanted Outliers	38
3.3.1	Innocent until proven guilty	38
3.3.2	Violin plots	38
3.3.3	Manual check	39
3.4	Missing Data	40
3.4.1	"Common sense" is not sensible here	40
3.4.2	Missing categorical data	41
3.4.3	Missing numeric data	41
4	Feature Engineering	42
4.1	Domain Knowledge	43
4.1.1	Boolean masks	43
4.1.2	Link with Exploratory Analysis	43

4.2	Interaction Features	45
4.2.1	Examples	45
4.3	Sparse Classes	46
4.3.1	Similar classes	46
4.3.2	Other classes	46
4.4	Dummy Variables	48
4.4.1	Example: Project 2	48
4.4.2	Get dummies	48
4.5	Remove Unused	49
4.5.1	Unused features	49
4.5.2	Redundant features	49
4.5.3	Analytical base table	49
4.6	Rolling Up	50
5	Model Training	51
5.1	Split Dataset	52
5.1.1	Training and test sets	52
5.1.2	Cross-validation (intuition)	52
5.1.3	10-fold cross-validation	53
5.2	Model Pipelines	54
5.2.1	Preprocessing	54
5.2.2	Standardization	54
5.2.3	Preprocessing parameters	54
5.2.4	Pipelines and cross-validation	55
5.2.5	Pipeline dictionaries	55
5.3	Declare Hyperparameters	57
5.3.1	Model parameters vs. hyperparameters	57
5.3.2	Hyperparameter grids (regression)	57
5.3.3	Hyperparameter grids (classification)	58
5.3.4	Hyperparameter dictionary	59
5.4	Fit and tune models	60
5.4.1	Grid search	60
5.4.2	Looping through dictionaries	60
5.5	Select Winner	62
5.5.1	Cross-validated score	62
5.5.2	Performance metrics (regression)	62
5.5.3	Performance metrics (classification)	62
5.5.4	Test performance	63
5.5.5	Saving the winning model	64

6	Project Delivery	65
6.1	Confirm Model	66
6.1.1	Import model and load ABT	66
6.1.2	Recreate training and test sets	66
6.1.3	Predict the test set again	67
6.2	Pre-Modeling Functions	68
6.2.1	Data cleaning function	68
6.2.2	Feature engineering function	69
6.3	Model Class	70
6.3.1	Python classes	70
6.3.2	Custom class	70
6.4	Model Deployment	71
6.4.1	Jupyter notebook	71
6.4.2	Executable script	71
7	Regression Algorithms	73
7.1	The "Final Boss" of Machine Learning	74
7.1.1	How to find the right amount of complexity	74
7.1.2	How overfitting occurs	74
7.2	Regularization	75
7.2.1	Flaws of linear regression	75
7.2.2	The number of features is too damn high!	75
7.2.3	Cost functions	76
7.2.4	Example: sum of squared errors	76
7.2.5	Coefficient penalties	77
7.2.6	L_1 regularization	77
7.2.7	L_2 regularization	77
7.3	Regularized Regression	78
7.3.1	Lasso	78
7.3.2	Ridge	78
7.3.3	Elastic-Net	78
7.4	Ensemble Methods	80
7.4.1	Decision trees (kinda)	80
7.4.2	Non-linear relationships	81
7.4.3	Unconstrained decision trees	81
7.4.4	Bagging vs. boosting	82
7.5	Tree Ensembles	83
7.5.1	Random forests	83
7.5.2	Boosted trees	83

8	Classification Algorithms	84
8.1	Binary Classification	85
8.1.1	Positive / negative classes	85
8.1.2	Class probabilities	85
8.2	Noisy Conditional	86
8.2.1	Methodology	86
8.2.2	Synthetic dataset	86
8.3	Logistic Regression	88
8.3.1	Linear vs. logistic regression	88
8.3.2	Predict vs. predict_proba	88
8.4	Regularized Logistic Regression	90
8.4.1	Penalty strength	90
8.4.2	Penalty type	90
8.5	Tree Ensembles	92
8.5.1	Random forests	92
8.5.2	Boosted trees	92
9	Clustering Algorithms	93
9.1	K-Means	94
9.1.1	Intuition	94
9.1.2	Euclidean distance	95
9.1.3	Number of clusters	96
9.2	Feature Sets	97
9.2.1	Creating feature sets	97
9.2.2	Finding clusters	97
A	Appendix	99
A.1	Area Under ROC Curve	99
A.1.1	Confusion matrix	99
A.1.2	TPR and FPR	100
A.1.3	Probability thresholds	100
A.1.4	ROC curve	101
A.1.5	AUROC	103
A.2	Data Wrangling	104
A.2.1	Customer-Level feature engineering	104
A.2.2	Intermediary levels	105
A.2.3	Joining together the ABT	105
A.3	Dimensionality Reduction	106
A.3.1	The Curse of Dimensionality	106
A.3.2	Method 1: Thresholding	107
A.3.3	Method 2: Principal Component Analysis	108

1. The Big Picture

Welcome to the *Companion E-Book* for **The Machine Learning Masterclass** by [Elite Data Science](#).

This e-book is meant to be a guide that you can use to review all of the key concepts you've learned throughout the course, and it assumes you've already completed at least some of the course.

Therefore, we're going to jump right into what we consider to be the "**heart of machine learning**" (and that's no exaggeration).

You see, at its core, machine learning is about building models that can represent the world in some important way. All of the fancy algorithms out there are really just trying to do 1 thing: learn these representations (a.k.a. "models") from data.

As a result, there's a concept called **model complexity** that is incredibly important. It really determines how "good" a model is. (This is closely related to the **bias-variance tradeoff**, which you may have also heard of. However, we've found the lens of model complexity to be more intuitive.)

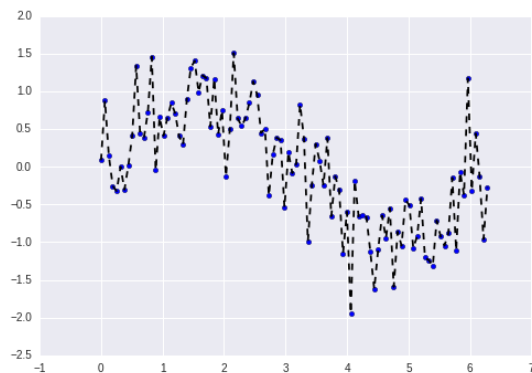


Figure 1.1: Is this a good model? Keep reading to find out!

1.1 Model Complexity

Model complexity is one of the most important concepts for practical machine learning.

But before we can discuss model complexity, let's take a step back and talk about what a "model" actually is.

1.1.1 Models vs. machine learning

Often, these two terms get mixed up, but we want to be explicit about the difference.

In fact, let's first put machine learning aside and talk about models at an abstract level.

Models are simplified representations of the real world that still accurately capture a desired relationship.

In Project 1, we gave the example of a child who learns that stove-burner because he stuck his hand over a candle.

- That connection, "**red and bright means painful**," is a model.
- It's a **simplified representation**; the child didn't need to know anything about fire, combustion, thermodynamics, human anatomy, or nerve endings.
- For a model to be useful, it must still accurately capture a useful relationship.



Figure 1.2: Red and bright means painful.

There are different ways to build models, and **machine learning is only one of them**. Machine learning builds models by learning patterns from data.

- Therefore, you can think of a model as an "output" of the machine learning process.
- In other words, models are the "lessons" learned from the data.

1.1.2 Mapping functions

Another way to understand models is to think of them as **mapping functions between input features and target variables**:

$$\hat{y} = f(x)$$

We don't use a lot of math in this course, but it's helpful to introduce some basic notation so that we can have a shared language for illustrating these concepts.

- x is the input feature.
- \hat{y} is the predicted target variable.
- $f()$ is the mapping function between x and \hat{y}

Therefore, the purpose of machine learning is to **estimate that mapping function** based on the dataset that you have.

1.1.3 Model complexity

So which type of algorithm should you use to build the model? Which algorithm "settings" should you tune? How many features should you include?

These will be central questions that we'll explore in this course, and SPOILER WARNING: the answers are all tied to model complexity.

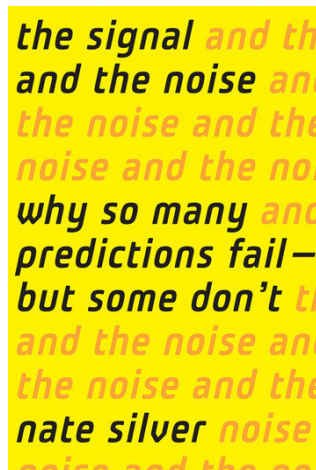
Model complexity is a model's inherent flexibility to represent complex relationships between input features and target variables.

This definition will become crystal clear in just a few moments when we get to the examples.

1.1.4 Signal vs. noise

One practical challenge in real-world machine learning problems is dealing with **noise**.

- Think of noise as "randomness" in your data that you can't control.
- For example, in the Iris dataset, there is natural fluctuation in petal sizes, even within the same species.



the signal and the noise and the noise and the noise and the noise why so many predictions fail – but some don't the signal and the noise and the noise and the noise nate silver noise

Figure 1.3: *The Signal and the Noise*, by Nate Silver

Therefore, one way to judge models is by their ability to **separate the signal from that noise**.

- Think of the signal as the "**true underlying relationship**" between input features and target variables.
- Different machine learning algorithms are simply different ways to estimate that signal.

1.1.5 Overfitting vs. underfitting

So how does this relate to model complexity?

Well, in applied machine learning, model complexity manifests as **overfitting** or **underfitting**.

- *Overfitting* occurs when the model learns the noise instead of the signal. This tends to happen when the model is **too complex**.
- *Underfitting* occurs when the model is not flexible enough to learn the signal. This tends to happen when the model is **too simple**.

In this course, we explore practical techniques for finding the ideal balance and choosing the right level of complexity.

But for now, let's jump into toy example dig into this concept a bit more!

1.2 Noisy Sine Wave

We're going to switch over to a toy example so we can really develop a practical intuition behind model complexity.

1.2.1 Methodology

Remember, a model is only useful if it can accurately approximate the "true state of the world" (i.e. the "**true underlying relationship**" between input features and target variables).

Therefore, to study this concept, we're going to create a synthetic dataset where we already know "true underlying relationship."

1. First, we're going to use a single input feature, x .
2. Then, we're going to generate values for the target variable, y , based on a **predetermined mapping function**.
3. Next, we're going to add **randomly generated noise** to that dataset.
4. Once we've done that, we can try different algorithms on our synthetic dataset.
5. We already know the "true underlying relationship"... it's the predetermined mapping function.
6. Finally, we can compare how well models of different complexities can separate the **signal** from the randomly generated noise.

Let's dive right in.

1.2.2 Synthetic dataset

First, for our predetermined mapping function, we'll use the sine wave.

$$y = \sin(x)$$

However, we're going to add random noise to it, turning it into a **noisy sine wave**:

$$y = \sin(x) + \epsilon$$

```
# input feature
x = np.linspace(0, 2*np.pi, 100)

# noise
np.random.seed(321)
noise = np.random.normal(0, .5, 100)

# target variable
y = np.sin(x) + noise
```

1.2.3 Plot and interpretation

Figure 1.4 displays that synthetic dataset.

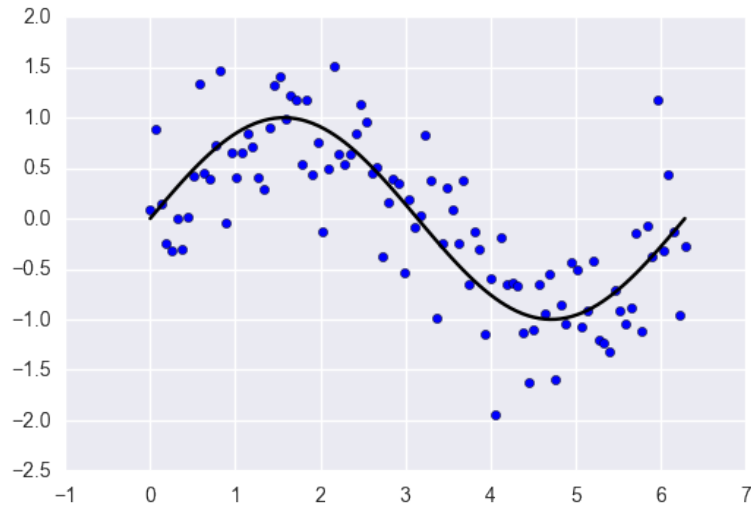


Figure 1.4: The synthetic dataset

In the plot, the *blue dots* represent the synthetic dataset, with x on the x-axis and $y = \sin(x) + \epsilon$ on the y-axis.

The **smooth black line** represents the sine wave $f() = \sin(x)$ without any noise. This is the "true underlying relationship" between x and y before adding random noise.

Next, we'll see how well models of varying complexity are able to learn this signal.

1.3 Mean Model

Now that we have a synthetic dataset, let's build some models and see how they do.

To be 100% clear, our goal is to build a model that can **predict the target variable y from a given value for the input feature x** .

1.3.1 Bare bones simplicity

Let's start with a very simple model.

In fact, this is probably the simplest model you can actually build that still "learns" from the data, and it's called the **mean model**.

$$\hat{y} = \bar{y}$$

This literally means that we will always predict the average of all y 's in the dataset, no matter what the value of x is.

- \hat{y} is the **predicted** value of y .
- \bar{y} is the average of all y 's in the dataset.

As you can guess, we call this the mean model because our prediction will always just be the mean of the y 's in the dataset, regardless of x .

```
# Build model
pred = np.mean(df.y)
```

1.3.2 Analysis

To tell if this is a good model, we don't even need to do any calculations... We can just look at the plot and think about it intuitively.

```
# Scatterplot of x and y
plt.scatter(df.x, df.y)

# Overlay horizontal line for the prediction
plt.plot(df.x, [pred]*len(df.x), 'k--')
```



Figure 1.5: These observations will always be predicted poorly using a mean model.

Take a look at Figure 1.5 See those observations in the big red circles?

- Predictions for those values of x will always be poor, no matter how much data you collect!
- This type of model **does not have the flexibility to learn the curves and slopes in the dataset.**

In other words, our mean model is **underfit** to the data. It's not complex enough.

1.4 Linear Regression

Next, let's introduce **linear regression**.

You may have seen these before, but if you haven't, they're really quite simple.

1.4.1 Adding complexity

Think of linear regression as fitting a straight line to the data, but there's an important difference that makes them **more complex** than the mean model.

- The mean model can only fit **horizontal** lines.
- A linear regression can fit lines with **slopes**.

Here's the formula:

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

- \hat{y} is the predicted value of y for a given x
- β_0 is called the **intercept**, and it's where "the line crosses zero" (more on this later).
- β_1 is the **coefficient** of the input feature x , and it's the slope of the line.

When you **fit** (a.k.a. **train**) a linear regression model, you're really just estimating the best values for β_0 and β_1 based on your dataset.

All it takes is 5 lines of code:

```
# LinearRegression from Scikit-Learn
from sklearn.linear_model import LinearRegression

# Initialize instance of linear regression
lm = LinearRegression()

# Separate our input features and target variable
features = df.drop('y', axis=1)
target = df.y

# Fit model to the data
lm.fit(features, target)
```

After fitting the model, we can access easily the intercept and coefficient.

```
print( lm.intercept_ )
# 0.823867512027

print( lm.coef_ )
# [-0.26773758]
```

1.4.2 Analysis

Again, let's plot this model.



Figure 1.6: Linear regression model

So how did this model do?

We'll it's pretty obvious that this linear model fits the synthetic dataset better than the mean model, even just by eyeballing it.

- By allowing the line to have a slope, you've permitted more flexibility than the horizontal line of the mean model.
- As a result, the model was at least able to represent the major "trend" in the dataset.

However, we can also see a limitation of linear regression: the **assumption of linearity**.

- Linear regression assumes that the target variable has linear relationships with the input features.
- Therefore, "**curvy**" **relationships** like our sine wave cannot be easily represented using a simple linear regression.

In other words, our model is still **underfit**, and not complex/flexible enough!

1.5 Polynomial Regression

Let's see what we can do to represent the "curvy" relationship. We'll take a look at **polynomial linear regression**.

1.5.1 Even more complexity

Polynomial linear regression fits curvy lines to the data. It does so by adding polynomial terms for x , which are simply x raised to some power.

For example, here's how to create a "**second-order**" polynomial model:

- First, create x^2 as another input feature into your model.
- Estimate another coefficient, β_2 , which is for x^2 .

