**Classification**

* Overview

**Classification**

[What is Machine Learning](https://www.youtube.com/watch?v=iLu9XyZ55oI)

**Questions that data science methods can answer**

* + **Is this new observation A or B (or C, D, or E) (Classification)**
  + How many or how much of something (Regression)
  + What groupings exist in the data already (Clustering)
  + What should we expect to happen next? (Time Series Analysis)
  + Is this weird? (Anomaly Detection)

**Predicting categorical outcomes**

Classification is a technique for labeling the class of an observation. This is done through the modeling of the patterns in the related data which drive the outcome.

The primary goal of developing a classification model is to generalize patterns. This is done so that the category/class of new data can be identified with a high degree of certainty. Classification can be performed on structured or unstructured data. It can be used to predict binary classes (2 classes) or multi-classes (>2 classes).

In this module, we will explore **supervised** (the data is labeled) machine learning related to **classification** (the target variable is categorical) using **structured** (the data can be naturally stored in rows and columns) data. We will work through the data science pipeline, as well as improving our knowledge of the python programming language.

**Vocabulary**

* + Classifier: An algorithm that maps the input data to a specific category.
  + Classification Model: A series of steps that takes the patterns of input variables, generalizes those patterns, and applies them to new data in order to predict the class.
  + Feature: A feature, aka input/independent variable, is an individual measurable property of a phenomenon being observed.
  + Binary Classification: Classification with two possible outcomes, e.g. pass/fail.
  + Multiclass Classification: Classification with more than two classes, where each sample is assigned to one and only one target label, e.g. Grade levels of students in school (1st-12th).

**Main ideas**

* + Train on data with answers and labels (supervised).
  + This training will produce a decision rule that will be used to classify future data.
  + Using a boundary between points or a distance between points, we classify new data points into A or B (or C or D or E)

**Process**

**Planning**

* + Clearly define your goals, your timeline, and how you plan to get there.
  + Understand what your MVP will be and set milestones.
  + Know who your stakeholders are and understand the final delivery product.
  + Set some initial questions that you plan to investigate.

**Acquire**

* + Acquire structured data from a clipboard, excel, Google sheets, or SQL and read it into pandas.
  + Understand and summarize the data through aggregates, descriptive stats and distribution plots.

**Prepare**

* + Clean the data by converting datatypes and handling missing values.
  + Split our observations into 3 samples - train, validate, test.

**Explore**

* + Univariate analysis: Look at the distribution of your variables individually.
  + Bivariate/multivariate analysis: Look at the relationship of two or more variables.
    - We will discuss the meaning of "drivers", variables vs. features, and the target variable. We will discuss the importance of documenting questions and hypotheses.
    - Visualize the interaction of variables, especially independent variables with the dependent variable using charts such as scatterplots, jointplots, pairgrids, and heatmaps to identify drivers.
    - Run statistical tests to verify relationships. Test hypotheses that involve a categorical variable are t-tests and chi-squared tests.
    - Conclude with documenting answers for those questions, and documenting takeaways and findings at each step of exploration.

**Model**

* + Preprocessing: We will prepare our data specifically for modeling. In this module, we will focus on encoding our values. Machine learning models can not accept string values, therefore, we will turn them all into numbers. Scaling our data is also important for distance-based algorithms. Scaling will be discussed in the regression module.
  + Establish Baseline: We will learn about the importance of establishing a "baseline model" or baseline score and ways to complete this task. The baseline for classification is typically the mode of the dependent variable.
  + Build Models: We will build classification models. What does that mean? We will use well established algorithms to extract the patterns in the data. This will create a model that will allow us to compute predictions for each observation.
  + Model Evaluation: We will compare classification models by computing evaluation metrics. These metrics will measure how well a model did at predicting the target variable, based on different priorities.
  + Model Selection and Testing: We will evaluate the model on the unseen data (the out-of-sample validate and test datasets). We will use validate to tune hyperparameters, and then test on our best model.

**Common Classification Algorithms**

* + Logistic Regression (sklearn.linear\_model.LogisticRegression)
  + Decision Tree (sklearn.tree.DecisionTreeClassifier)
  + Naive Bayes (sklearn.naive\_bayes.BernoulliNB)
  + K-Nearest Neighbors (sklearn.neighbors.KNeighborsClassifier)
  + Random Forest (sklearn.ensemble.RandomForestClassifier)
  + Support Vector Machine (sklearn.svm.SVC)
  + Stochastic Gradient Descent (sklearn.linear\_model.SGDClassifier)
  + AdaBoost (sklearn.ensemble.AdaBoostClassifier)
  + Bagging (sklearn.ensemble.BaggingClassifier)
  + Gradient Boosting (sklearn.ensemble.GradientBoostingClassifier)

See [the sklearn docs on supervised methods](https://scikit-learn.org/stable/supervised_learning.html#supervised-learning) for more.

* Data Acquisition

**Acquisition**

Planning - **Acquisition** - Preparation - Exploratory Analysis - Modeling - Product Delivery

**Goals**

At the end of this lesson, you will know:

* + How to read data from a **csv** using read\_csv. This can be a csv stored locally or a Google Sheet or a file stored in Amazon Web Service (AWS) S3.
  + How to read data from your **local clipboard** using pandas.read\_clipboard. This can be useful for quickly transferring data to/from a spreadsheet.
  + How to read data from **Microsoft Excel** using read\_excel
  + How to read data from a mySQL server using read\_sql(sql\_query, connection\_url). Using this, you can read data using a SQL query to a database. You must have the required drivers installed, and a specially formatted url string must be provided.
  + How, when and why to cache data locally

**Talking to a database**

When connecting to a database, you'll need to have a **database driver** installed that can communicate with the DBMS. For this class we'll use the pymysql driver to talk to our cloud MySQL database.

python -m pip install pymysql

In addition, whenever a connection to a database is needed, we'll need to supply the **database url**. The general form looks like this:

mysql+pymysql://USER:PASSWORD@HOST/DATABASE\_NAME

Where USER, PASSWORD, HOST, and DATABASE\_NAME are placeholders for the actual values that will come from your env.py.

**Local Clipboard**

For a quick transfer of data from a spreadsheet or csv, select the data you wish to acquire and then copy it to your clipboard using command-C. Then acquire that data into a dataframe using the pandas function, read\_clipboard().

Try this out by opening a sheet of data, such as the Google Sheet titled *students* (found in Google Classroom -> Data Section). Select all of the data, copy it to your clipboard, and then in your python environment:

`# import pandas import pandas as pd

**create the dataframe from data on the clipboard**

students\_df = pd.read\_clipboard()

**view your new dataframe**

students\_df.head()`

**Google Sheets**

* + Get the shareable link url: (example) <https://docs.google.com/spreadsheets/d/BLAHBLAHBLAH/edit#gid=NUMBER>
  + Turn that into a CSV export URL:
    - Replace /edit with /export
    - Add format=csv to the beginning of the query string.
    - new url: <https://docs.google.com/spreadsheets/d/BLAHBLAHBLAH/export?format=csv&gid=NUMBER>
  + Pass it to pd.read\_csv, which can take a URL.