Your formula now becomes:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2$$

This x^2 term now allows the model to represent a curve. If you want to add even more flexibility, you can add more and more polynomial terms.

For example, here's a "**third-order**" polynomial model:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

You can add as many polynomials as you want. Each one has another coefficient that must be estimated, and each one increases the complexity of the model.

1.5.2 Analysis

In Project 1, we fitted and plotted 3 different polynomial linear regressions.

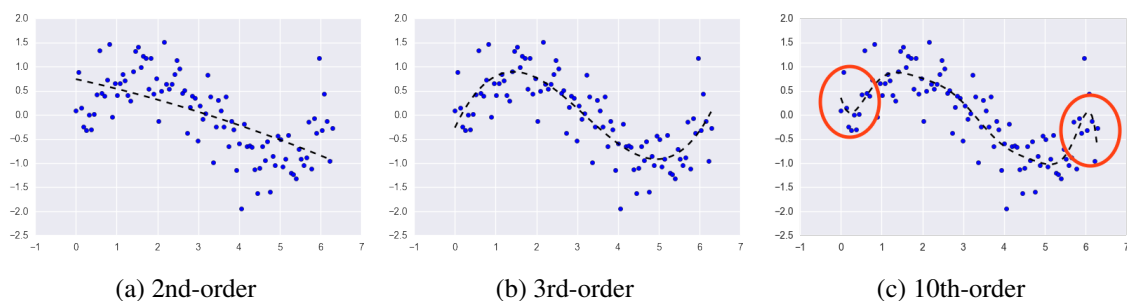


Figure 1.7: Polynomial linear regression

In the 2nd-order polynomial regression, there's a slight bend in the line (you may have to squint to see it).

- Its formula is $\hat{y} = \beta_0 + \beta_1x + \beta_2x^2$
- It's moving in the right direction, but the model is still too inflexible.

The 3rd-order polynomial regression actually looks pretty good.

- Its formula is $\hat{y} = \beta_0 + \beta_1x + \beta_2x^2 + \beta_3x^3$
- It actually approximates our original sine wave (a.k.a. mapping function, a.k.a. signal, a.k.a. "**true underlying relationship**") very well.
- This is probably the best level of complexity for our model.

The 10th-order polynomial regression is **overfit** to the dataset.

- Its formula is $\hat{y} = \beta_0 + \beta_1x + \beta_2x^2 + \dots + \beta_9x^9 + \beta_{10}x^{10}$
- See how the predictions line curves in at the edge values of x ?
- By giving it so much complexity, it has actually learned the random **noise**.
- Therefore, the 10th-order polynomial model is too complex for this dataset.

1.6 Decision Trees

Finally, just for fun, let's see what happens when you crank model complexity way up.

1.6.1 Complexity overload

For that, we'll use **decision trees**.

Intuitively, decision trees "learn" by creating **splits in the data** to separate subsets of the data based on your target variable.

For example, this is a simple one for classification based on Iris dataset:

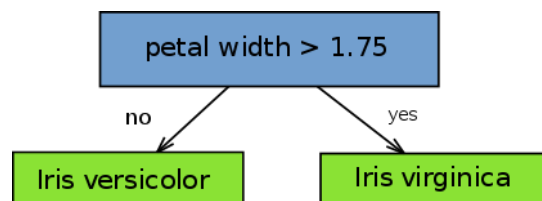


Figure 1.8: Decision tree (technically a "decision stump")

All it's saying is that if the petal width is over 1.75, predict 'virginica' for the species, otherwise predict 'versicolor'.

That seems pretty simple... so why is this consider a **complex** type of model?

- Well, in that Iris example, we've limited the tree to one level (therefore, it only has one split).
- Theoretically, the number of levels (and thus the number of splits) can be **infinite**.
- Plus, this "branching" mechanism can represent **non-linear relationships**.

```
# Decision tree from Scikit-Learn  
from sklearn.tree import DecisionTreeRegressor
```

1.6.2 Analysis

Let's fit and plot and **unconstrained** decision tree on this dataset.

It's "unconstrained" because we won't limit the depth the tree. We will let it grow to its little heart's content!

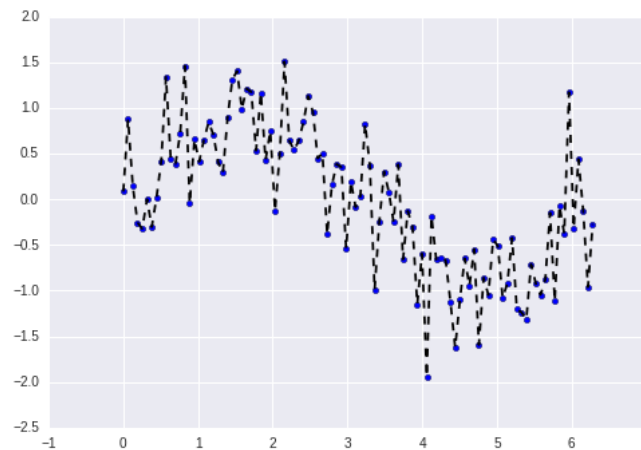


Figure 1.9: Decision tree

Ok, that's clearly very **overfit**.

- Because the decision tree was unconstrained, it was allowed to make as many splits as it wanted to.
- As a result, *it kept making branches until each input feature x value was in its own leaf!*
- In other words, it completely memorized the noise instead of isolating the signal.

1.6.3 Tree ensembles

Fortunately, as it turns out, there are ways to **take advantage of the complexity/flexibility of decision trees while reducing the chance of overfitting**.

This can be accomplished by combining predictions from 100's of **constrained** decision trees into a single model.

These models are called **tree ensembles**, and they perform very well in practice.

- These are methods that fit many (usually 100+) decision trees and combine their predictions.
- They tend to perform well across many types of problems.
- We'll cover them in much more detail later.

2. Exploratory Analysis

We recommend always completing a few essential exploratory analysis steps at the start of any project.

The purpose is the "**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 model from good to great).
3. You'll get a "feel" for the dataset, which will help you communicate results and deliver greater business 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 an infinite number of possible plots, charts, and tables, but you only need a **handful** to "get to know" the data well enough to work with it.

We'll review what those are in this chapter.

- Think of this as a "first date" with the dataset.
- Pareto Principle: We'll try to use 20% of the effort to learn 80% of what we need.
- You'll do ad-hoc data exploration later anyway, so you don't need to be 100% comprehensive right now.

2.1 Basic Information

The first step to understanding your dataset is to display its basic information.

2.1.1 Shape

First, display the dimensions of the dataset.

```
# Dataframe dimensions
df.shape
```

The `shape` attribute returns a tuple of length 2.

- The first value is the number of rows, or **observations**.
- The second value is the number of columns, or **features**.
- For Supervised Learning, one of those features is the **target variable**.

This is an easy way to confirm how much data you have.

2.1.2 Datatypes

Next, it's a good idea to confirm the data types of your features.

```
# Column datatypes
df.dtypes

# Filter and display only df.dtypes that are 'object'
df.dtypes[df.dtypes == 'object']
```

The `dtypes` attribute returns the data types of each feature.

- Those with `int64` or `float64` are **numeric** features.
- Those with `object` are **categorical** features.

This quick check can give you a heads up if any features have incorrect data types. Sometimes, when you transfer data from one environment to another (e.g. from SQL to Python), some features may be imported as incorrect data types.

If you have a **data dictionary**, this is a good time to cross-reference it and make sure each column was imported correctly.

2.1.3 Example observations

Finally, display some example observations from the dataset. This will give you a "feel" for the kind of data in each feature, and it's a good way to confirm that the data types are indeed all correct.

```
# First 5 rows
```

```
df.head()

# First 10 rows
df.head(10)

# Last 10 rows
df.tail(10)
```

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?
- What types of classes are there in the categorical features?

Often, it can helpful to display the first 10 rows of data instead of only the first 5. You can do so by passing an argument into the `head()` function.

Finally, it can also save you a lot of trouble down the road if you get in the habit of looking at the last 10 rows of data. Specifically, you should look for **corrupted data** hiding at the very end.

2.2 Distributions of numeric features

One of the most enlightening data exploration tasks is plotting the distributions of your features.

2.2.1 Histogram grid

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

```
# Plot histogram grid
df.hist(figsize=(14,14), xrot=-45)

# Clear the text "residue"
plt.show()
```

Here are a few things to look out for:

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

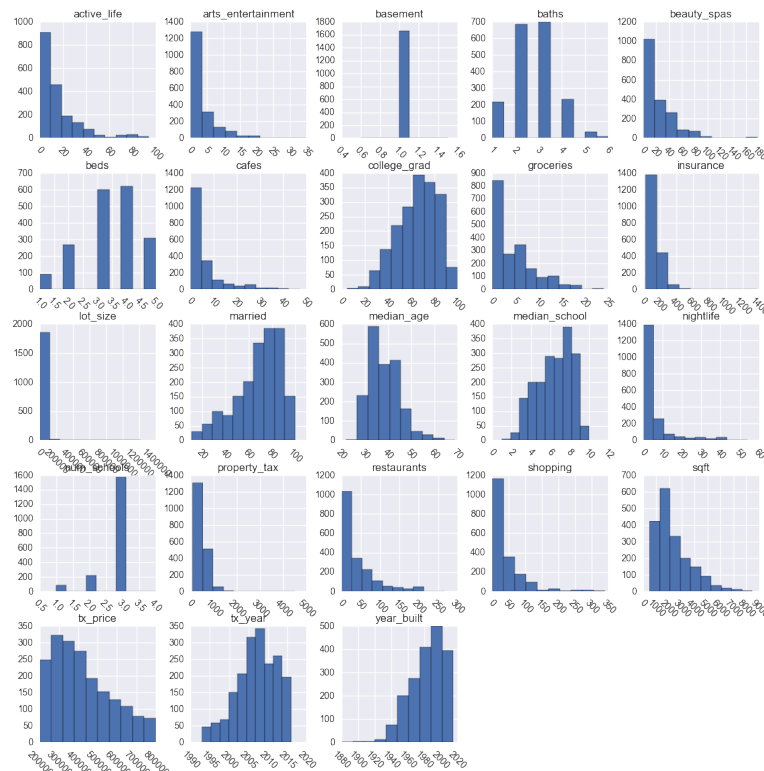


Figure 2.1: Histogram grid (Project 2)

2.2.2 Summary statistics

However, sometimes you may need to see formal **summary statistics**, which provide information such as means, standard deviations, and quartiles.

```
# Summarize numeric features
df.describe()
```

	tx_price	beds	baths	sqft	year_built	lot_size	basement	restaurants
count	1883.000000	1883.000000	1883.000000	1883.000000	1883.000000	1.883000e+03	1657.0	1883.000000
mean	422839.807754	3.420605	2.579926	2329.398832	1982.963887	1.339262e+04	1.0	40.210303
std	151462.593276	1.068554	0.945576	1336.991858	20.295945	4.494930e+04	0.0	46.867012
min	200000.000000	1.000000	1.000000	500.000000	1880.000000	0.000000e+00	1.0	0.000000
25%	300000.000000	3.000000	2.000000	1345.000000	1970.000000	1.542000e+03	1.0	7.000000
50%	392000.000000	3.000000	3.000000	1907.000000	1986.000000	6.098000e+03	1.0	23.000000
75%	525000.000000	4.000000	3.000000	3005.000000	2000.000000	1.176100e+04	1.0	58.000000
max	800000.000000	5.000000	6.000000	8450.000000	2015.000000	1.220551e+06	1.0	266.000000

Figure 2.2: Summary statistics of numeric features (Project 2)

It's especially useful to confirm the **boundaries** (min and max) of each feature, just to make sure there are no glaring errors.

2.3 Distributions of categorical features

Next, take a look at the distributions of your categorical features.

2.3.1 Bar plots

Bar plots help you visualize distributions of categorical features.

```
# Summarize categorical features
df.describe(include=['object'])

# Plot bar plot for each categorical feature
for feature in df.dtypes[df.dtypes == 'object'].index:
    sns.countplot(y=feature, data=df)
    plt.show()
```

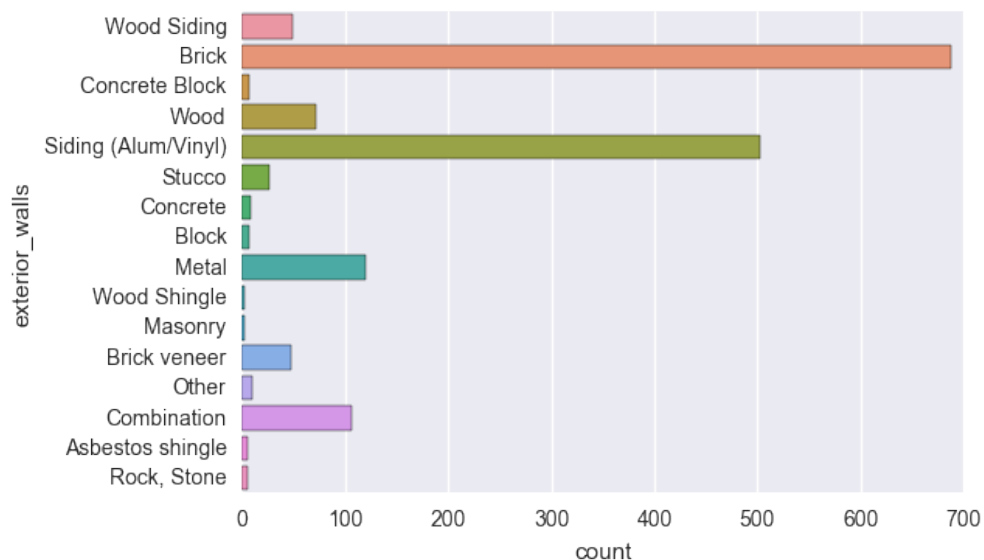


Figure 2.3: Bar plot with sparse classes (Project 2)

2.3.2 Sparse classes

One thing to look out for are **sparse classes**, which are classes that have a very small number of observations.

Figure 2.3 is from Project 2. As you can see, some of the classes (e.g. 'Concrete Block', 'Concrete', 'Block', 'Wood Shingle', etc.) have very short bars. These are sparse classes.

They tend to be problematic when building models.

- In the best case, they don't influence the model much.

- In the worst case, they can cause the model to be overfit.

Therefore, we recommending making a mental note to **combine** or **reassign** some of these classes later.

2.4 Segmentations

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

2.4.1 Box plots

One of the first segmentations you should try is segmenting the **target variable** by key categorical features.

```
# Segment tx_price by property_type and plot distributions
sns.boxplot(y='property_type', x='tx_price', data=df)
```

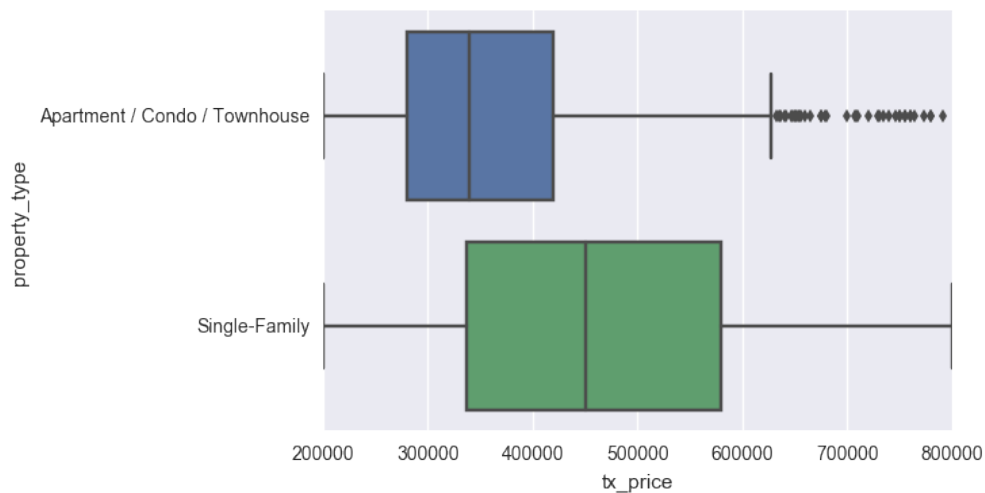


Figure 2.4: Boxplot segmentation (Project 2)

Figure 2.4 is an example from Project 2. As you can see, in general, it looks like single family homes are more expensive.

2.4.2 Groupby

What if you want to compare classes across many features?