`import pandas as pd

sheet\_url = '<https://docs.google.com/spreadsheets/d/1Uhtml8KY19LILuZsrDtlsHHDC9wuDGUSe8LTEwvdI5g/edit#gid=341089357>'

csv\_export\_url = sheet\_url.replace('/edit#gid=', '/export?format=csv&gid=')

df\_googlesheet = pd.read\_csv(csv\_export\_url) df\_googlesheet.head()`

|  | **PassengerId** | **Survived** | **Pclass** | **Name** | **Sex** | **Age** | **SibSp** | **Parch** | **Ticket** | **Fare** | **Cabin** | **Embarked** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| 1 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 2 | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S |
| 3 | 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 | S |
| 4 | 5 | 0 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN | S |

**Hosted or Local .CSV files**

`# File paths can be local.csv files or hosted .csv files url = "<https://gist.githubusercontent.com/ryanorsinger/bec2f59a9cef8ae7428cb70b3541354a/raw/ef64298da52e5d70f4d388f5fd48eccdb02ed3f1/ice_cream.csv>"

df = pd.read\_csv(url) df.head()`

|  | **flavor** | **pints** |
| --- | --- | --- |
| 0 | moolenium crunch | 11.057570 |
| 1 | bubblegum | 6.288724 |
| 2 | chubby hubby | 7.660815 |
| 3 | bubblegum | 6.644338 |
| 4 | neopolitan | 13.600125 |

**SQL**

Create a dataframe from the passengers table in the MySQL database, titanic\_db.

**Database Credentials**

It's a bad idea to store your database access credentials (i.e. your username and password) in plaintext in your source code. There are many different ways one could manage secrets like this, but a simple way is to store the values in a python file that is not included along with the rest of your source code. This is what we have done with the env module.

`import env

def get\_db\_url(db, user=env.user, host=env.host, password=env.password): return f'mysql+pymysql://{user}:{password}@{host}/{db}'

url = get\_db\_url('titanic\_db')

df = pd.read\_sql('SELECT \* FROM passengers', url)

df.head()`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **class** | **deck** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | None | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | C | Cherbourg | 0 |
| 2 | 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | None | Southampton | 1 |
| 3 | 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | C | Southampton | 0 |
| 4 | 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | None | Southampton | 1 |

We will create a function that we can reference later to acquire the data:

`def get\_titanic\_data():

url = get\_db\_url('titanic\_db')

return pd.read\_sql(('SELECT \* FROM passengers'), url)`

We'll store this function in a file named acquire.py.

**Caching Data**

Because data acquisition can take time, it's a common practice to write the data locally to a .csv file and use that csv file when data is required in the future. This process is outlined below:

* + Do whatever you need to do to produce the dataframe that you need.
    - For example df = pd.read\_sql('SELECT...', ...)
    - Or your dataframe cound include joins, multiple data sources, etc...
  + Use df.to\_csv("titanic.csv") to write that dataframe to a file.
  + In your data acquisition function, first check to see if the csv file exists. If it does, read from the csv file, otherwise get the data "fresh".

Here is an example using the titanic dataset:

`import os

def get\_titanic\_data(): filename = "titanic.csv"

if os.path.isfile(filename):

return pd.read\_csv(filename)

else:

# Create connection url

url = get\_db\_url('titanic\_db')

# Read the SQL query into a dataframe

df = pd.read\_sql(('SELECT \* FROM passengers'), url)

# Write that dataframe to disk for later. Called "caching" the data for later.

df.to\_csv(filename)

# Return the dataframe to the calling code

return df`

**Exercises**

The end product of these exercises is a jupyter notebook, acquire.ipynb, **and** an acquire.py file. The notebook will contain all your work as you move through the exercises. The acquire.py file should contain the final functions that acquire the data into a pandas dataframe.

* + Make a new repo called classification-exercises on both GitHub and within your codeup-data-science directory. This will be where you do your work for this module.
  + Inside of your local classification-exercises repo, create a file named .gitignore with the following contents:
  + env.py
  + .DS\_Store
  + .ipynb\_checkpoints/
  + \_\_pycache\_\_
  + \*.csv

Add and commit your .gitignore file before moving forward.

* + Create or copy your env.py file inside of classification-exercises
    - Run git status
    - The env.py file should not be tracked by git
  + Do the following exercises in a jupyter notebook titled acquire.ipynb.
  + Use pydata to import the iris data. Create a pandas dataframe, df\_iris, from this data.
    - print the first 3 rows
    - print the number of rows and columns (shape)
    - print the column names
    - print the data type of each column
    - print the summary statistics for each of the numeric variables
  + Read the data from [this google sheet](https://docs.google.com/spreadsheets/d/1Uhtml8KY19LILuZsrDtlsHHDC9wuDGUSe8LTEwvdI5g/edit?usp=sharing) into a dataframe, df\_google.
    - print the first 3 rows
    - print the number of rows and columns
    - print the column names
    - print the data type of each column
    - print the summary statistics for each of the numeric variables
    - print the unique values for each of your categorical variables
  + Download the google sheet from Exercise 6 into an .xslx (File → Download → Microsoft Excel). Read the downloaded file into a dataframe named df\_excel.
    - assign the first 100 rows to a new dataframe, df\_excel\_sample
    - print the number of rows of your original dataframe
    - print the first 5 column names
    - print the column names that have a data type of object
    - compute the range for each of the numeric variables.

Make a new python module, acquire.py to hold the following data acquisition functions:

* + Make a function named get\_titanic\_data that returns the Titanic data from the codeup data science database as a pandas data frame. Obtain your data from the *Codeup Data Science Database*.
  + Make a function named get\_iris\_data that returns the data from the iris\_db on the codeup data science database as a pandas data frame. The returned data frame should include the actual name of the species in addition to the species\_ids. Obtain your data from the *Codeup Data Science Database*.
  + Make a function named get\_telco\_data that returns the data from the telco\_churn database in SQL. In your SQL, be sure to join contract\_types, internet\_service\_types, payment\_types tables with the customers table, so that the resulting dataframe contains all the contract, payment, and internet service options. Obtain your data from the *Codeup Data Science Database*.
  + Once you've got your get\_titanic\_data, get\_iris\_data, and get\_telco\_data functions written, now it's time to add caching to them. To do this, edit the beginning of the function to check for the local filename of telco.csv, titanic.csv, or iris.csv. If they exist, use the .csv file. If the file doesn't exist, then produce the SQL and pandas necessary to create a dataframe, then write the dataframe to a .csv file with the appropriate name.