- You could plot more bar plots, but you can also use Pandas's `.groupby()` function.
- `.groupby()` can segment by a categorical feature and then calculate a metric (such as mean, median, max, etc.) across **all** numeric features.

```
# Segment by property_type and display class means
df.groupby('property_type').mean()
```

```
# Segment by property_type and display
```

```
#    class means and standard deviations
df.groupby('property_type').agg(['mean', 'std'])
```

	tx_price	beds	baths	sqft	year_built	lot_size	basement
property_type							
Apartment / Condo / Townhouse	366614.034869	2.601494	2.200498	1513.727273	1988.936488	3944.239103	1.0
Single-Family	464644.711111	4.029630	2.862037	2935.865741	1978.523148	20417.666667	1.0

Figure 2.5: Groupby segmentation (Project 2)

In Figure 2.5, we see an example from Project 2. Some questions to consider include:

- On average, which type of property is larger?
- Which type of property is has larger lots?
- Which type of property is in areas with more nightlife options/more restaurants/more grocery stores?
- Do these relationships make intuitive sense, or are any surprising to you?

Finally, you are not limited to only calculating a single metric after performing a groupby. You can use the `.agg()` function to calculate a list of different metrics.

2.5 Correlations

Correlations allow you to look at the **relationships between numeric features and other numeric features**.

2.5.1 Intuition

Correlation is a value between -1 and 1 that represents how closely values for two separate features move in unison.

- **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**.

2.5.2 DataFrame correlations

Pandas DataFrames come with a useful function for calculating correlations: `.corr()`

```
# Calculate correlations between numeric features
correlations = df.corr()
```

This creates a big rectangular dataset. It has the same number of rows and columns, which is just the number of numeric features in the dataset. Each cell is the correlation between the row feature and the column feature.

Things to look out for include:

- Which features are strongly correlated with the target variable?
- Are there interesting or unexpected strong correlations between other features?
- Again, you're primarily looking to gain a better intuitive understanding of the data, which will help you throughout the rest of the project.

2.5.3 Heatmaps

Correlation heatmaps allow you to visualize the correlation grid and make it easier to digest.

When plotting a heatmap of correlations, it's often helpful to do four things:

1. Change the background to white. This way, 0 correlation will show as white
2. Annotate the cell with their correlations values
3. Mask the top triangle (less visual noise)
4. Drop the legend (colorbar on the side)

```
# Change color scheme
sns.set_style("white")

# Generate a mask for the upper triangle
mask = np.zeros_like(correlations, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Make the figsize 9 x 8
plt.figure(figsize=(9,8))

# Plot heatmap of correlations
sns.heatmap(correlations * 100,
            annot=True,
            fmt='%.0f',
            mask=mask,
            cbar=False)
```

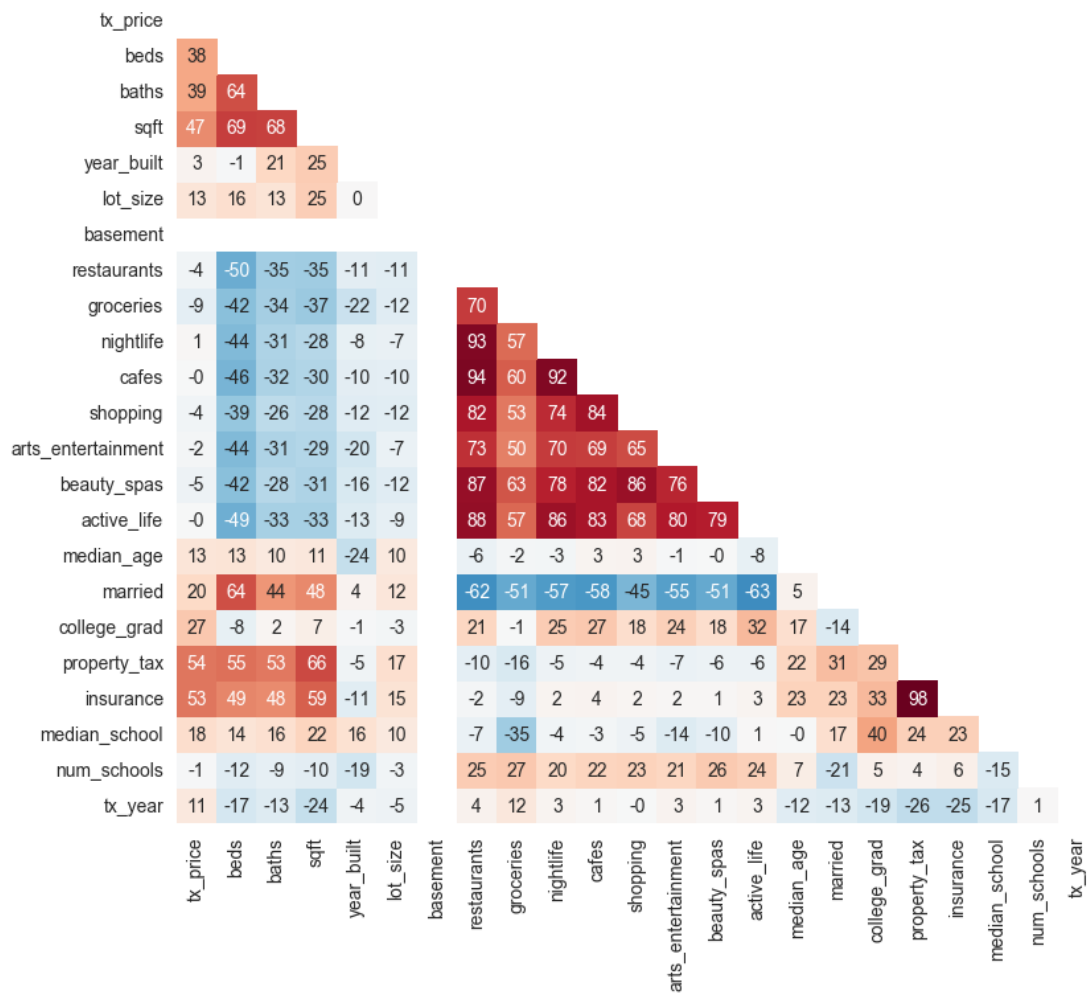


Figure 2.6: Correlations heatmap (Project 2)

3. Data Cleaning

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.

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

- Use this as a "blueprint" for efficient data cleaning.
- Obviously, different types of data will require different types of cleaning.
- However, the systematic approach laid out in this module can always serve as a good starting point.

3.1 Unwanted Observations

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

This includes **duplicate** observations and **irrelevant** observations.

3.1.1 Duplicate

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

```
# Drop duplicates
df = df.drop_duplicates()
```

Fortunately, dropping duplicates is easy, thanks to Pandas's built-in `drop_duplicates()` function.

3.1.2 Irrelevant

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

You should always keep your **project scope** in mind. What types of observations do you care about? What are you trying to model? How will the model be applied?

```
# Drop temporary workers
df = df[df.department != 'temp']
```

This is also a great time to review your charts from the Exploratory Analysis step. You can look at the distribution charts for categorical features to see if there are any classes that shouldn't be there.

For example, in Project 3, we dropped temporary workers from the dataset because we only needed to build a model for permanent, full-time employees.

Checking for irrelevant observations **before engineering features** can save you many headaches down the road.

3.2 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."

3.2.1 Wannabe indicator variables

First, check for variables that should actually be binary indicator variables.

These are variables that **should** be either 0 or 1. However, maybe they were saved under different logic. The `.fillna()` function is very handy here.

```
# Before :
df.basement.unique()
# array([ nan,    1.])

# Missing basement values should be 0
df['basement'] = df.basement.fillna(0)

# After :
df.basement.unique()
# array([ 0.,    1.] )
```

In the example from Project 2, even though NaN represents "missing" values, those are actually meant to indicate properties without basements.

Therefore, we filled them in with the value 0 to turn 'basement' into a true indicator variable.

3.2.2 Typos and capitalization

Next, check for typos or inconsistent capitalization. This is mostly a concern for categorical features.

One easy way to check is by displaying the class distributions for your categorical features (which you've probably already done during Exploratory Analysis).

```
# Class distributions for 'roof'
sns.countplot(y='roof', data=df)

# 'composition' should be 'Composition'
df.roof.replace('composition', 'Composition', inplace=True)

# 'asphalt' should be 'Asphalt'
df.roof.replace('asphalt', 'Asphalt', inplace=True)

# 'shake-shingle' and 'asphalt , shake-shingle '
```

```
#      should be 'Shake Shingle'
df.roof.replace(['shake-shingle', 'asphalt,shake-shingle'],
                'Shake Shingle', inplace=True)
```

These types of errors can be fixed easily with the `.replace()` function.

- The first argument is the class to replace. As you can see in the third line of code, this can also be a list of classes.
- The second argument is the new class label to use instead.
- The third argument, `inplace=True`, tells Pandas to change the original column (instead of creating a copy).

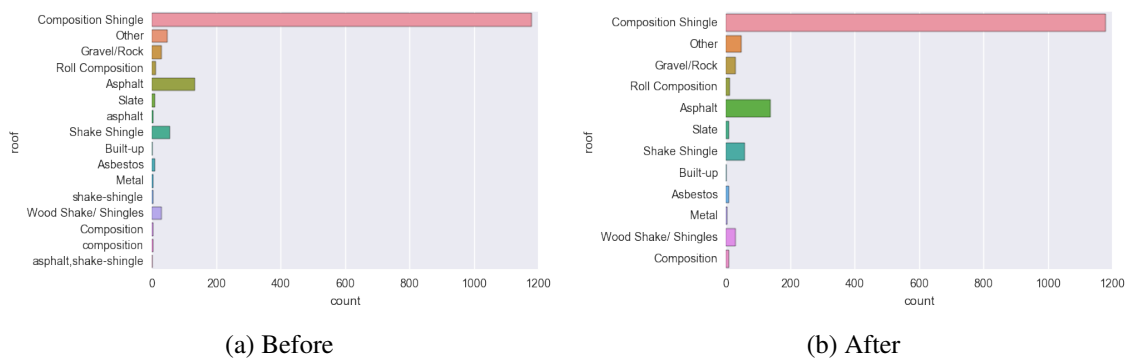


Figure 3.1: Replacing typos and capitalization errors (Project 2)

3.2.3 Mislabeled classes

Finally, check for classes that are labeled as separate classes when they 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.

```
# 'information_technology' should be 'IT'
df.department.replace('information_technology', 'IT',
                      inplace=True)
```

Again, a quick and easy way to check is by plotting class distributions for your categorical features.

3.3 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.

3.3.1 Innocent until proven guilty

However, you need justification for removal. 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. Good reasons include:

1. Suspicious measurements that are unlikely to be real data.
2. Outliers that belong in a different population.
3. Outliers that belong to a different problem.

3.3.2 Violin plots

To check for *potential* outliers, you can use violin plots. Violin plots serve the same purpose as box plots, but they provide more information.

- A box plot only shows **summary statistics** such as median and interquartile range.
- A violin plot shows the entire **probability distribution** of the data. (In case you're wondering, you don't need a statistics degree to interpret them.)

```
# Violin plot of 'sqft'
sns.violinplot(df.sqft)

# Violin plot of 'lot_size'
sns.violinplot(df.lot_size)
```

Here's how to interpret violin plots:

- In a violin plot, the **thin horizontal bar** represents the range (min to max) of the data.
- The **thick horizontal bar** is the interquartile range (25th percentile to 75th percentile).
- The **white dot** is the median.
- Finally, the **thickness of the "violin"** represents the estimated probability density.

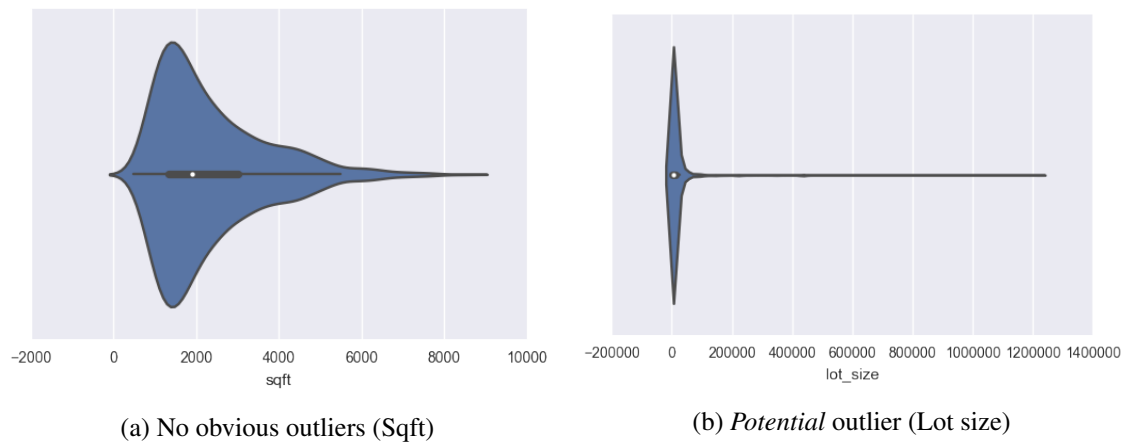


Figure 3.2: Violin plots (Project 2)

In Figure 3.2, we see an example of a violin plot that shows a potential outlier. The violin plot for `'lot_size'` has a long and skinny tail.

3.3.3 Manual check

After identifying potential outliers with violin plots, you should do a manual check by displaying those observations.

```
# Sort df.lot_size and display the top 5 observations
df.lot_size.sort_values(ascending=False).head()

# Remove lot_size outliers
df = df[df.lot_size <= 500000]
```

Then, you can use a **boolean mask** to remove outliers by filtering to only keep wanted observations.

3.4 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 Scikit-Learn algorithms do not accept missing values.

3.4.1 "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 values from 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.



Figure 3.3: *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 always want to tell your algorithm that a value was missing because **missingness is informative**. The two most commonly suggested ways of handling missing data don't do that.

3.4.2 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 Scikit-Learn technical requirement for no missing values.

```
# Display number of missing values by feature (categorical)
df.select_dtypes(include=['object']).isnull().sum()

# Fill missing values in exterior_walls with 'Missing'
df['exterior_walls'] = df['exterior_walls'].fillna('Missing')

# Fill missing categorical values
for column in df.select_dtypes(include=['object']):
    df[column] = df[column].fillna('Missing')
```

Sometimes if you have many categorical features, it may be more convenient to write a loop that labels missing values in all of them.

3.4.3 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 Scikit-Learn's technical requirement of no missing values.

```
# Display number of missing values by feature (numeric)
df.select_dtypes(exclude=['object']).isnull().sum()

# Indicator variable for missing last_evaluation
df['last_evaluation_missing'] =
    df.last_evaluation.isnull().astype(int)

# Fill missing values in last_evaluation with 0
df.last_evaluation.fillna(0, inplace=True)
```

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.

When used properly, this technique almost always performs better than imputation methods in practice.

4. Feature Engineering

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

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!

Now, we will tour various types of features that you can engineer, and we will introduce several **heuristics** to help spark new ideas.

This is not an exhaustive compendium of all feature engineering because there are limitless possibilities for this step.

Finally, we want to emphasize that this skill will naturally improve as you gain more experience.

- Use this as a "checklist" to seed ideas for feature engineering.
- There are limitless possibilities for this step, and you can get pretty creative.
- This skill will improve naturally as you gain more experience and domain expertise!

4.1 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**. You have a lot of "creative freedom" to think of ideas for new features.

4.1.1 Boolean masks

Boolean masks allow you to easily create indicator variables based on conditionals. They empower you to specifically isolate information that you *suspect* might be important for the model.

```
# Indicator variable for properties with 2 beds and 2 baths
df['two_and_two'] = ((df.beds == 2) &
                     (df.baths == 2)).astype(int)

# Display percent of rows where two_and_two == 1
df.two_and_two.mean()

# Indicator feature for transactions between 2010 and 2013
df.tx_year.between(2010, 2013).astype(int)
```

You can also quickly check the proportion of observations that meet the condition.

4.1.2 Link with Exploratory Analysis

Of course, we won't always have a lot of domain knowledge for the problem. In these situations, we should rely on exploratory analysis to provide us hints.

Figure 4.1 shows a scatterplot from Project 3. From the plot, we can see that there are 3 clusters of people who've left.

1. First, we have people with high '`last_evaluation`' but low '`satisfaction`'. Maybe these people were overqualified, frustrated, or **unhappy** in some other way.
2. Next, we have people with low '`last_evaluation`' and medium '`satisfaction`'. These were probably **underperformers** or poor cultural fits.
3. Finally, we have people with high '`last_evaluation`' and high '`satisfaction`'. Perhaps these were **overachievers** who found better offers elsewhere.

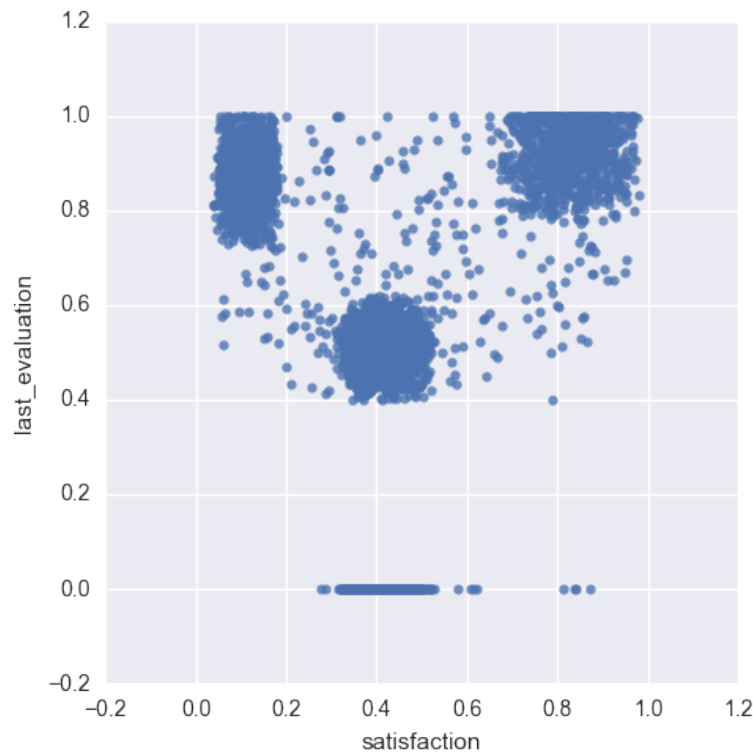


Figure 4.1: Scatterplot from Exploratory Analysis (Project 3)

These roughly translated to 3 **indicator features** that we engineered.

```
# Create indicator features
df['underperformer'] = ((df.last_evaluation < 0.6) &
                        (df.last_evaluation_missing == 0)).astype(int)

df['unhappy'] = (df.satisfaction < 0.2).astype(int)

df['overachiever'] = ((df.last_evaluation > 0.8) &
                      (df.satisfaction > 0.7)).astype(int)
```

As you can tell, "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 ideas.

4.2 Interaction Features

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

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

4.2.1 Examples

Here are a few examples from the projects.

```
# Property age feature (Project 2)
df['property_age'] = df.tx_year - df.year_built

# School score feature (Project 2)
df['school_score'] = df.num_schools * df.median_school

# Sales feature (Project 4 - transaction level)
df['Sales'] = df.Quantity * df.UnitPrice
```

In Project 2, we knew the transaction year and the year the property was built in. However, the more useful piece of information that *combining* these two features provided is the **age of the property at the time of the transaction**.

In Project 2, we also knew the the number of schools nearby and their median quality score. However, we suspected that what's really important is **having many school options, but only if they are good**.

In Project 4, we knew the number of units per order (Quantity) and the price per unit (UnitPrice), but what we really needed was the **actual amount paid for the order (Sales)**.

4.3 Sparse Classes

The next heuristic we'll consider is grouping sparse classes together (in our categorical features).

Sparse classes are those that have very few total observations. They can be problematic for certain machine learning algorithms because they can cause 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 the each one has at least around 50 observations. As with any "rule" of thumb, use this as a guideline (not actually as a *rule*).

4.3.1 Similar classes

To begin, we can group classes that are pretty similar in reality. Again, the `.replace()` function comes in handy.

```
# Group 'Wood Siding' and 'Wood Shingle' with 'Wood'
df.exterior_walls.replace(['Wood Siding', 'Wood Shingle'],
                           'Wood', inplace=True)
```

For example, in Project 2, we decided to group 'Wood Siding', 'Wood Shingle', and 'Wood' into a single class. In fact, we just labeled all of them as 'Wood'.

4.3.2 Other classes

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

```
# List of classes to group
other_exterior_walls = ['Concrete Block', 'Stucco', 'Masonry',
                        'Other', 'Asbestos shingle']

# Group other classes into 'Other'
df.exterior_walls.replace(other_exterior_walls, 'Other',
                           inplace=True)
```

For example, in Project 2, we labeled 'Stucco', 'Other', 'Asbestos shingle', 'Concrete Block', and 'Masonry' as 'Other'.

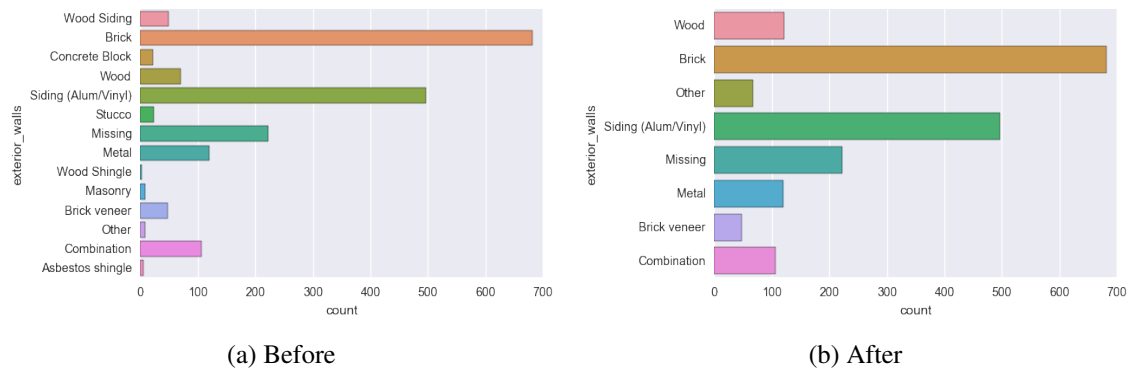


Figure 4.2: Grouping sparse classes (Project 2)

Figure 4.2 shows the "before" and "after" class distributions for one of the categorical features from Project 2. 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.

4.4 Dummy Variables

Scikit-Learn 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) features 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 Scikit-Learn’s technical requirement.

4.4.1 Example: Project 2

In Project 2, after grouping sparse classes for the 'roof' feature, we were left with 5 classes:

- 'Missing'
- 'Composition Shingle'
- 'Other'
- 'Asphalt'
- 'Shake Shingle'

These translated to 5 dummy variables:

- 'roof_Missing'
- 'roof_Composition Shingle'
- 'roof_Other'
- 'roof_Aspphalt'
- 'roof_Shake Shingle'

So if an observation in the dataset had a roof made of 'Composition Shingle', it would have the following values for the dummy variables:

- 'roof_Missing' = 0
- 'roof_Composition Shingle' = 1
- 'roof_Other' = 0
- 'roof_Asphalt' = 0
- 'roof_Shake Shingle' = 0

4.4.2 Get dummies

Creating a dataframe with dummy features is as easy calling the `.get_dummies()` function and passing in a list of the features you'd like to create dummy variables for.

```
# Create new dataframe with dummy features
df = pd.get_dummies(df, columns=['exterior_walls',
                                  'roof',
                                  'property_type'])
```


4.5 Remove Unused

Finally, remove unused or redundant features from the dataset.

4.5.1 Unused features

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

4.5.2 Redundant features

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

For example, in Project 2, since we used `'tx_year'` and `'year_built'` to create the `'property_age'` feature, we ended up removing them.

- Removing `'tx_year'` was also a good idea because we didn't want our model being overfit to the transaction year.
- Since we'll be applying it to future transactions, we wanted the algorithms to focus on learning patterns from the other features.

Sometimes there's no clear right or wrong decision for this step, and that's OK.

```
# Drop 'tx_year' and 'year_built' from the dataset
df = df.drop(['tx_year', 'year_built'], axis=1)
```

4.5.3 Analytical base table

Finally, save the new DataFrame that you've cleaned and then augmented through feature engineering.

We prefer to name it the **analytical base table** because it's what you'll be building your models on.

```
# Save analytical base table
df.to_csv('analytical_base_table.csv', index=None)
```

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.

4.6 Rolling Up

For some problems, you'll need to roll up the data into a higher level of granularity through a process called data wrangling.

During this process, you'll aggregate new features. You practiced this in Project 4, and we review it in [Appendix A.2 - Data Wrangling](#).

5. Model Training

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 3 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**.

We'll set up our entire modeling process to allow our algorithms as much flexibility as possible to learn from the data while **safeguarding against overfitting**.

- This model training process works for almost any machine learning problem because it's based on strong fundamentals (Pass, Catch, Shoot).
- As you'll see, it's designed to maximize speed, efficiency, and especially **consistency**.
- It swaps algorithms and automatically finds the best parameters for each one (because algorithms are commodities).

5.1 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 to decide if it's good or not.
- 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 it! Remember, a model should be judged on its **ability to predict new, unseen data**.

5.1.1 Training and test sets

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.

```
# Function for splitting training and test set
from sklearn.model_selection import train_test_split

# Create separate object for target variable
y = df.tx_price

# Create separate object for input features
X = df.drop('tx_price', axis=1)

# Split X and y into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=1234)
```

You should always split your data *before* you begin building models.

- 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.

5.1.2 Cross-validation (intuition)

Splitting our data into training and test sets will help us get a reliable estimate of model performance on unseen data, which will help us avoid overfitting by picking a **generalizable** model.

Next, it's time to introduce a concept that will help us **tune** (i.e. pick the best "settings" for) our models: cross-validation.

Cross-validation is a method for getting a reliable estimate of model performance using *only your training data*. By combining this technique and the train/test split, you can tune your models without using your test set, saving it as truly unseen data.

5.1.3 10-fold cross-validation

Cross-validation refers to a family of techniques that serve the same purpose. One way to distinguish the techniques is by the number of **folds** you create.

The most common one, **10-fold**, 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 remaining "hold-out" fold (e.g. the last 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 turns out to be pretty reliable.

5.2 Model Pipelines

Up to now, you've explored the dataset, cleaned it, and engineered new features.

Most importantly, you created an excellent analytical base table that has set us up for success.

5.2.1 Preprocessing

However, sometimes we'll want to **preprocess** the training data even more before feeding it into our algorithms.

For example, we may want to...

- Transform or scale our features.
- Perform automatic feature reduction (e.g. PCA).
- Remove correlated features.

The key is that these types of preprocessing steps should be performed **inside the cross-validation loop**.

5.2.2 Standardization

Standardization is the most popular preprocessing technique in machine learning. It transforms all of your features to the same **scale** by subtracting means and dividing by standard deviations.

This is a useful preprocessing step because some machine learning algorithms will "overemphasize" features that are on larger scales. Standardizing a feature is pretty straightforward:

1. First, subtract its mean from each of its observations.
2. Then, divide each of its observations by its standard deviation.

This makes the feature's distribution **centered around zero, with unit variance**. To standardize the entire dataset, you'd simply do this for each feature.

5.2.3 Preprocessing parameters

To perform standardization, we needed 2 pieces of information directly from the training set:

1. The means of each feature
2. The standard deviations of each feature

These are called **preprocessing parameters** because they actually **needed to be learned** from the training data.

So if we want to preprocess new data (such as our test set) in the exact same way, we'd need to use the *exact same preprocessing parameters that we learned from the training set* (not those from the test set!).

- Mathematically speaking, this preserves the exact original transformation.

- Practically speaking, if new observations come in 1 at a time, you can't standardize them without already having preprocessing parameters to use (i.e. those learned from your training set).

5.2.4 Pipelines and cross-validation

So WTF is a model "pipeline?"

As you'll recall, we'll be tuning (i.e. finding the best settings for) our model with cross-validation.

- We'll split the training set into 10 different folds.
- Therefore, when we apply preprocessing, *we should only learn preprocessing parameters from the 9 training folds in each iteration.*
- In other words, we need to store our preprocessing steps in a "model pipeline" and perform cross-validation on that entire pipeline.

Here's how the new 10-fold cross-validation process would look (the new steps are **bolded**):

1. Split your data into 10 equal folds.
2. **Preprocess 9 training folds, learning the preprocessing parameters.**
3. Train your model on those newly preprocessed 9 folds.
4. **Preprocess the hold-out fold using the same preprocessing parameters from the training folds.**
5. Evaluate your model on the newly preprocessed hold-out fold.
6. Perform steps **(2) - (5)** 10 times, each time holding out a different fold.
7. Average the performance across all 10 folds.

5.2.5 Pipeline dictionaries

Now, what you just learned is a sophisticated modeling process that has been battle-tested and proven in the field. Luckily, you won't have to implement it from scratch!

Scikit-Learn comes with a handy `make_pipeline()` function that helps you glue these steps together. (See? Told you Scikit-Learn comes with everything!)

```
# Function for creating model pipelines
from sklearn.pipeline import make_pipeline

# For standardization
from sklearn.preprocessing import StandardScaler

# Single pipeline
make_pipeline(StandardScaler(), Lasso(random_state=123))
```

However, we recommend storing all your model pipelines in a single **dictionary**, just so they're better organized.

```
# Create pipelines dictionary
pipelines = {
    'lasso' : make_pipeline(StandardScaler(),
                           Lasso(random_state=123)),
    'ridge' : make_pipeline(StandardScaler(),
                           Ridge(random_state=123))
}

# Add a pipeline for 'enet'
pipelines['enet'] = make_pipeline(StandardScaler(),
                                  ElasticNet(random_state=123))
```

Finally, note that we set a consistent `random_state` to ensure replicable results.

5.3 Declare Hyperparameters

Up to now, 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**.

5.3.1 Model parameters vs. hyperparameters

There are two types of parameters we need to worry about when using machine learning algorithms.

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 express "higher-level" structural settings for modeling 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 training the model because they cannot be learned from the data.

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

5.3.2 Hyperparameter grids (regression)

Most algorithms have many different options of hyperparameters to tune. Fortunately, for each algorithm, typically only a few hyperparameters really influence model performance.

You'll declare the hyperparameters to tune in a dictionary, which is called a "**hyperparameter grid**".

See below for recommended values to try.

```
# Lasso hyperparameters
lasso_hyperparameters = {
    'lasso__alpha' : [0.001, 0.005, 0.01,
                      0.05, 0.1, 0.5, 1, 5, 10]
}

# Ridge hyperparameters
ridge_hyperparameters = {
    'ridge__alpha': [0.001, 0.005, 0.01,
                     0.05, 0.1, 0.5, 1, 5, 10]
}

# Elastic Net hyperparameters
```

```

enet_hyperparameters = {
    'elasticnet__alpha': [0.001, 0.005, 0.01,
                          0.05, 0.1, 0.5, 1, 5, 10],
    'elasticnet__l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9]
}

# Random forest hyperparameters
rf_hyperparameters = {
    'randomforestregressor__n_estimators': [100, 200],
    'randomforestregressor__max_features': ['auto', 'sqrt',
                                             0.33],
}

# Boosted tree hyperparameters
gb_hyperparameters = {
    'gradientboostingregressor__n_estimators': [100, 200],
    'gradientboostingregressor__learning_rate': [0.05, 0.1,
                                                  0.2],
    'gradientboostingregressor__max_depth': [1, 3, 5]
}

```

5.3.3 Hyperparameter grids (classification)

Classification has analogous algorithms to regression.

See below for recommended values to try.

```

# Logistic Regression hyperparameters
l1_hyperparameters = {
    'logisticregression__C': np.linspace(1e-3, 1e3, 10),
}

l2_hyperparameters = {
    'logisticregression__C': np.linspace(1e-3, 1e3, 10),
}

# Random Forest hyperparameters
rf_hyperparameters = {
    'randomforestclassifier__n_estimators': [100, 200],
    'randomforestclassifier__max_features': ['auto', 'sqrt',
                                             0.33]
}

# Boosted Tree hyperparameters
gb_hyperparameters = {

```

```
'gradientboostingclassifier__n_estimators': [100, 200],  
'gradientboostingclassifier__learning_rate': [0.05, 0.1,  
                                              0.2],  
'gradientboostingclassifier__max_depth': [1, 3, 5]  
}
```

5.3.4 Hyperparameter dictionary

Just as with the model pipelines, we recommend storing all your hyperparameter grids in a single dictionary to stay organized.

```
# Create hyperparameters dictionary  
hyperparameters = {  
    'rf' : rf_hyperparameters,  
    'gb' : gb_hyperparameters,  
    'lasso' : lasso_hyperparameters,  
    'ridge' : ridge_hyperparameters,  
    'enet' : enet_hyperparameters  
}
```

5.4 Fit and tune models

Now that we have our `pipelines` and `hyperparameters` dictionaries declared, we're ready to tune our models with cross-validation.