**Make sure your env.py and csv files are *not* being pushed to GitHub!**

* Data Preparation

**Preparation**

Planning - Acquisition - **Preparation** - Exploratory Analysis - Modeling - Product Delivery

**Goal:** Prepare, tidy, and clean the data so that it is ready for exploration and analysis.

**Input:** 1 or more dataframes acquired through the "acquire" step.

**Output:** 1 dataset split into 3 samples in the form of dataframes: train, validate & test.

**Artifact:** [prepare.py](http://prepare.py)

**Overview**

* + Summarize our data:
    - head(), describe(), info(), isnull(), value\_counts(), shape, ...
    - plt.hist(), plt.boxplot()
    - document takeaways (nulls, datatypes to change, outliers, ideas for features, etc.)
  + Clean the data:
    - rename columns
    - missing values: drop columns with too many missing values, drop rows with too many missing values, fill with zero where it makes sense, and then make note of any columns you want to impute missing values in (you will need to do that on split data).
    - **outlier**: an observation point that is distant from other observations <https://www.theanalysisfactor.com/outliers-to-drop-or-not-to-drop/>
      * ignore, drop rows, snap to a selected max/min value, create bins (cut, qcut)
    - data errors: drop the rows/observations with the errors, correct them to what it was intended
    - address text normalization issues: correct and standardize tex (e.g. deck 'C' 'c')
    - creating new variables out of existing variables (e.g. z = x - y)
  + Split the data:
    - split our data into train, validate and test sample dataframes
    - why? overfitting: model is not generalizable. It fits the data you've trained it on "too well". 3 points does not necessarily mean a parabola.
    - **train:** *in-sample*, explore, impute mean, scale numeric data (max() - min()...), fit our ml algorithms, evaluate our models.
    - **validate**: *out-of-sample* confirm our top models have not overfit, evaluate our top models on unseen data. Using validate performance results, we pick the top **1** model.
    - **test**: *out-of-sample*, how we expect our top model to perform in production, on unseen data in the future. **ONLY USED ON 1 MODEL.**

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **algorithm vs. model**

* + **algorithm:** the method that sklearn provides, such as decision\_tree, knn, ..., y = mx+b
  + **model:** that algorithm specific to our data, e.g. regression: the model would contain the slope value and intercept value. y = .2x+5 </aside>

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Should I do *this* on the full dataset or on the train sample?**

*this*: the action, method, function, step you are about to take on your data.

* + Are you comparing, looking at the relationship or summary stats or visualizations with 2+ variables?
  + Are you using an sklearn method?
  + Are you moving into the explore stage of the pipeline?

If ONE or more of these is yes, then you should be doing it on your train sample. If **ALL** are no, then the entire dataset is fine.

</aside>

**Summarize the Acquired Data**

`import pandas as pd import numpy as np import matplotlib.pyplot as plt

**import splitting and imputing functions**

from sklearn.model\_selection import train\_test\_split from sklearn.impute import SimpleImputer

**turn off pink boxes for demo**

import warnings warnings.filterwarnings("ignore")

**import our own acquire module**

import acquire`

We'll use the function we defined in the last lesson to acquire our data:

df = acquire.get\_titanic\_data()

`# rows & columns

df.shape`

(891, 13)

`# first n rows

df.head(2)`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **class** | **deck** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | None | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | C | Cherbourg | 0 |

`# Get information about the dataframe: column names, rows, datatypes, non-missing values.

[df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 891 entries, 0 to 890

Data columns (total 13 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 passenger\_id 891 non-null int64

1 survived 891 non-null int64

2 pclass 891 non-null int64

3 sex 891 non-null object

4 age 714 non-null float64

5 sibsp 891 non-null int64

6 parch 891 non-null int64

7 fare 891 non-null float64

8 embarked 889 non-null object

9 class 891 non-null object

10 deck 203 non-null object

11 embark\_town 889 non-null object

12 alone 891 non-null int64

dtypes: float64(2), int64(6), object(5)

memory usage: 90.6+ KB

`# Get summary statistics for numeric columns.

df.describe()`

|  | **passenger\_id** | **survived** | **pclass** | **age** | **sibsp** | **parch** | **fare** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 891.000000 | 891.000000 | 891.000000 | 714.000000 | 891.000000 | 891.000000 | 891.000000 | 891.000000 |
| mean | 445.000000 | 0.383838 | 2.308642 | 29.699118 | 0.523008 | 0.381594 | 32.204208 | 0.602694 |
| std | 257.353842 | 0.486592 | 0.836071 | 14.526497 | 1.102743 | 0.806057 | 49.693429 | 0.489615 |
| min | 0.000000 | 0.000000 | 1.000000 | 0.420000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 222.500000 | 0.000000 | 2.000000 | 20.125000 | 0.000000 | 0.000000 | 7.910400 | 0.000000 |
| 50% | 445.000000 | 0.000000 | 3.000000 | 28.000000 | 0.000000 | 0.000000 | 14.454200 | 1.000000 |
| 75% | 667.500000 | 1.000000 | 3.000000 | 38.000000 | 1.000000 | 0.000000 | 31.000000 | 1.000000 |
| max | 890.000000 | 1.000000 | 3.000000 | 80.000000 | 8.000000 | 6.000000 | 512.329200 | 1.000000 |

`# Check out distributions of numeric columns.

num\_cols = df.columns[[df[col].dtype == 'int64' for col in df.columns]] for col in num\_cols: plt.hist(df[col]) plt.title(col) plt.show()`

`# Use .describe with object columns.

obj\_cols = df.columns[[df[col].dtype == 'O' for col in df.columns]] for col in obj\_cols: print(df[col].value\_counts()) print(df[col].value\_counts(normalize=True, dropna=False)) print('----------------------')`

male 577

female 314

Name: sex, dtype: int64

male 0.647587

female 0.352413

Name: sex, dtype: float64

----------------------

S 644

C 168

Q 77

Name: embarked, dtype: int64

S 0.722783

C 0.188552

Q 0.086420

None 0.002245

Name: embarked, dtype: float64

----------------------

Third 491

First 216

Second 184

Name: class, dtype: int64

Third 0.551066

First 0.242424

Second 0.206510

Name: class, dtype: float64

----------------------

C 59

B 47

D 33

E 32

A 15

F 13

G 4

Name: deck, dtype: int64

None 0.772166

C 0.066218

B 0.052750

D 0.037037

E 0.035915

A 0.016835

F 0.014590

G 0.004489

Name: deck, dtype: float64

----------------------

Southampton 644

Cherbourg 168

Queenstown 77

Name: embark\_town, dtype: int64

Southampton 0.722783

Cherbourg 0.188552

Queenstown 0.086420

None 0.002245

Name: embark\_town, dtype: float64

----------------------

`# Create bins for fare using .value\_counts.