5.4.1 Grid search

Remember the cross-validation loop that we discussed earlier? Well `GridSearchCV` essentially performs that entire loop for each combination of values in the hyperparameter grid (i.e. it wraps another loop around it).

- As its name implies, this class performs cross-validation on a **hyperparameter grid**.
- It will try each **combination of values** in the grid.
- It neatly wraps the entire cross-validation process together.

It then calculates **cross-validated scores** (using performance metrics) for each combination of hyperparameter values and picks the combination that has the best score.

```
# Helper for cross-validation
from sklearn.model_selection import GridSearchCV

# Create cross-validation object from Lasso pipeline
# and Lasso hyperparameters
model = GridSearchCV(pipelines['lasso'],
                     hyperparameters['lasso'],
                     cv=10,
                     n_jobs=-1)

# Fit and tune model
model.fit(X_train, y_train)
```

5.4.2 Looping through dictionaries

Because we already set up and organized our `pipelines` and `hyperparameters` dictionaries, we can fit and tune models with all of our algorithms in a single loop.

```
# Create empty dictionary called fitted_models
fitted_models = {}

# Loop through model pipelines ,
# tuning each one and saving it to fitted_models
for name, pipeline in pipelines.items():
    # Create cross-validation object
    model = GridSearchCV(pipeline, hyperparameters[name],
                        cv=10, n_jobs=-1)
```

```
# Fit model on X_train , y_train
model.fit(X_train, y_train)

# Store model in fitted_models[name]
fitted_models[name] = model

# Print '{name} has been fitted '
print(name, 'has been fitted.')
```

This step can take a few minutes, depending on the speed and memory of your computer.

5.5 Select Winner

Finally, it's time to evaluate our models and pick the best one.

Regression models are generally more straightforward to evaluate than classification models.

5.5.1 Cross-validated score

One of the first ways to evaluate your models is by looking at their cross-validated score on the training set.

```
# Display best_score_ for each fitted model
for name, model in fitted_models.items():
    print( name, model.best_score_ )
```

These scores represent different metrics depending on your machine learning task.

5.5.2 Performance metrics (regression)

Holdout R^2 : For regression, the default scoring metric is the average R^2 on the holdout folds.

- In rough terms, R^2 is the "percent of the variation in the target variable that can be explained by the model."
- Because is the average R^2 from the *holdout folds*, higher is almost always better.
- That's all you'll need to know for this course, but there are plenty of additional explanations of this metric that you can find online.

Mean absolute error (MAE): Another metric we looked at was mean absolute error, or MAE.

- Mean absolute error (or MAE) is the average absolute difference between predicted and actual values for our target variable.
- It has the added benefit that it's very easily interpretable. That means it's also easier to communicate your results with key stakeholders.

```
# Import r2_score and mean_absolute_error functions
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
```

To evaluate our models, we also want to see their performance on the test set, and not just their cross-validated scores. These functions from Scikit-Learn help us do so.

5.5.3 Performance metrics (classification)

Holdout accuracy: For classification, the default scoring metric is the average accuracy on the holdout folds.

- Accuracy is simply the percent of observations correctly classified by the model.

- Because is the average accuracy from the *holdout folds*, higher is almost always better.
- However, straight accuracy is not always the best way to evaluate a classification model, especially when you have **imbalanced classes**.

Area under ROC curve (AUROC): Area under ROC curve is the most reliable metric for classification tasks.

- It's equivalent to the probability that a randomly chosen Positive observation ranks higher (has a higher predicted probability) than a randomly chosen Negative observation.
- Basically, it's saying... if you grabbed two observations and exactly one of them was the positive class and one of them was the negative class, what's the likelihood that your model can distinguish the two?
- Therefore, it doesn't care about imbalanced classes.
- See [Appendix A.1 - Area Under ROC Curve](#) for a more detailed explanation.

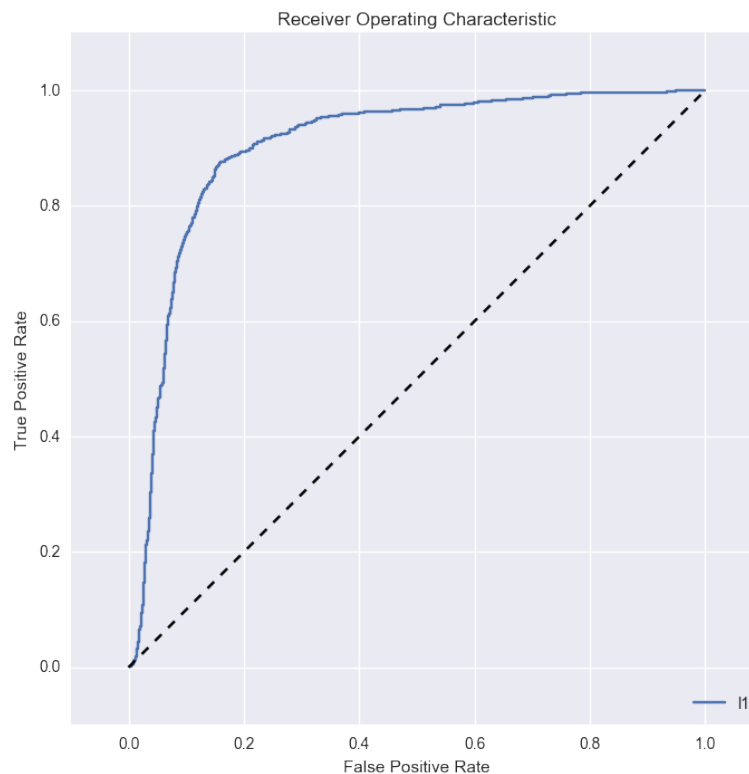


Figure 5.1: ROC Curve (Project 3)

```
# Classification metrics
from sklearn.metrics import roc_curve, auc
```

5.5.4 Test performance

After importing the performance metric functions, you can write a loop to apply them to the test set.

Performance on the test set is the best indicator of whether a model is **generalizable** or not.

```
# Test performance
for name, model in fitted_models.items():
    pred = model.predict(X_test)
    print( name )
    print( '-----' )
    print( 'R^2:', r2_score(y_test, pred ))
    print( 'MAE:', mean_absolute_error(y_test, pred))
    print()
```

These questions help you pick the winning model:

- Which model had the best performance on the test set?
- Does it perform well across various performance metrics?
- Did it also have (one of) the best cross-validated scores?
- Does it satisfy the win condition?

5.5.5 Saving the winning model

After you pick your winner, saving it is pretty straightforward.

We recommend directly saving the model `Pipeline` object instead of saving the entire `GridSearchCV` object.

```
# Pickle package
import pickle

# Save winning pipeline
with open('final_model.pkl', 'wb') as f:
    pickle.dump(fitted_models['rf'].best_estimator_, f)
```

6. Project Delivery

You've come a long way up to this point. You've taken this project from a simple dataset all the way to a high-performing predictive model.

Now, we'll show you how you can use your model to predict new (**raw**) data and package your work together into an executable script that can be called from the command line.



We promise project delivery will be much easier than this...

- How to confirm your model was saved correctly.
- How to apply your model to raw data.
- How to package your work into a single, versatile executable script.

6.1.3 Predict the test set again

Finally, you can use your model to predict `X_test` again and confirm if you get the same scores for your performance metrics (which will be different between classification and regression).

```
# Predict X_test
pred = model.predict_proba(X_test)

# Get just the prediction for the positive class (1)
pred = [p[1] for p in pred]

# Print AUROC
print( 'AUROC:', roc_auc_score(y_test, pred) )
```

6.2 Pre-Modeling Functions

If new data arrives in the same format as the original **raw data**, we wouldn't be able to apply our model to it in the same way. We'd need to first clean the new data the same way and engineer the same features.

All we need to do is write a few functions to **convert the raw data to the same format as the analytical base table**.

- That means we need to bundle together our data cleaning steps.
- Then we need to bundle together our feature engineering steps.
- We can skip the exploratory analysis steps because we didn't alter our dataframe then.

6.2.1 Data cleaning function

The key is to only include steps that altered your dataframe. Here's the example from Project 3:

```
def clean_data(df):
    # Drop duplicates
    df = df.drop_duplicates()

    # Drop temporary workers
    df = df[df.department != 'temp']

    # Missing filed_complaint values should be 0
    df['filed_complaint'] = df.filed_complaint.fillna(0)

    # Missing recently_promoted values should be 0
    df['recently_promoted'] = df.recently_promoted.fillna(0)

    # 'information_technology' should be 'IT'
    df.department.replace('information_technology', 'IT',
                          inplace=True)

    # Fill missing values in department with 'Missing'
    df['department'].fillna('Missing', inplace=True)

    # Indicator variable for missing last_evaluation
    df['last_evaluation_missing'] =
        df.last_evaluation.isnull().astype(int)

    # Fill missing values in last_evaluation with 0
    df.last_evaluation.fillna(0, inplace=True)

    # Return cleaned dataframe
    return df
```

6.2.2 Feature engineering function

And here's the feature engineering example from Project 3:

```
def engineer_features(df):  
    # Create indicator features  
    df['underperformer'] = ((df.last_evaluation < 0.6) &  
                            (df.last_evaluation_missing == 0)).astype(int)  
  
    df['unhappy'] = (df.satisfaction < 0.2).astype(int)  
  
    df['overachiever'] = ((df.last_evaluation > 0.8) &  
                          (df.satisfaction > 0.7)).astype(int)  
  
    # Create new dataframe with dummy features  
    df = pd.get_dummies(df, columns=['department', 'salary'])  
  
    # Return augmented DataFrame  
    return df
```

6.3 Model Class

Next, we packaged these functions together into a single model class. This is a convenient way to keep all of the logic for a given model in one place.

6.3.1 Python classes

Remember how when we were training our model, we imported `LogisticRegression`, `RandomForestClassifier`, and `GradientBoostingClassifier`?

We called them "algorithms," but they are technically **Python classes**.

Python classes are structures that allow us to group related code, logic, and functions in one place. Those familiar with object-oriented programming will have recognized this concept.

For example, each of those algorithms have the `fit()` and `predict_proba()` functions that allow you to train and apply models, respectively.

6.3.2 Custom class

We can construct our own **custom Python classes** for our models.

- Thankfully, it doesn't need to be nearly as complex as those other algorithm classes because we're not actually using this to train the model.
- Instead, we already have the model saved in a `final_model.pkl` file.
- We only need to include logic for cleaning data, feature engineering, and predicting new observations.

```
class EmployeeRetentionModel:
    def __init__(self, model_location):
        with open(model_location, 'rb') as f:
            self.model = pickle.load(f)

    def predict_proba(self, X_new, clean=True, augment=True):
        if clean:
            X_new = self.clean_data(X_new)

        if augment:
            X_new = self.engineer_features(X_new)

        return X_new, self.model.predict_proba(X_new)

    def clean_data(self, df):
        # ...

    def engineer_features(self, df):
        # ...
```

6.4 Model Deployment

In this course, we covered 2 different ways to deploy your models.

1. Keep it in Jupyter Notebook
2. Port it to an executable script

6.4.1 Jupyter notebook

We're going to start with the **easiest** way to deploy your model: Keep it in Jupyter Notebook.

Not only is this the easiest way, but this is also our recommended way for Data Scientists and Business Analysts who are personally responsible for maintaining and applying the model.

Keeping the model in Jupyter Notebook has several huge benefits.

- It's seamless to update your model.
- You can perform ad-hoc data exploration and visualization.
- You can write detailed comments and documentation, with images, bullet lists, etc.

We didn't choose Jupyter Notebook for teaching this course just because it's a great interactive environment... we also chose it because it's one of the most popular IDE's among professional data scientists.

If you keep your model in Jupyter Notebook, you can directly use the model class you defined earlier.

```
# Initialize an instance
retention_model = EmployeeRetentionModel('final_model.pkl')

# Predict raw data
_, pred1 = retention_model.predict_proba(raw_data,
                                         clean=True, augment=True)

# Predict cleaned data
_, pred2 = retention_model.predict_proba(cleaned_data,
                                         clean=False, augment=True)

# Predict cleaned and augmented data
_, pred3 = retention_model.predict_proba(augmented_data,
                                         clean=False, augment=False)
```

6.4.2 Executable script

However, there will definitely be situations in which Jupyter notebooks are not enough. The most common scenarios are if you need to **integrate** your model with other parts of an application or **automate** it.

For these use cases, you can simply package your model into an **executable script**.

You can call these scripts from any command line program. That means you can:

- Host your model on the cloud (like AWS)
- Integrate it as part of a web application (have other parts of the application call the script)
- Or automate it (schedule recurring jobs to run the script)

Implementing each of those use cases is outside the scope of this course, but we've shown you how to actually set up the script, which is the first step for each of those use cases.

We've included an example script in the [project_files/](#) directory of the **Project 3 Workbook Bundle**.

Once you open it up (you can open it up in Jupyter browser or in any text editor), you'll see that you've already written the libraries to import and the custom model class, including the data cleaning, feature engineering, and predict probability functions.

All that's left is a little bit of logic at the bottom to handle command line requests.

To run the script, you can call it from the command line after navigating to the Workbook Bundle's folder.

```
$ python project_files/retention_model.py project_files/new_data.csv  
predictions.csv final_model.pkl True True
```

This saves a new file that includes the predictions.

nt_procurement	department_product	department_sales	department_support	salary_high	salary_low	salary_medium	prediction
	0	0	0	1	0	0	0.00
	1	0	0	0	1	0	0.02
	0	1	0	0	1	0	0.00
	0	0	0	1	0	0	0.00
	0	0	0	0	1	0	1.00

Figure 6.1: New file with predictions

7. Regression Algorithms

Next, we want to introduce you to 5 useful algorithms for regression tasks:

1. Lasso regression
2. Ridge regression
3. Elastic-Net
4. Random forests
5. Boosted trees

However, remember, *algorithms are commodities*.

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.

7.1 The "Final Boss" of Machine Learning

Before we can get to the algorithms, we need to first cover a few key concepts. Earlier, we introduced model complexity, even calling it the "**heart of machine learning**."

Now we're going to dive deeper into a practical challenge that arises from model complexity: **overfitting**.

7.1.1 How to find the right amount of complexity

As it turns out, for practical machine learning, it's typically better to **allow more complexity in your models, and then implement other safeguards against overfitting**.

- The reason is that if your models are too simple, they can't learn complex patterns no matter how much data you collect.
- However, if your models are more complex, you can use **easy-to-implement tactics** to avoid overfitting. In addition, collecting more data will naturally reduce the chance of overfitting.

And that's why we consider **overfitting** (and not underfitting) the "**final boss**" of machine learning.

7.1.2 How overfitting occurs

Overfitting is a dangerous mistake that causes your model to "**memorize**" the **training data** instead of the true underlying pattern.

- This prevents the model from being **generalizable** to new, unseen data.
- Many of the practical skills from in this course (**regularization, ensembling, train/test splitting, cross-validation, etc.**) help you fight overfitting!
- In fact, our entire model training workflow is designed to defeat overfitting.

However, picking the right algorithms can also prevent overfitting! Just to be clear, there are two main ways models can become **too complex**, leading to overfitting.

1.) The first is the **inherent structure** of different algorithms.

- e.g. Decision trees are more complex than linear regression because they can represent non-linear relationships.
- e.g. Polynomial regression is more complex than simple linear regression because they can represent curvy relationships.

2.) The second is the **number of input features** into your model.

- e.g. A linear regression with 1 input feature is less complex than a linear regression with the same input feature plus 9 others.
- e.g. In a way, third-order polynomial regression is just linear regression plus 2 other features (x^2, x^3) that were engineered from the original one, x .

Both of these two forms of model complexity can lead to overfitting, and we've seen how to deal with both of them.

7.2 Regularization

Next, we'll discuss the first "advanced" tactic for improving the performance of our models. It's considered pretty "advanced" in academic machine learning courses, but we believe it's really pretty easy to understand and implement.