**Using sort = false will sort by bin values as opposed to the frequency counts.**

df.fare.value\_counts(bins=5, sort=False)`

(-0.513, 102.466] 838

(102.466, 204.932] 33

(204.932, 307.398] 17

(307.398, 409.863] 0

(409.863, 512.329] 3

Name: fare, dtype: int64

`# Find columns with missing values and the total of missing values.

missing = df.isnull().sum() missing[missing > 0]`

age 177

embarked 2

deck 688

embark\_town 2

dtype: int64

**Takeaways**

* + embarked == embark\_town, so remove embarked & keep embark\_town
  + class == pclass, so remove class & keep pclass (already numeric)
  + drop deck...way too many missing values
  + fill embark\_town with most common value ('Southampton')
  + drop age column
  + encode or create dummy vars for sex & embark\_town.

**Clean the Data**

`# Drop duplicates...run just in case; reassign and check the shape of my data.

df = df.drop\_duplicates() df.shape`

(891, 13)

`# Drop columns with too many missing values for now and reassign; check the shape of my data.

cols\_to\_drop = ['deck', 'embarked', 'class', 'age'] df = df.drop(columns=cols\_to\_drop) df.shape`

(891, 9)

`# Validate that the columns are dropped.

df.head(1)`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **sibsp** | **parch** | **fare** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 1 | 0 | 7.25 | Southampton | 0 |

We could fill embark\_town with the most common value, 'Southampton', by hard-coding the value using the fillna() function, as below. Or we could use an imputer. We will demonstrate the imputer *after* the train-validate-test split.

`# Run .fillna() on the entire df.

df['embark\_town'] = df.embark\_town.fillna(value='Southampton')`

`# Validate that missing values in embark\_town have been handled.

df.embark\_town.isna().sum()`

0

Create a function that will automate these steps for when we need to reproduce our process.

def clean\_data(df): ''' This function will drop any duplicate observations, drop ['deck', 'embarked', 'class', 'age'], fill missing embark\_town with 'Southampton' and create dummy vars from sex and embark\_town. ''' df = df.drop\_duplicates() df = df.drop(columns=['deck', 'embarked', 'class', 'age']) df['embark\_town'] = df.embark\_town.fillna(value='Southampton') return df

Acquire my data again to test my function below.

df = acquire.get\_titanic\_data() df.head(2)

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **class** | **deck** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | None | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | C | Cherbourg | 0 |

df = clean\_data(df) df.head(2)

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **sibsp** | **parch** | **fare** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 1 | 0 | 7.2500 | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 1 | 0 | 71.2833 | Cherbourg | 0 |

df.info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 891 entries, 0 to 890

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 passenger\_id 891 non-null int64

1 survived 891 non-null int64

2 pclass 891 non-null int64

3 sex 891 non-null object

4 sibsp 891 non-null int64

5 parch 891 non-null int64

6 fare 891 non-null float64

7 embark\_town 891 non-null object

8 alone 891 non-null int64

dtypes: float64(1), int64(6), object(2)

memory usage: 69.6+ KB

**Split the Data**

`# 20% test, 80% train\_validate

**then of the 80% train\_validate: 30% validate, 70% train.**

train, test = train\_test\_split(df, test\_size=.2, random\_state=123, stratify=df.survived) train, validate = train\_test\_split(train, test\_size=.3, random\_state=123, stratify=train.survived)`

`# Validate my split.

print(f'train -> {train.shape}') print(f'validate -> {validate.shape}') print(f'test -> {test.shape}')`

train -> (498, 9)

validate -> (214, 9)

test -> (179, 9)

Create a function that will split the Titanic data into train, validate, and test DataFrames when we provide a DataFrame. It will also stratify by survived, the categorical target variable.

def split\_data(df): ''' take in a DataFrame and return train, validate, and test DataFrames; stratify on survived. return train, validate, test DataFrames. ''' train\_validate, test = train\_test\_split(df, test\_size=.2, random\_state=123, stratify=df.survived) train, validate = train\_test\_split(train\_validate, test\_size=.3, random\_state=123, stratify=train\_validate.survived) return train, validate, test

Test out my function and validate my split.

train, validate, test = split\_data(df)

`# Validate my split

print(f'train -> {train.shape}') print(f'validate -> {validate.shape}') print(f'test -> {test.shape}')`

train -> (498, 9)

validate -> (214, 9)

test -> (179, 9)

**Impute Missing Values**

We can impute values using the mean, median, mode (most frequent), or a constant value. We will use sklearn.imputer.SimpleImputer to do this.

* + Create the imputer object, selecting the strategy used to impute (mean, median or mode (strategy = 'most\_frequent').
  + Fit to train. This means compute the mean, median, or most\_frequent (i.e. mode) for each of the columns that will be imputed. Store that value in the imputer object.
  + Transform train: fill missing values in train dataset with that value identified
  + Transform test: fill missing values with that value identified

`# Get fresh Titanic data to use with missing values in embark\_town again.

df = acquire.get\_titanic\_data() train, validate, test = split\_data(df)`

`# ONLY look at the train dataset after we split our data.

[train.info](http://train.info)()`

<class 'pandas.core.frame.DataFrame'>

Int64Index: 498 entries, 583 to 744

Data columns (total 13 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 passenger\_id 498 non-null int64

1 survived 498 non-null int64

2 pclass 498 non-null int64

3 sex 498 non-null object

4 age 401 non-null float64

5 sibsp 498 non-null int64

6 parch 498 non-null int64

7 fare 498 non-null float64

8 embarked 497 non-null object

9 class 498 non-null object

10 deck 106 non-null object

11 embark\_town 497 non-null object

12 alone 498 non-null int64

dtypes: float64(2), int64(6), object(5)

memory usage: 54.5+ KB

* + Create the SimpleImputer object, which we will store in the variable imputer. In the creation of the object, we will specify the strategy to use (mean, median, most\_frequent). Essentially, this is creating the instructions and assigning them to a variable, imputer.

imputer = SimpleImputer(missing\_values = None, strategy='most\_frequent')

* + Fit the imputer to the columns in the training df. This means that the imputer will determine the most\_frequent value, or other value depending on the strategy called, for each column.

imputer = imputer.fit(train[['embark\_town']])

* + It will store that value in the imputer object to use upon calling transform. We will call transform on our train, validate, and test datasets to fill any missing values.

`train[['embark\_town']] = imputer.transform(train[['embark\_town']])

validate[['embark\_town']] = imputer.transform(validate[['embark\_town']])

test[['embark\_town']] = imputer.transform(test[['embark\_town']])`

`# Validate that there are no longer any Null values in embark\_town.

train.embark\_town.value\_counts(dropna=False)`

Southampton 355

Cherbourg 102

Queenstown 41

Name: embark\_town, dtype: int64

Create a function that will run through all of the steps above when I pass the train, validate, and test DataFrames as arguments. **My clean\_data() function is already taking care of the missing values in embark\_town, using .fillna() on the entire DataFrame. If I want to use this impute\_mode() function, I could go back and tweak my clean\_data() function, either/or but I don't need both.**

def impute\_mode(train, validate, test): ''' take in train, validate, and test DataFrames, impute mode for embark\_town, and return train, validate, and test DataFrames ''' imputer = SimpleImputer(missing\_values = None, strategy='most\_frequent') train[['embark\_town']] = imputer.fit\_transform(train[['embark\_town']]) validate[['embark\_town']] = imputer.transform(validate[['embark\_town']]) test[['embark\_town']] = imputer.transform(test[['embark\_town']]) return train, validate, test

Use our helper functions clean\_data() and split\_data() in our single prep\_titanic\_data() function that takes in the Titanic DataFrame and returns cleaned and split datasets with the mode value imputed for missing values in embark\_town.

`# Acquire fresh Titanic data to test our function.

df = acquire.get\_titanic\_data()`

def prep\_titanic\_data(df): ''' This function takes in a df and will drop any duplicate observations, drop ['deck', 'embarked', 'class', 'age'], fill missing embark\_town with 'Southampton' create dummy vars from sex and embark\_town, and perform a train, validate, test split. Returns train, validate, and test DataFrames ''' df = clean\_data(df) train, validate, test = split\_data(df) return train, validate, test

`# Run final prepare function and verify that the function is working properly.

train, validate, test = prep\_titanic\_data(df) [train.info](http://train.info)()`

<class 'pandas.core.frame.DataFrame'>

Int64Index: 498 entries, 583 to 744

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 passenger\_id 498 non-null int64

1 survived 498 non-null int64

2 pclass 498 non-null int64

3 sex 498 non-null object

4 sibsp 498 non-null int64

5 parch 498 non-null int64

6 fare 498 non-null float64

7 embark\_town 498 non-null object

8 alone 498 non-null int64

dtypes: float64(1), int64(6), object(2)

memory usage: 38.9+ KB

**Exercises**

The end product of this exercise should be the specified functions in a python script named prepare.py. Do these in your classification\_exercises.ipynb first, then transfer to the [prepare.py](http://prepare.py) file.

This work should all be saved in your local classification-exercises repo. Then add, commit, and push your changes.

Using the Iris Data:

* + Use the function defined in acquire.py to load the iris data.
  + Clean up the column names - replace the period with an underscore and lowercase.
  + Drop the species\_id and measurement\_id columns.
  + Rename the species\_name column to just species.
  + Create a function named prep\_iris that accepts the untransformed iris data, and returns the data with the transformations above applied.

Using the Titanic dataset

* + Use the function defined in acquire.py to load the Titanic data.
  + Drop any unnecessary, unhelpful, or duplicated columns.
  + Create a function named prep\_titanic that accepts the raw titanic data, and returns the data with the transformations above applied.

Using the Telco dataset

* + Use the function defined in acquire.py to load the Telco data.
  + Drop any unnecessary, unhelpful, or duplicated columns. This could mean dropping foreign key columns but keeping the corresponding string values, for example.
  + Handle null values.
  + Create a function named prep\_telco that accepts the raw telco data, and returns the data with the transformations above applied.

Split your data

* + Write a function to split your data into train, test and validate datasets. Add this function to [prepare.py](http://prepare.py).
  + Run the function in your notebook on the Iris dataset, returning 3 datasets, train\_iris, validate\_iris and test\_iris.
  + Run the function on the Titanic dataset, returning 3 datasets, train\_titanic, validate\_titanic and test\_titanic.
  + Run the function on the Telco dataset, returning 3 datasets, train\_telco, validate\_telco and test\_telco.
* Exploratory Analysis

**Exploratory Data Analysis**

Planning - Acquisition - Preparation - **Exploratory Analysis** - Modeling - Product Delivery

In this lesson, we introduce exploratory data analysis, called EDA, which is the "explore" stage in the DS Pipeline.

**What is EDA?**

Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypotheses and to check assumptions with the help of summary statistics and graphical representations.

[Prasad Patil](https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15)

In short, we learn nearly all of our takeaways and insights from the data in the exploration stage. Modeling is certainly useful, but exploration is where we human beings actually learn the story contained in our data. The more we explore our data and learn the domain and the context of the data, the better the outcome we produce with our efforts. Exploring (and getting to know the domain of the data) can lead to feature engineering, feature elimination to reduce noise, and domain based outlier handling. Thhis means our exploration efforts can directly lead to better models.

**Why EDA?**

We explore the interactions of the attributes and target variable to help discover drivers of our target variable and redundant or interdependent attributes.

* + Discover features that are driving the outcome (target).
  + Learn the vast majority of our takeaways and interesting stories from the data.
  + Discover if we need to drop features, if we need to handle missing values, or if there's value to combining features.

**EDA Process**

We start by thinking about our data and asking questions.

If there's a plan or even a chance you will build models with this data, be sure to split the dataset and **only explore train**. By exploring only train data, we keep our out-of-sample data unseen. If you are absolutely not going to model, it's OK to explore the entire dataset. Any exploration more in-depth than a histogram or .value\_counts should only be done on train.

* + Hypothesize: Form and document your initial hypotheses about how the predictors (independent variables, features, or attributes) interact with the target (y-value or dependent variable). You can do this in the form of questions in a natural language (as opposed to "statistical" language).
  + Visualize: use visualization techniques to identify drivers. When a visualization needs to be followed up with a statistical test, do so.
  + Test your hypotheses when visualization isn't immediately clear. Use the appropriate statistical tests (t-tests, correlation, chi-square)

**General Recipe**

* + Bivariate Stats. Bivariate means two variables.
    - Plot the interactions of each variable with the target. Document your takeaways.
      * For numeric to numeric, use a scatterplot or lineplot
      * For numeric to categorical variables, see <https://seaborn.pydata.org/tutorial/categorical.html>
    - Explore interaction of independent variables using viz and/or hypothesis testing to address interdependence.
  + Multivariate Stats (more than 2 variables): Ask additional questions of the data, such as how subgroups compare to each-other and to the overall population. Answer questions using visualizations and/or hypothesis testing.
    - If you're using seaborn's [relplot](https://seaborn.pydata.org/generated/seaborn.relplot.html) or [catplot](https://seaborn.pydata.org/generated/seaborn.catplot.html), use the hue or col arguments to add extra dimension(s) to the visuals.
    - Using [sns.pairplot](https://seaborn.pydata.org/generated/seaborn.pairplot.html) with hue may be helpful. With too many columns, however, it can produce visuals that are too noisy to be useful.
    - See <https://seaborn.pydata.org/tutorial/axis_grids.html> for more multivariate options
  + Statistical Tests: If the visualizations are not crystal clear, it's important to conduct hypothesis tests.
    - With numeric to numeric, test for correlation with Pearson's R for linear and Spearman's R for non-linear relationships.
    - For numeric to categorical, compare the means of two populations or a subgroup to the population using a [t-test](https://ds.codeup.com/stats/compare-means/), if your samples are normally(ish) distributed but have different variances (as determined by calling .var() on each column/Series, [ANOVA](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html) to compare means from more than 2 groups, or a [Mann-Whitney u-test](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html) if the data does not match the [assumptions of a t-test](https://www.investopedia.com/ask/answers/073115/what-assumptions-are-made-when-conducting-ttest.asp).
    - With categorical to categorical variables, use $\chi^2$, [chi-squared test](https://ds.codeup.com/stats/compare-group-membership/).