But before we get to it, let's bring back our friend simple linear regression and drag it through the mud a little bit.

7.2.1 Flaws of linear regression

You see, in practice, simple linear regression models **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.

7.2.2 The number of features is too damn high!

Remember, one type of over-complexity occurs if the model has **too many** features relative to the number of observations.



This model is too damn complex!

Let's take an extreme example with simple linear regression:

- 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.

So, how can we reduce the number of features in our model (or dampen their effect)?

The answer is **regularization**.

7.2.3 Cost functions

To understand regularization, we must first introduce the idea of a cost function.

Cost functions measure the error of your predictions for a given task.

- In other words, they quantify how inaccurate you are.
- They help you compare performance across many models.
- They are also commonly known as **loss functions** or **error functions**. We will use these terms interchangeably.
- They are primarily relevant for Supervised Learning.

As you might guess, machine learning algorithms attempt to minimize cost functions.

For example, when an algorithm "learns" the best coefficients for a linear regression model, it's actually finding the coefficients that minimize a cost function for your dataset.

To make this even clearer, let's take a look at the most commonly used cost function:

7.2.4 Example: sum of squared errors

The standard cost function for regression tasks is called the **sum of squared errors**. Again, there's not a lot of math in this course, but here's where a simple formula makes this much easier to explain:

$$C = \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- y_i is the **actual value** of the target variable for a given observation i
- \hat{y}_i , pronounced "y-hat," is the **predicted value** of the target variable for that same observation i
- $(y_i - \hat{y}_i)^2$ is the squared difference between predicted and actual values (**squared error**)
- $\sum_{i=1}^N (y_i - \hat{y}_i)^2$ is the sum of the squared errors across all observations in the dataset.

Therefore, the more accurate your model is, the lower this cost will be. Linear regression algorithms seek to minimize this function, C , by trying different values for the coefficients.

Overfitting occurs when the cost for a model is **low on the training data, but high for new, unseen data**.

So how does regularization come into play?

7.2.5 Coefficient penalties

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

It does so by adding a **penalty factor** to the cost function.

- This penalty discourages large coefficients.
- This dampens coefficients or removes features entirely (by setting their coefficients to 0), depending on the type of penalty used.
- The "strength" of that penalty factor is **tunable**.

Furthermore, there are two types of penalties.

7.2.6 L_1 regularization

L_1 penalizes the **absolute size** of model coefficients. The new *regularized cost function* becomes:

$$C = \sum_{i=1}^N (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^F |\beta_j|$$

- $|\beta_j|$ is the **absolute size** of a coefficient for a given feature j .
- $\sum_{j=1}^F |\beta_j|$ is the sum of all the absolute coefficient sizes for all features in the model.
- λ , pronounced "lambda," is the **tunable "strength"** of the penalty. Typically, you'll try values between 0 and 10.
- $\sum_{i=1}^N (y_i - \hat{y}_i)^2$ is the same sum of squared errors as in the unregularized case.

7.2.7 L_2 regularization

Likewise, L_2 penalizes the **squared size** of model coefficients. The new *regularized cost function* becomes:

$$C = \sum_{i=1}^N (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^F \beta_j^2$$

- β_j^2 is the **squared size** of a coefficient for a given feature j .
- $\sum_{j=1}^F \beta_j^2$ is the sum of all the squared coefficient sizes for all features.

As you can see, the regularized cost functions are simply the unregularized cost function plus the L_1 or L_2 penalty.

7.3 Regularized Regression

As discussed previously, in the context of linear regression, regularization works by adding penalty terms to the cost function. The algorithms that employ this technique are known as regularized linear regression.

There are 3 types of regularized linear regression algorithms we've used in this course:

1. Lasso
2. Ridge
3. Elastic-Net

They correspond with the amount of L_1 and L_2 penalty included in the cost function.

Oh and in case you're wondering, there's really no definitive "better" type of penalty. It really depends on the dataset and the problem. That's why we recommend trying different algorithms that use a range of penalty mixes (*Algorithms are commodities*).

7.3.1 Lasso

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

- Lasso regression completely relies on the L_1 penalty (**absolute size**).
- Practically, this leads to coefficients that can be exactly 0 (**automatic feature selection**).
- Remember, the "strength" of the penalty should be **tuned**.
- A stronger penalty leads to more coefficients pushed to zero.

```
# Import Lasso Regression
from sklearn.linear_model import Lasso
```

7.3.2 Ridge

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

- Ridge regression completely relies on the L_2 penalty (**squared**).
- Practically, this leads to smaller coefficients, but it doesn't force them to 0 (**feature shrinkage**).
- Remember, the "strength" of the penalty should be **tuned**.
- A stronger penalty leads coefficients pushed closer to zero.

```
# Import Ridge Regression
from sklearn.linear_model import Ridge
```

7.3.3 Elastic-Net

Elastic-Net is a compromise between Lasso and Ridge.

- Elastic-Net combines both L_1 and L_2 penalties.

- The **ratio** of L_1 and L_2 penalties should be tuned.
- The strength of the penalties should also be tuned.
- Ridge and Lasso regression are **special cases** of Elastic-Nets if you set the ratio to completely favor one of the penalty types over the other (setting the ratio to 0 or 1).

```
# Import Elastic Net
from sklearn.linear_model import ElasticNet
```

By the way, you may be wondering why we bothered importing **separate algorithms** for Ridge and Lasso if Elastic-Net combines them.

- It's because Scikit-Learn's implementation of `ElasticNet` can act funky if you set the penalty ratio to 0 or 1.
- Therefore, you should use the separate `Ridge` and `Lasso` classes for those special cases.
- Plus, for some problems, Lasso or Ridge are just as effective while being easier to tune.

7.4 Ensemble Methods

Awesome, we've just seen 3 algorithms that can prevent 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? Well, we need to move away from linear models to do that. We need to bring in a new category of algorithms.

7.4.1 Decision trees (kinda)

Decision trees model data as a "tree" of hierarchical branches. They make branches until they reach "leaves" that represent predictions. They can be used for both regression and classification.

Intuition: To explain the intuition behind decision trees, we'll borrow an answer from a related [Quora discussion](#).

Let's imagine you are playing a game of Twenty Questions. Your opponent has secretly chosen a subject, and you must figure out what she chose. At each turn, you may ask a yes-or-no question, and your opponent must answer truthfully. How do you find out the secret in the fewest number of questions?

It should be obvious some questions are better than others. For example, asking "Is it a basketball" as your first question is likely to be unfruitful, whereas asking "Is it alive" is a bit more useful. Intuitively, we want each question to significantly narrow down the space of possibly secrets, eventually leading to our answer.

That is the basic idea behind decision trees. At each point, we consider a set of questions that can partition our data set. We choose the question that provides the best split (often called maximizing **information gain**), and again find the best questions for the partitions. We stop once all the points we are considering are of the same class (in the naive case). Then classifying is easy. Simply grab a point, and chuck him down the tree. The questions will guide him to his appropriate class.

That explanation was for classification, but their use in regression is analogous.

Mechanics: A decision tree starts with all observations grouped together.

1. First, it splits the observations into two separate **branches** at a chosen feature and a chosen value for that feature.
2. It determines the feature and the value by considering which branches will maximize **information gain** for the target variable.
3. Within each branch, the tree continues to split into more and more branches based on information gain.
4. Finally, each **leaf** (branch with no child branches) is a predicted value.

When predicting new data points, the tree simply passes it down through the hierarchy (going down through the branches based on the splitting criteria) until it reaches a leaf.

Due to their **branching structure**, decision trees have the advantage of being able to model non-linear relationships.

7.4.2 Non-linear relationships

Let's take a hypothetical scenario for the real-estate dataset from Project 2.

For example, let's say that...

- Within single family properties, `'lot_size'` is positively correlated with `'tx_price'`. In other words, larger lots command higher prices for single family homes.
- But within apartments, `'lot_size'` is negatively correlated with `'tx_price'`. In other words, smaller lots command higher prices for apartments (i.e. maybe they are more upscale).

Linear regression models cannot naturally express this "flipping" of correlations.

- Linear regression only learns one coefficient for `'lot_size'`, and it's either positive or negative, not both.
- You must **explicitly** add an interaction variable between `'lot_size'` and `'property_type'`.
- However, it's **unreliable** if you need to explicitly add an interaction variable for each non-linear relationship!

On the other hand, decision trees can **naturally** express this relationship.

- First, they can split on `'property_type'`, creating separate branches for single family properties and apartments.
- Then, within each branch, they can create two separate splits on `'lot_size'`, each representing a different relationship.

This branching mechanism is exactly what leads to higher model complexity.

7.4.3 Unconstrained decision trees

In fact, if you allow them to grow limitlessly, they can completely "memorize" the training data, just from creating more and more branches.

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

We already saw this in our toy example with the noisy sine wave from Project 1.



Figure 7.1: Training set predictions from unconstrained decision tree (Project 1)

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

That's where **tree ensembles** come in!

7.4.4 Bagging vs. boosting

But before talking about tree ensembles, let's first discuss ensemble methods in general.

Ensembles are machine learning methods for combining multiple individual models into a single model.

There are a few different methods for ensembling. In this course, we cover two of them:

1. **Bagging** attempts to *reduce the chance of overfitting complex models*.
 - It trains a large number of "strong" learners in parallel.
 - A **strong learner** is a model that's allowed to have high complexity.
 - It then combines all the strong learners together in order to "smooth out" their predictions.
2. **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 model that has limited complexity.
 - Each one in the sequence focuses on learning from the mistakes of the one before it.
 - It 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.

7.5 Tree Ensembles

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

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

```
# Import Random Forest
from sklearn.ensemble import RandomForestRegressor
```

In practice, random forests tend to perform very well.

- They can be used right out of the box to beat many other models that take up to weeks to develop.
- They are the perfect **"swiss-army-knife"** algorithm that you can almost always rely on to get good results.
- They don't have many complicated parameters to tune.

7.5.2 Boosted trees

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

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

```
# Import Gradient Boosted Trees
from sklearn.ensemble import GradientBoostingRegressor
```

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.

8. Classification Algorithms

Next, we'll dive into a few more key concepts. In particular, we want to introduce you to 4 useful algorithms for classification tasks:

1. L_1 -regularized logistic regression
2. L_2 -regularized logistic regression
3. Random forests
4. Boosted trees

As you might suspect, they are analogous to their regression counterparts, with a few key differences that make them more appropriate for classification.

In applied machine learning, individual algorithms should be swapped in and out depending on which performs best for the problem and the dataset.

Therefore, in this module, we will focus on **intuition** and **practical benefits** over math and theory.

8.1 Binary Classification

Classification with 2 classes is so common that it gets its own name: *binary classification*.

8.1.1 Positive / negative classes

In Project 3, we had two possible classes for an employee's status: 'Left' and 'Employed'.

However, when we constructed our analytical base table, we converted the target variable from 'Left' / 'Employed' into 1 / 0.

```
print( raw_df.status.unique() )  
# [ 'Left' 'Employed' ]  
  
print( abt_df.status.unique() )  
# [1  0]
```

In binary classification,

- 1 ('Left') is also known as the "**positive**" class.
- 0 ('Employed') is also known as the "**negative**" class.

In other words, the positive class is simply the primary class you're trying to identify.

8.1.2 Class probabilities

For regression tasks, the output of a model is pretty straightforward. In Project 2: Real-Estate Tycoon, the output was a prediction for the transaction price of the property.

Therefore, you might logically conclude that the output for a classification task should be a prediction for the status of the employee.

However, in almost every situation, we'd prefer the output to express some level of **confidence** in the prediction, instead of only the predicted class.

Therefore, we actually want the output to be **class probabilities**, instead of just a single class prediction.

- For binary classification, the predicted probability of the positive class and that of the negative class will **sum to 1**.
- For general classification, the predicted probabilities of all the classes will sum to 1.

This will become super clear once we get to the example shortly.

8.2 Noisy Conditional

We're going to use another toy example, just as we did in Project 1.

This time, we're going to build models for a **noisy conditional**.

8.2.1 Methodology

Remember, a model is only useful if it can accurately approximate the "true state of the world" (i.e. the "**true underlying relationship**" between input features and target variables).

Therefore, we're going to create a **synthetic dataset** for which we already know "true underlying relationship."

1. First, we're going to use a single input feature, x .
2. Then, we're going to generate values for the target variable, y , based on a **predetermined mapping function**.
3. Next, we're going to add **randomly generated** noise to that dataset.
4. Once we've done that, we can try different algorithms on our synthetic dataset.
5. We already know the "true underlying relationship"... it's the predetermined mapping function.
6. Finally, we can compare how well models of different complexities can separate the signal from the randomly generated noise.

8.2.2 Synthetic dataset

First, for our predetermined mapping function, we'll use a conditional function that checks if $x > 0.5$. Therefore, the "true underlying relationship" between x and y will be:

If $x > 0.5$ then $y = 1$.

If $x \leq 0.5$ then $y = 0$.

However, we're going to add random noise to it, turning it into a **noisy conditional**:

If $x + \epsilon > 0.5$ then $y = 1$.

If $x + \epsilon \leq 0.5$ then $y = 0$.

```
# Input feature
x = np.linspace(0, 1, 100)

# Noise
np.random.seed(555)
noise = np.random.uniform(-0.2, 0.2, 100)

# Target variable
y = ((x + noise) > 0.5).astype(int)
```

Figure 8.1 shows what that dataset looks like.

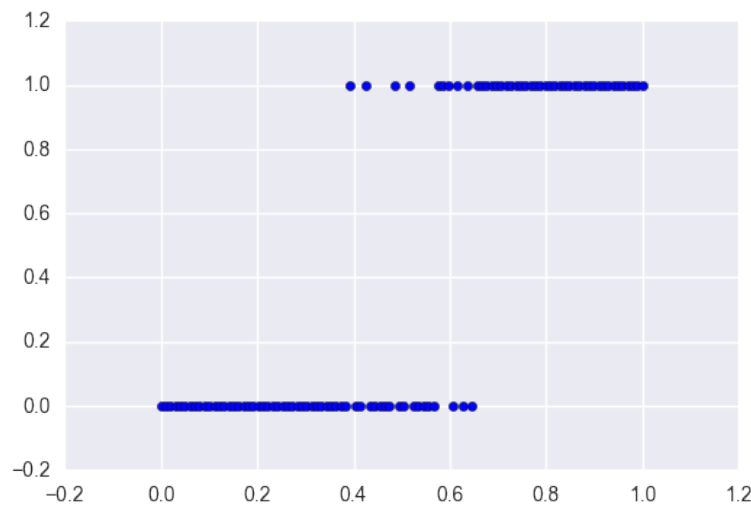


Figure 8.1: Noisy conditional synthetic dataset

As you can see, observations with $x > 0.5$ should have $y = 1$ and those with $x \leq 0.5$ should have $y = 0$. However, as you get closer to $x = 0.5$, the noise plays a bigger factor, and you get some overlap.

Therefore, here's what we generally want out of a good model:

- If $x > 0.5$, we want our model to predict higher probability for $y = 1$.
- If $x \leq 0.5$, we want our model to predict higher probability for $y = 0$.
- If x is closer to 0.5, we want **less confidence** in the prediction.
- If x is farther from 0.5, we want **more confidence** in the prediction.

Finally, there's two more important nuances that we need from a good model:

1. We don't want probability predictions above 1.0 or below 0.0, as those don't make sense.
2. We want **very high confidence** by the time we pass 0.3 or 0.7, because that's when noise is no longer a factor.

8.3 Logistic Regression

First, let's discuss **logistic regression**, which is the classification analog of linear regression.

8.3.1 Linear vs. logistic regression

Remember, linear regression fits a "straight line" with a slope (technically, it's a straight hyperplane in higher dimensions, but the concept is similar).

In Figure 8.2, you can see 2 flaws with linear regression for this type of binary classification problem.

1. First, you're getting predicted probabilities above 1.0 and below 0.0! That just doesn't make sense.
2. Second, remember, we want **very high confidence** by the time we pass 0.3 or 0.7, because that's when noise is no longer a factor. We can certainly do much better on this front.