**Standing Orders** for Exploration

* + **Document** your initial questions or assumptions. Write them down (in your README or notebook) so they are concrete and not in your head.
  + **Document** your takeaways after each visualization. Even if your takeaway is, "there is nothing interesting between *var1* and *target*".
  + **Document** your answer to each question.
  + When you run statistical tests to answer your questions, **Document** your null and alternative hypothesis, the test you run, the test results, and your conclusion.
  + **Document** your takeaways, in case that wasn't clear. It is a huge component of your final deliverable/analysis.
  + **Document** your action plan. What are your next steps and/or new questions based on what you have learned? I recommend documenting, continuing through all of your questions, and then going back and taking action only after you have answered your initial questions.

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from sklearn.model\_selection import train\_test\_split

from acquire import get\_titanic\_data

np.random.seed(123)

**Acquire**

Acquire Titanic data from our MySQL database

df = get\_titanic\_data()

df.head(2)

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **class** | **deck** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | None | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | C | Cherbourg | 0 |

df.isna().sum()

passenger\_id 0

survived 0

pclass 0

sex 0

age 177

sibsp 0

parch 0

fare 0

embarked 2

class 0

deck 688

embark\_town 2

alone 0

dtype: int64

# Useful helper for checking for nulls

# What proportion of each column is empty?

df.isna().mean()

passenger\_id 0.000000

survived 0.000000

pclass 0.000000

sex 0.000000

age 0.198653

sibsp 0.000000

parch 0.000000

fare 0.000000

embarked 0.002245

class 0.000000

deck 0.772166

embark\_town 0.002245

alone 0.000000

dtype: float64

**Prepare**

Prepare the Titanic data. We apply the same steps before splitting so that we handle edge cases identically.

* + drop deck since most of the data is missing
  + drop rows where age or embarked is missing
  + drop passenger\_id, since it adds no new information
  + drop class, as encoded values are in pclass
  + create dummy vars & drop sex, embark\_town

**Questions for 2nd draft**

* + Let's investigate and determine what are best options are for handling the missing ages

# drop rows where age or embarked is null, drop column 'deck', drop passenger\_id

def prep\_titanic(df):

'''

take in titanc dataframe, remove all rows where age or embarked is null,

get dummy variables for sex and embark\_town,

and drop sex, deck, passenger\_id, class, and embark\_town.

'''

df = df[(df.age.notna()) & (df.embarked.notna())]

df = df.drop(columns=['deck', 'passenger\_id', 'class'])

return df

df = prep\_titanic(df)

df.head(2)

|  | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Southampton | 0 |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | Cherbourg | 0 |

**Split** data into Train, Validate, Test

def train\_validate\_test\_split(df, target, seed=123):

'''

This function takes in a dataframe, the name of the target variable

(for stratification purposes), and an integer for a setting a seed

and splits the data into train, validate and test.

Test is 20% of the original dataset, validate is .30\*.80= 24% of the

original dataset, and train is .70\*.80= 56% of the original dataset.

The function returns, in this order, train, validate and test dataframes.

'''

train\_validate, test = train\_test\_split(df, test\_size=0.2,

random\_state=seed,

stratify=df[target])

train, validate = train\_test\_split(train\_validate, test\_size=0.3,

random\_state=seed,

stratify=train\_validate[target])

return train, validate, test

# Stratify with categorical target variables

train, validate, test = train\_validate\_test\_split(df, target='survived')

train.shape, validate.shape, test.shape

((398, 10), (171, 10), (143, 10))

# Stratification means we'll get even proportions of the target variable in each data set

train.survived.mean(), validate.survived.mean(), test.survived.mean()

(0.4045226130653266, 0.40350877192982454, 0.40559440559440557)

**Interactions of 2 Variables**

For bivariate exploration, we should analyze each feature with respect to the target variable and document takeaways.

A scatterplot will help us visualize numeric to numeric columns.

A catplot, barplot, or boxplot will help us visualize categorical variables with numeric variables. A nice thing about a .barplot is that by setting the binary target to the y axis, we see proportions.

Always write down what questions you have before you explore.

Always document your findings and takeaways, even if the takeaway is "There's nothing here between x and y"

Some findings from bi-variate exploration may need more detailed investigation with an additional variable.

train.head(2)

|  | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 450 | 0 | 2 | male | 36.0 | 1 | 2 | 27.75 | S | Southampton | 0 |
| 543 | 1 | 2 | male | 32.0 | 1 | 0 | 26.00 | S | Southampton | 0 |

**Questions to answer**

* + pclass to survived. What's the relationship between survival and class of ticket?
  + alone to survived. Is traveling with people more helpful for surviving?
  + gender to survived. What's the relationship between survival and gender?
  + fare to survived. What's the relationship between fare and survival?

What further combinations of features could you analyze? What about binning age into young, middle, and older? Are there any features that would make sense to create?

**1. What's the relationship between survival and class of ticket?**

plt.title("It pays to travel first and second class")

sns.barplot(x="pclass", y="survived", data=train)

population\_survival\_rate = train.survived.mean()

plt.axhline(population\_survival\_rate, label="Population survival rate")

plt.legend()

plt.show()

The above visual is quite clear, but if it was less clear or we want more certainty, we'll run a hypothesis test.