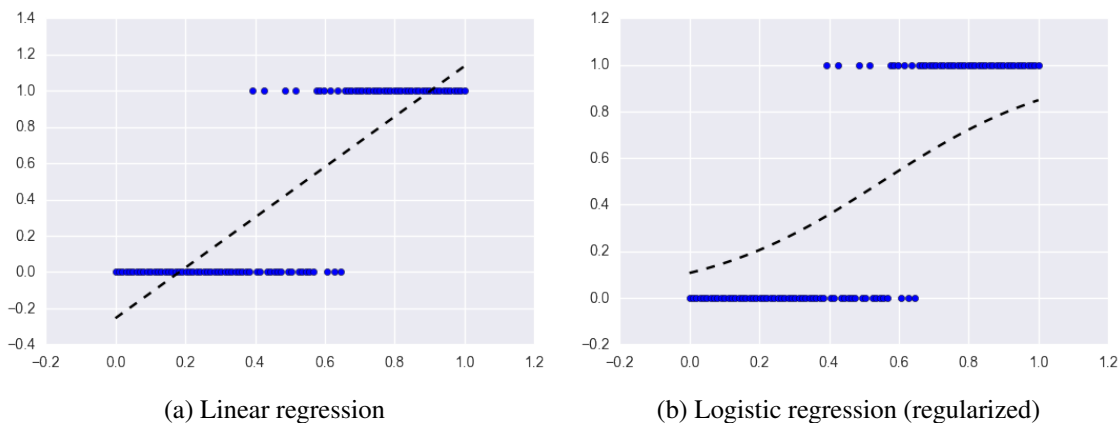


Figure 8.2: Linear vs. Logistic Regression

Also in Figure 8.2, you can see how logistic regression differs.

1. First, we're no longer getting predictions that are above 1.0 or below 0.0, so that's nice.
2. However, we're still not getting **very high confidence** for $x < 0.3$ and $x > 0.7$. It kinda looks like the model is being "too conservative."

We'll see how we can improve on that second point shortly.

8.3.2 Predict vs. predict_proba

First, it's important to note that the `.predict()` function behaves quite differently for classification. It will give you the predicted **classes** directly.

However, as mentioned earlier, we almost always prefer to get the **class probabilities** instead, as they express a measurement of *confidence* in the prediction.

```
# Logistic regression
```



```
model = LogisticRegression()
model.fit(X, y)

# Class predictions
model.predict(X)

# Class probability predictions
pred = model.predict_proba(X)
pred = [p[1] for p in pred]
```

Typically, `.predict_proba()` is the more useful function. It returns the actual class probabilities, which allow you to calculate certain metrics (such as the AUROC) and set custom thresholds (i.e. if false positives are more costly than false negatives).

Here's the full code snippet for fitting and plotting the logistic regression predictions from Figure 8.2.

```
# Logistic regression
model = LogisticRegression()
model.fit(X, y)

# Predict probabilities
pred = model.predict_proba(X)

# Just get the positive class predictions
pred = [p[1] for p in pred]

# Plot dataset and predictions
plt.scatter(X, y)
plt.plot(X, pred, 'k--')
plt.show()
```

8.4 Regularized Logistic Regression

Logistic regression has regularized versions that are analogous to those for linear regression.

Regularized logistic regression also works by adding a **penalty factor** to the cost function.

8.4.1 Penalty strength

As in Lasso, Ridge, and Elastic-Net, the strength of regularized logistic regression's penalty term is tunable.

However, for logistic regression, it's not referred to as `alpha`. Instead, it's called `C`, which is the inverse of the regularization strength.

- Therefore, higher `C` means a weaker penalty and lower `C` means a stronger penalty.
- By default, `C=1`.

In other words, our logistic regression was being regularized by default, which is why it was a bit conservative.

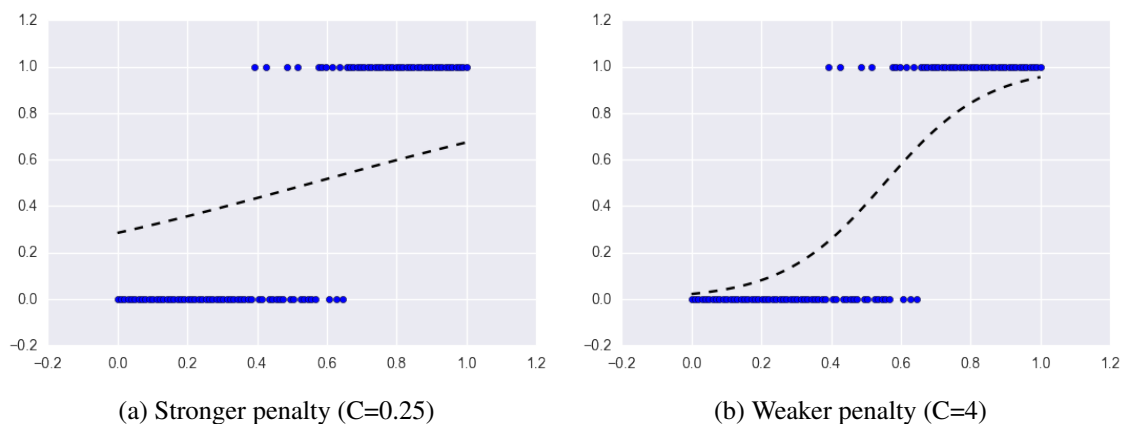


Figure 8.3: Penalty strength

In Figure 8.3, we see what happens when we make the penalty 4 times stronger (`C=0.25`) or 4 times weaker (`C=4`).

For this synthetic dataset, it actually looks like weaker penalties produce better models.

The key is that this is a **hyperparameter** that you should tune using the modeling process from earlier.

8.4.2 Penalty type

By default, the `LogisticRegression()` algorithm uses the L_2 penalty.

- For linear regression, L_2 regularization was called Ridge regression.
- For logistic regression, we'll simply call it **L_2 -regularized logistic regression**.

Likewise...

- For linear regression, L_1 regularization was called Lasso regression.
- For logistic regression, we'll simply call it L_1 -**regularized logistic regression**.

We can set the penalty type when we initialize the algorithm, like so:

```
# L1-regularized logistic regression
l1 = LogisticRegression(penalty='l1', random_state=123)

# L2-regularized logistic regression
l2 = LogisticRegression(penalty='l2', random_state=123)
```

Even though we set the penalty type as a hyperparameter, for this course, we will treat them as separate algorithms (just to be consistent).

8.5 Tree Ensembles

The same tree ensembles we used for regression can be applied to classification.

They work in nearly the same way, except they expect classes for the target variable.

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

```
# Import Random Forest CLASSIFIER
from sklearn.ensemble import RandomForestClassifier
```

In practice, random forests tend to perform very well.

- They can be used right out of the box to beat many other models that take up to weeks to develop.
- They are the perfect **"swiss-army-knife"** algorithm that you can almost always rely on to get good results.
- They don't have many complicated parameters to tune.

8.5.2 Boosted trees

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

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

```
# Import Gradient Boosted Trees CLASSIFIER
from sklearn.ensemble import GradientBoostingClassifier
```

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.

9. Clustering Algorithms

For clustering problems, the chosen input features are usually more important than which algorithm you use.

One reason is that we don't have a clear performance metric for evaluating the models, so it usually doesn't help that much to try many different algorithms.

Furthermore, clustering algorithms typically create clusters based on some similarity score or "**distance**" between observations. It calculates these distances based on the input features.

Therefore, we need to make sure we only feed in relevant features for our clusters.

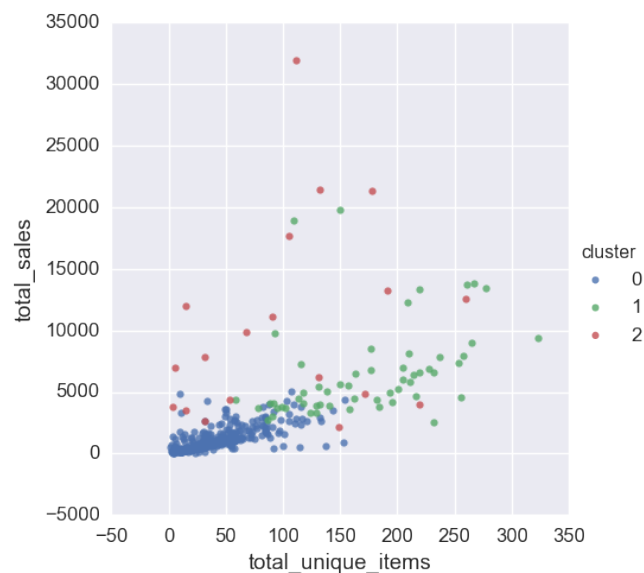


Figure 9.1: Cluster analysis

9.1 K-Means

For clustering, however, it's usually not very fruitful to try multiple algorithms.

- The main reason is that we don't have labeled data... after all, this is Unsupervised Learning.
- In other words, there are no clear **performance metrics** that we can calculate based on a "ground truth" for each observation.
- Typically, the best clustering model is the one that creates clusters that are **intuitive and reasonable** in the eyes of the key stakeholder.

Therefore, which algorithm you choose is typically less important than the input features that you feed into it.

9.1.1 Intuition

K-Means is a very straightforward and popular algorithm for clustering.

It works as follows:

1. You declare a hyperparameter k , the number of different clusters you want.
2. The algorithm initializes k random points in your feature space (called **centroids**).
3. For each observation in your dataset, its **distance** to each centroid is calculated.
4. Each observation is assigned to the cluster of its closest centroid.
5. Then, each centroid is shifted to the **mean** of the observations assigned to that cluster
6. Steps (3) - (5) are repeated until the centroids no longer shift.

In practice, all you need to remember is that this process produces k clusters that "clump together" observations based on **distance**.

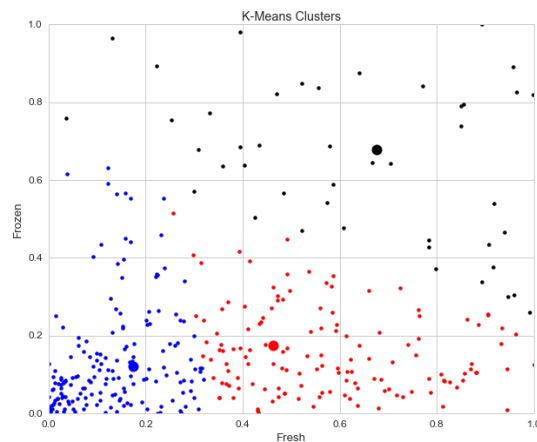


Figure 9.2: K-Means groups observations by distance to centroids.

```
# K-Means algorithm
from sklearn.cluster import KMeans
```

9.1.2 Euclidean distance

So how are those distances calculated? Well, the default way is to calculate the **euclidean distance** based on feature values. This is the ordinary "straight-line" distance between two points.

Imagine you had only two features, x_1 and x_2 .

- Let's say you have an observation with values $x_1 = 5$ and $x_2 = 4$.
- And let's say you had a centroid at $x_1 = 1$ and $x_2 = 1$.

To calculate the euclidean distance between the observation and the centroid, you'd simply take:

$$\sqrt{(5-1)^2 + (4-1)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$$

You might recognize that as simply the Pythagorean theorem from geography.

- It calculates the straight-line distance between two points on the coordinate plane.
- Those points are defined by your feature values.
- This generalizes to any number of features (i.e. *you can calculate straight-line distance between two points in any-dimensional feature space*).

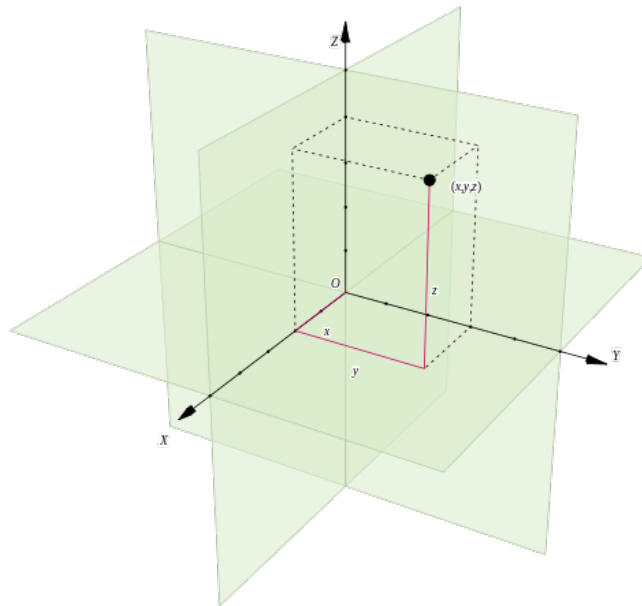


Figure 9.3: Straight-line distance can be calculated between two points in any-dimensional feature space.

There are other forms of distances as well, but they are not as common. They are typically used in special situations. The default euclidean distance is usually a good starting point.

9.1.3 Number of clusters

So, how many clusters should you set?

- As with much of Unsupervised Learning, there's no right or wrong answer.
- Typically, you should consider how your client/key stakeholder will use the clusters.
- You should always feel free to adapt this number depending on what you need.

Finally, keep these 5 tips in mind for clustering:

1. Keep your algorithm simple.
2. Only input relevant features.
3. Standardize your features before clustering.
4. Try different sets of features and compare the clusters created.
5. Communicate proactively with the key stakeholder!

9.2 Feature Sets

Because K-Means creates clusters based on distances, and because distances are calculated by between observations defined by their feature values, the **features you choose to input into the algorithm heavily influence the clusters that are created.**

In Project 4, we looked at 3 possible feature sets and compared the clusters created from them:

1. Only purchase pattern features ("Base DF")
2. Purchase pattern features + item features chosen by thresholding ("Threshold DF")
3. Purchase pattern features + principal component features from items ("PCA DF")

9.2.1 Creating feature sets

The easiest way to prepare your feature sets for clustering is to create separate objects for each feature set you wish to try.

In Project 4, we did so like this:

```
# Import analytical base table
base_df = pd.read_csv('analytical_base_table.csv',
                      index_col=0)

# Import thresholded item features
threshold_item_data = pd.read_csv('threshold_item_data.csv',
                                   index_col=0)

# Import PCA item features
pca_item_data = pd.read_csv('pca_item_data.csv', index_col=0)

# Join base_df with threshold_item_data
threshold_df = base_df.join(threshold_item_data)

# Join base_df with pca_item_data
pca_df = base_df.join(pca_item_data)
```

9.2.2 Finding clusters

Once you have your feature sets ready, fitting the clustering models is very simple (and the process is a small subset of the process for Supervised Learning).

1. First, we make the pipeline with standardization.
2. Then, we fit the pipeline to the feature set (a.k.a. the training data).
3. Next, we call `.predict()` on the same feature set to get the clusters.
4. Finally, we can visualize the clusters.
5. Repeat steps (1) - (4) for each feature set.

Here's how you'd perform steps (1) - (4) for one feature set:

```
# K-Means model pipeline
k_means = make_pipeline(StandardScaler(),
                        KMeans(n_clusters=3, random_state=123))

# Fit K-Means pipeline
k_means.fit(base_df)

# Save clusters to base_df
base_df['cluster'] = k_means.predict(base_df)

# Scatterplot, colored by cluster
sns.lmplot(x='total_sales', y='avg_cart_value',
           hue='cluster', data=base_df, fit_reg=False)
```

The Seaborn library also allows us to visualize the clusters by using the `hue=` argument to its `.lmplot()` function.

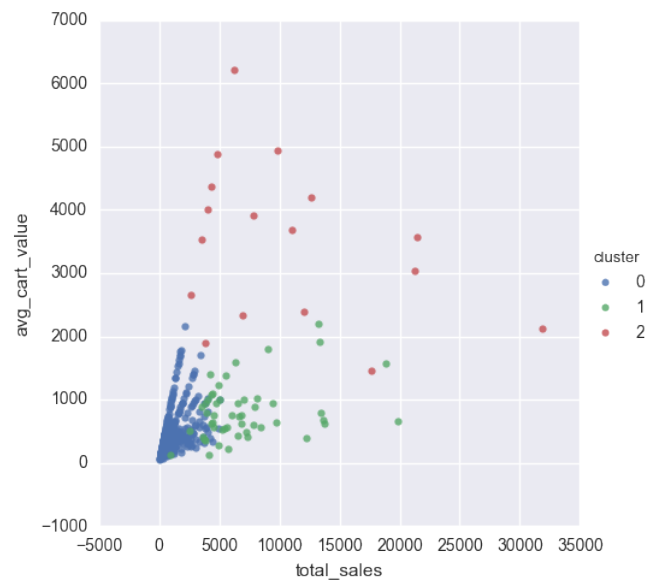


Figure 9.4: Scatterplot, colored by cluster

Again, it's worth reminding that there aren't any clear, widely-accepted **performance metrics** for clustering tasks because we don't have labeled "ground truth."