# Let's run a chi squared to compare proportions, to have more confidence

alpha = 0.05

null\_hypothesis = "survival and class of ticket are independent"

alternative\_hypothesis = "there is a relationship between class of ticket and survival"

# Setup a crosstab of observed survival to pclass

observed = pd.crosstab(train.survived, train.pclass)

chi2, p, degf, expected = stats.chi2\_contingency(observed)

if p < alpha:

print("Reject the null hypothesis that", null\_hypothesis)

print("Sufficient evidence to move forward understanding that", alternative\_hypothesis)

else:

print("Fail to reject the null")

print("Insufficient evidence to reject the null")

p

Reject the null hypothesis that survival and class of ticket are independent

Sufficient evidence to move forward understanding that there is a relationship between class of ticket and survival

6.956716773744683e-15

**2. Is traveling with people more helpful for surviving?**

plt.title("Better not travel alone.")

sns.barplot(x="alone", y="survived", data=train)

population\_survival\_rate = train.survived.mean()

plt.axhline(population\_survival\_rate, label="Population survival rate")

plt.legend()

plt.show()

# If the visual was unclear or we need more confirmation, run a hypothesis test

# Let's run a chi squared to compare proportions

alpha = 0.05

null\_hypothesis = "survival and traveling alone are independent"

alternative\_hypothesis = "there is a relationship between survival and traveling alone"

# Setup a crosstab of observed survival to pclass

observed = pd.crosstab(train.survived, train.alone)

chi2, p, degf, expected = stats.chi2\_contingency(observed)

if p < alpha:

print("Reject the null hypothesis that", null\_hypothesis)

print("Sufficient evidence to move forward understanding that", alternative\_hypothesis)

else:

print("Fail to reject the null")

print("Insufficient evidence to reject the null")

Reject the null hypothesis that survival and traveling alone are independent

Sufficient evidence to move forward understanding that there is a relationship between survival and traveling alone

**3. What's the relationship between survival and gender?**

plt.title("Women more likely to survive")

sns.barplot(x="sex", y="survived", data=train)

population\_survival\_rate = train.survived.mean()

plt.axhline(population\_survival\_rate, label="Population survival rate")

plt.legend()

plt.show()

# Let's run a chi squared to compare proportions, to have more confidence

alpha = 0.05

null\_hypothesis = "survival and gender are independent"

alternative\_hypothesis = "there is a relationship between survival and gender"

# Setup a crosstab of observed survival to pclass

observed = pd.crosstab(train.survived, train.sex)

chi2, p, degf, expected = stats.chi2\_contingency(observed)

if p < alpha:

print("Reject the null hypothesis that", null\_hypothesis)

print("Sufficient evidence to move forward understanding that", alternative\_hypothesis)

else:

print("Fail to reject the null")

print("Insufficient evidence to reject the null")

Reject the null hypothesis that survival and gender are independent

Sufficient evidence to move forward understanding that there is a relationship between survival and gender

**4. What's the relationship between fare and survival?**

Comparing Numeric vs. a Categorical Example

If the visual is not completely clear, the need for a hypothesis test increases.

* + First, set your confidence level and your alpha (Confidence level of 95% and alpha of 0.05 are good)
  + Select the appropriate test for your 2 variables:
    - If numeric to numeric, run a correlation test
    - Compare proportions with $\chi^2$
    - Compare means with t-test or Mann-Whitney u-test.
  + Run tests, form conclusions, and document your results

(rinse and then repeat with other variables of interest)

1 tailed Mann-Whitney test. For more on Mann-Whitney, see the [Wikipedia article](https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test)

Write down your hypothesis setup

* + $H\_0$: Survived passenger fares cost less than or equal to non-surviving passenger fare
  + $H\_a$: Surviving passenger fares cost more than non-surviving passenger fares

# Subset the data into survived and not-surviving status

not\_survived = train[train.survived == 0]

survived = train[train.survived == 1]

# check variances of the two groups

not\_survived.fare.var(), survived.fare.var()

(846.4473286218973, 7299.474755819128)

plt.title("What's the relationship of fare and survival?")

plt.hist(survived.fare, label="survived")

plt.hist(not\_survived.fare, label="perished")

plt.legend()

plt.show()

from scipy import stats

null\_hypothesis = "fares paid by survived passengers are less than or equal to fares paid by non-surviving passengers"

alternative\_hypothesis = "fares paid by surviving passengers are higher than non-surviving passengers"

# Use Mann Whitney u-test when t-test assumptions do not hold.

# The distributions are not normal and the variances are not equal, so we'll do mann-whitney

t, p = stats.mannwhitneyu(survived.fare, not\_survived.fare, alternative="greater")

if p < alpha:

print("We find evidence to reject the null hypothesis that", null\_hypothesis)

print("We find evidence evidence to move forward with the understaind that", alternative\_hypothesis)

else:

print("We fail to reject the null hypothesis")

print("There is no evidence that", null\_hypothesis)

We find evidence to reject the null hypothesis that fares paid by survived passengers are less than or equal to fares paid by non-surviving passengers

We find evidence evidence to move forward with the understaind that fares paid by surviving passengers are higher than non-surviving passengers

p

4.665118844596931e-14

**Takeaways**

* + It pays to travel first and second class
  + Better to travel with family/company
  + Women much more likely to survive
  + Survivors paid higher fares, which matches with class of ticket

**Important Questions**

* + What else can you determine by looking at sibsp, parch, fare, or age compared to our target survived?
  + If the visuals are not *crystal clear* in indicating a relationship, then we should conduct hypothesis testing with the appropriate variables along with the target.
  + For a second draft:
    - Any other variables we can combine together?
    - Would it make sense to create a variable like is\_child which would be a boolean?
    - Would it make sense to investigate each independent variable with each-other and not only the target?
  + What else do you notice?
  + And what else can we learn if we look at the interaction of more than two variables at a time?

**Interaction of 3+ Variables**

During multivariate analysis, we often add another dimension to our data, such as the target variable as color.

We also ask additional, more specific and targeted questions of the data, such as how subgroups compare to each-other and to the overall population. We then answer these questions using visualizations and/or hypothesis testing.

Some options for multivariate visuals:

* + If you have multiple numeric columns, generating a [pairplot](https://seaborn.pydata.org/generated/seaborn.pairplot.html) with the target variable set to the hue argument might help. It may also be too noisy.
  + A [relplot](https://seaborn.pydata.org/generated/seaborn.relplot.html) of a numeric x, a numeric y, and a hue or col argument using a category z. If you discover a good set of numeric columns in the pairplot, then it would be valuable to create a visual for that pairing along with the target.
  + We can also use hue or col along with [seaborn catplot](https://seaborn.pydata.org/generated/seaborn.catplot.html)
  + We can make subgroups based on multiple categorical features and compare to other groups or the population

Multivariate statistical tests exist, but are outside the scope of this course.

We can, however, create subgroups based on multiple categorical features and conduct hypothesis tests.