Therefore, it's usually more useful to just compare the clusters created by different feature sets, and then take them to the key stakeholder for feedback and more iteration.

A. Appendix

A.1 Area Under ROC Curve

Area under ROC curve is the most reliable metric for classification tasks.

- It's equivalent to the probability that a randomly chosen Positive observation ranks higher (has a higher predicted probability) than a randomly chosen Negative observation.
- Basically, it's saying... if you grabbed two observations and exactly one of them was the positive class and one of them was the negative class, what's the likelihood that your model can distinguish the two?
- Therefore, it doesn't care about imbalanced classes.

A.1.1 Confusion matrix

Before presenting the idea of an ROC curve, we must first discuss what a confusion matrix is.

For binary classification, there are 4 possible outcomes for any given prediction.

1. **True positive** - Predict 1 when the actual class is 1.
2. **False positive** - Predict 1 when the actual class is 0.
3. **True negative** - Predict 0 when the actual class is 0.
4. **False negative** - Predict 0 when the actual class is 1.

The confusion matrix displays this information for a set of class predictions.

```
# Predict classes using L1-regularized logistic regression
pred = fitted_models['l1'].predict(X_test)
```

```
# Import confusion_matrix
from sklearn.metrics import confusion_matrix
```

```
# Display confusion matrix for y_test and pred
print( confusion_matrix(y_test, pred) )
# [[1969  172]
#   [ 221  452]]
```

Here's how to interpret that output:

- The **first column** includes observations that were predicted to be 1.
- The **second column** includes observations that were predicted to be 0.
- The **first row** includes observations that were actually 1.
- The **second row** includes observations that were actually 0.

Therefore, among the 2814 observations in the test set (Project 3), 2421 were correctly classified.

- 1969 were **true positives** (first column, first row)
- 452 were **true negatives** (second column, second row)

On the other hand, 393 were incorrectly classified.

- 1221 were **false positives** (first column, second row)
- 172 were **false negatives** (second column, first row)

As you can see, if you combine this information, it goes far beyond simple accuracy. Namely, you separately evaluate the performance for each class. This means the dominant class will not dominate your performance metric.

A.1.2 TPR and FPR

The confusion matrix allows you to calculate two important metrics that are both combinations of elements in the matrix.

True positive rate (TPR), also known as **recall**, is defined as $\frac{TP}{TP+FN}$. In other words, it's the proportion of all positive observations that are correctly predicted to be positive.

False positive rate (FPR) is defined as $\frac{FP}{FP+TN}$. In other words, it's the proportion of all negative observations that are incorrectly predicted to be positive.

Obviously, we want TPR to be higher and FPR to be lower... However, they are intertwined in an important way.

A.1.3 Probability thresholds

Remember, we can predict a probability for each class using `.predict_proba()`, instead of the class directly.

For example:

```
# Predict PROBABILITIES using L1-regularized logistic regression
pred = fitted_models['l1'].predict_proba(X_test)
```

```
# Get just the prediction for the positive class (1)
pred = [p[1] for p in pred]

# Display first 10 predictions
pred[:10]

# [0.030570070257148009,
#  0.004441966482297899,
#  0.007296300193244642,
#  0.088097865803861697,
#  0.071150950128417365,
#  0.48160946301549462,
#  0.12604877174578785,
#  0.6152946894912692,
#  0.72665929094601023,
#  0.13703595544287492]
```

The default behavior of `.predict()` for binary classification is to predict 1 (positive class) if the probability is greater than 0.5. So among those first 10 predictions in the code snippet above, the 8th and 9th would be classified as 1 and the others would be classified as 0.

In other words, 0.5 is the default **threshold**.

However, you can theoretically alter that threshold, depending on your goals. Lowering it will make positive class predictions more likely. Conversely, raising it will make negative class predictions more likely. The threshold you choose is independent of the model.

Here's where TPR and FPR are intertwined!

- If you lower the threshold, the true positive rate increases, but so does the false positive rate.
- Remember, TPR is the proportion of all actually positive observations that were predicted to be positive.
- That means a model that always predicts the positive class will have a true positive rate of 100%.
- However, FPR is the proportion of all actually negative observations that were predicted to be **positive**.
- Therefore, a model that always predicts the positive class will also have a false positive rate of 100%!

And that finally brings us to the ROC curve!

A.1.4 ROC curve

The ROC curve is a way to **visualize the relationship between TPR and FPR** for classification models. It simply plots the true positive rate and false positive rate at different thresholds.

We can create one like so:

```
# Calculate ROC curve from y_test and pred
fpr, tpr, thresholds = roc_curve(y_test, pred)

# Initialize figure
fig = plt.figure(figsize=(8,8))
plt.title('Receiver Operating Characteristic')

# Plot ROC curve
plt.plot(fpr, tpr, label='l1')
plt.legend(loc='lower right')

# Diagonal 45 degree line
plt.plot([0,1],[0,1], 'k--')

# Axes limits and labels
plt.xlim([-0.1,1.1])
plt.ylim([-0.1,1.1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

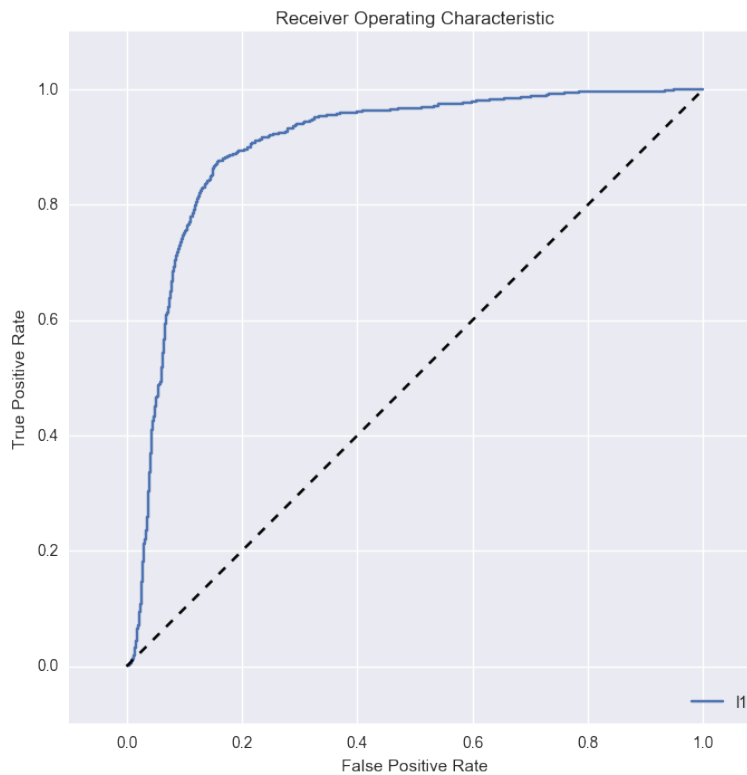


Figure A.1: ROC Curve (Project 3)

The 45 degree **dotted black line** represents the ROC curve of a hypothetical model that makes **completely random predictions**.

Basically, we want our model's curve, the *blue line*, to sit as far above that dotted black line as possible.

	FPR	TPR	Thresholds
532	0.759925	0.994056	0.009627
533	0.784680	0.994056	0.008120
534	0.784680	0.995542	0.008076
535	0.910322	0.995542	0.001962
536	0.910322	0.997028	0.001950
537	0.932742	0.997028	0.001393
538	0.932742	0.998514	0.001388
539	0.949089	0.998514	0.001051
540	0.949089	1.000000	0.001040
541	1.000000	1.000000	0.000027

Figure A.2: As the threshold decreases, both FPR *and* TPR increase.

A.1.5 AUROC

And finally that brings us to **AUROC**, or **area under ROC** curve.

This is literally the area under that *blue line*. The AUROC for a completely random model will be 0.5, and the maximum AUROC possible is 1.0.

Remember, that AUROC is equivalent to the probability that a randomly chosen positive observation ranks higher (has a higher predicted probability) than a randomly chosen negative observation.

```
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, pred)

# Calculate AUROC
print( auc(fpr, tpr) )
# 0.901543001458
```

So the interpretation of that output is that our L_1 -regularized logistic regression (Project 3) has a 90.15% chance of distinguishing between a positive observation and a negative one.

A.2 Data Wrangling

Often, the most interesting machine learning applications require you to wrangle your data first.

In Project 4, you were given a **transaction-level** dataset. In other words, each observation in the raw dataset is for a single transaction - one item, one customer, one purchase.

However, based on the project scope, you needed a **customer-level** dataset for machine learning. Remember, the goal was to produce clusters of customers.

Therefore, before applying machine learning models, you needed to first break down and restructure the dataset.

Specifically, you needed to **aggregate transactions by customer** and **engineer customer-level features**.

- This step blends together exploratory analysis, data cleaning, and feature engineering.
- Here, feature engineering comes from aggregating the transaction-level data.
- You still have a lot of room for creativity in this step.

A.2.1 Customer-Level feature engineering

After cleaning the transaction-level dataset, we **rolled it up** to the customer level and aggregated customer-level features.

We wanted 1 customer per row, and we wanted the features to represent information such as:

- Number of unique purchases by the customer
- Average cart value for the customer
- Total sales for the customer
- Etc.

```
# Aggregate invoice data
invoice_data = df.groupby('CustomerID').InvoiceNo.agg({
    'total_transactions' : 'nunique' })

# Aggregate product data
product_data = df.groupby('CustomerID').StockCode.agg({
    'total_products' : 'count',
    'total_unique_products' : 'nunique' })

# Roll up sales data
sales_data = df.groupby('CustomerID').Sales.agg({
    'total_sales' : 'sum',
    'avg_product_value' : 'mean' })
```

A.2.2 Intermediary levels

You won't always be able to easily roll up to your desired level directly... Sometimes, it will be easier to create intermediary levels first.

```
# Aggregate cart-level data (i.e. invoice-level)
cart = df.groupby(['CustomerID', 'InvoiceNo']).Sales.agg({
    'cart_value' : 'sum' })

# Reset index
cart.reset_index(inplace=True)

# Aggregate cart data (at customer-level)
agg_cart_data = cart.groupby('CustomerID').cart_value.agg({
    'avg_cart_value' : 'mean',
    'min_cart_value' : 'min',
    'max_cart_value' : 'max'})
```

A.2.3 Joining together the ABT

Now you have multiple dataframes that each contain customer-level features.

Next, all you need to do is join them all together.

```
# Join together customer-level ABT
customer_df = invoice_data.join([product_data, sales_data,
                                agg_cart_data])

# Display customer-level data for first 5 customers
customer_df.head()

# Save analytical base table
customer_df.to_csv('analytical_base_table.csv')
```

	total_transactions	total_products	total_unique_products	avg_product_value	total_sales	min_cart_value
CustomerID						
12347	7	182	103	23.681319	4310.00	224.82
12348	4	31	22	57.975484	1797.24	227.44
12349	1	73	73	24.076027	1757.55	1757.55
12350	1	17	17	19.670588	334.40	334.40
12352	8	85	59	29.482824	2506.04	120.33

Figure A.3: Customer-level analytical base table

A.3 Dimensionality Reduction

Next, we'll introduce a concept that's especially important for Unsupervised Learning: *Dimensionality Reduction*.

In Project 4, our client wished to incorporate information about specific item purchases into the clusters.

For example, our model should be more likely to group together customers who buy similar items.

One logical way to represent this is to create dummy variables for each unique gift and feed those features into the clustering algorithm, alongside the purchase pattern features. Now, here's the problem... the retailer sells over 2000 unique gifts. That's over 2000 dummy variables.

This runs into a problem called "The Curse of Dimensionality."

A.3.1 The Curse of Dimensionality

So what is this ominous term called "**The Curse of Dimensionality?**"

In this context, "dimensionality" refers to the number of features in your dataset. The basic idea is that as the number of features increases, you'll need more and more observations to build any sort of meaningful model, especially for clustering.

Because cluster models are based on the "distance" between two observations, and distance is calculated by taking the differences between feature values, every observation will seem "far away" from each other if the number of features increases.

A [Quora](#) user has provided an excellent analogy, 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.

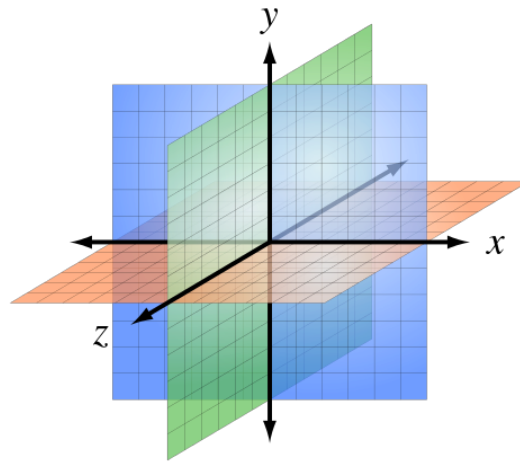


Figure A.4: Imagine searching for a penny as the number of dimensions increases.

For our practical purposes, it's enough to remember that when you have many features (high dimensionality), it makes clustering especially hard because every observation is "far away" from each other.

The amount of "space" that a data point could potentially exist in becomes larger and larger, and clusters become very hard to form.

Remember, in Project 4, this "curse" arose because we tried to create dummy variables for each unique item in the dataset. This led to over 2500 additional features that were created!

Next, we'll review the two methods for reducing the number of features in your dataset.

A.3.2 Method 1: Thresholding

One very **simple and straightforward way** to reduce the dimensionality of this item data is to set a **threshold** for keeping features.

- In Project 4, the rationale is that you might only want to keep **popular items**.
- For example, let's say item A was only purchased by 2 customers. Well, the feature for item A will be 0 for almost all observations, which isn't very helpful.
- On the other hand, let's say item B was purchased by 100 customers. The feature for item B will allow more meaningful comparisons.

For example, this is how we kept item features for only the 20 most popular items.

```
# Get list of StockCodes for the 20 most popular items
top_20_items = item_data.sum().sort_values().tail(20).index

# Keep only features for top 20 items
top_20_item_data = item_data[top_20_items]
```

A.3.3 Method 2: Principal Component Analysis

Principal Component Analysis, or PCA, is a popular dimensionality reduction technique. Technically, PCA is considered its own task under machine learning.

Furthermore, it doesn't require a target variable, so it's actually considered an **Unsupervised Learning** task as well, making it quite useful for this project.

In a nutshell, PCA seeks to create new features by finding linear combinations of your existing ones. These new features, called **principal components**, are meant to maximize the "explained variance."

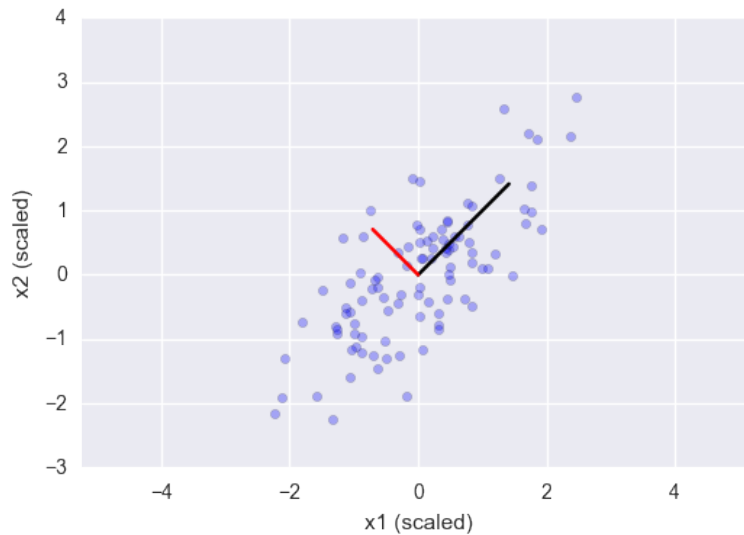


Figure A.5: First principal component (black) and second principal component (red)

Therefore, if you have many correlated features, you can use PCA to capture the key axes of variance and drastically reduce the number of features.

```
# Initialize instance of StandardScaler
scaler = StandardScaler()

# Fit and transform item_data
item_data_scaled = scaler.fit_transform(item_data)

# Initialize PCA transformation, only keeping 125 components
pca = PCA(n_components=125)

# Fit and transform item_data_scaled
PC_items = pca.fit_transform(item_data_scaled)
```

Please refer to the online lesson and your Companion Workbook for a more detailed explanation of PCA, along with a walkthrough using a toy example.