**In each class of ticket, survivors trend younger**

# At each class ticket, women have better survival rates

plt.title("In each class of ticket, survivors trend younger")

sns.boxplot(x="pclass", y="age", data=train, hue="survived")

population\_survival\_rate = train.survived.mean()

plt.axhline(population\_survival\_rate)

plt.show()

**Note** A scatterplot of 2 numeric columns by one or two categorical columns might be very useful for numeric targets on your y axis

# No discernable relationship between age and fare, but the pclass relates to survival

rel = sns.relplot(x='age', y='fare', data=train, col="pclass", hue="survived")

rel.fig.suptitle('Age to Fare, colum by pclass') # One way to title a relplot

plt.tight\_layout()

**Women more likely to survive at all ticket classes**

plt.title("Women more likely to survive at all ticket classes")

sns.barplot(x="pclass", y="survived", hue="sex", data=train)

population\_survival\_rate = train.survived.mean()

plt.show()

women\_in\_3rd\_class = train[(train.pclass == 3) & (train.sex == 'female')]

men\_in\_3rd\_class = train[(train.pclass == 3) & (train.sex == 'male')]

print(f"Overall survival rate:", round(train.survived.mean(), 2))

print(f"Overall female survival rate", round(train[train.sex == 'female'].survived.mean(), 2))

print(f"Overall male survival rate", round(train[train.sex == 'male'].survived.mean(), 2))

print(f"3rd class ticket survival rate", round(train[train.pclass == 3].survived.mean(), 2))

print(f"Survival rate for women in 3rd class:", round(women\_in\_3rd\_class.survived.mean(), 2))

print(f"Survival rate for men in 3rd class:", round(men\_in\_3rd\_class.survived.mean(), 2))

Overall survival rate: 0.4

Overall female survival rate 0.75

Overall male survival rate 0.2

3rd class ticket survival rate 0.21

Survival rate for women in 3rd class: 0.46

Survival rate for men in 3rd class: 0.1

**Traveling alone, at any class ticket is riskier**

cat = sns.catplot(data=train, x="pclass", y="survived", hue="alone", kind="bar")

cat.fig.suptitle("Traveling alone, at any class ticket is riskier")

plt.tight\_layout()

**Takeaways** 1. Women more likely to survive at all ticket classes 2. In each class of ticket, survivors trend younger 3. First class alone has better survival than 3rd class traveling with family 4. Fare and age don't seem to correlate, but fare and class of ticket appear related. 5. For both men and women passengers, survival and pclass are dependent

*What do you learn from these? What are some actions you wish to take?*

# Gender subset setup

male\_subset = train[train.sex == 'male']

female\_subset = train[train.sex == 'female']

null\_hypothesis = "survival and pclass are independent for female passengers"

alternative\_hypothesis = "survival and pclass are dependent for female passengers"

alpha = 0.05

observed = pd.crosstab(female\_subset.survived, female\_subset.pclass)

chi2, p, degf, expected = stats.chi2\_contingency(observed)

if p < alpha:

print("We reject the null hypothesis that", null\_hypothesis)

print("Evidence suggests that", alternative\_hypothesis)

else:

print("We fail to reject the null hypothesis")

print("We find insufficient evidence to support the claim that", alternative\_hypothesis)

We reject the null hypothesis that survival and pclass are independent for female passengers

Evidence suggests that survival and pclass are dependent for female passengers

null\_hypothesis = "survival and pclass are independent for male passengers"

alternative\_hypothesis = "survival and pclass are dependent for male passengers"

alpha = 0.05

observed = pd.crosstab(male\_subset.survived, male\_subset.pclass)

chi2, p, degf, expected = stats.chi2\_contingency(observed)

if p < alpha:

print("We reject the null hypothesis that", null\_hypothesis)

print("Evidence suggests that", alternative\_hypothesis)

else:

print("We fail to reject the null hypothesis")

print("We find insufficient evidence to support the claim that", alternative\_hypothesis)

We reject the null hypothesis that survival and pclass are independent for male passengers

Evidence suggests that survival and pclass are dependent for male passengers

**Get Creative**

Ask additional, more specific and targeted questions of the data, such as how subgroups compare to each-other and to the overall population. We then answer these questions using visualizations and/or hypothesis testing.

* + Is there a relationship between survival and parch for women travelers?
  + Is there a relationship between survival and parch for male travelers?
  + Is there a relationship between survival and sibsp for women travelers?
  + Is there a relationship between survival and sibsp for male travelers?

What other subgroups can you create and visualize?

What variables have you not worked with yet?

What other subgroups can you create and test?

**Conclusion**

Here we pull all of our takeaways and actions together into one place we can reference as we move forward.

* + Overall, 60% did NOT survive. This will be your baseline prediction to test the usefulness of features when you get to modeling.
  + Younger people appear more likely to survive. A new feature of is\_child might be helpful, or maybe binning age into 3 groups of "young", "middle", and "older" would be more helpful than age as a numeric column, alone.
  + Take all of your takeaways and document them in one place.
  + If you have time, maybe on a second iteration, you can dive deep into more variable pairings.

**Work fast to an MVP understanding of your data**

* + Focus on features that give us the biggest bang for your buck.
  + If there's 30% of a population that's responsible for 99% of the sales, starting with breaking down that 30% population into different groups

**When we have time for a second iteration**

* + Revisit some of the things you may have skipped earlier in order to get to an MVP.
  + For example, there were ~20% of rows with missing age. If you have time, check these out. Is there a set of the population that is similar such that we can impute an expected age value?
  + Explore creating your own features
    - Turning numeric columns like age into a category with is\_child, for example with a boolean.
    - Where does it make logical sense to combine columns?

**Exercises**

**Part I**

Do these exercises in explore.ipynb notebook. As always, add, commit, and push your changes.

**Section 1 - iris\_db:** Using iris data from our MySQL server and the methods used in the lesson above:

* + Acquire, prepare & split your data.
  + Univariate Stats
    - For each measurement type (quantitative variable): create a histogram, boxplot, & compute descriptive statistics (using .describe()).
    - For each species (categorical variable): create a frequency table and a bar plot of those frequencies.
    - Document takeaways & any actions.
  + Bivariate Stats
    - Visualize each measurement type (y-axis) with the species variable (x-axis) using barplots, adding a horizontal line showing the overall mean of the metric (y-axis).
    - For each measurement type, compute the descriptive statistics for each species.
    - For virginica & versicolor: Compare the mean petal\_width using the Mann-Whitney test (scipy.stats.mannwhitneyu) to see if there is a significant difference between the two groups. Do the same for the other measurement types.
    - Document takeaways & any actions.
  + Multivariate Stats
    - Visualize the interaction of each measurement type with the others using a pairplot (or scatter matrix or something similar) and add color to represent species.
    - Visualize two numeric variables of the species. Hint: sns.relplot with hue or col
    - Create a swarmplot using a melted dataframe of all your numeric variables. The x-axis should be the variable name, the y-axis the measure. Add another dimension using color to represent species. Document takeaways from this visualization.
    - Ask a specific question of the data, such as: is the sepal area signficantly different in virginica compared to setosa? Answer the question through both a plot and using a mann-whitney or t-test. If you use a t-test, be sure assumptions are met (independence, normality, equal variance).
    - Document takeaways and any actions.

**Part II**

Explore your titanic dataset more completely.

* + Determine drivers of the target variable
  + Determine if certain columns should be dropped
  + Determine if it would be valuable to bin some numeric columns
  + Determine if it would be valuable to combine multiple columns into one.

Does it make sense to combine any features?

Do you find any surprises?

Document any and all findings and takeaways in your notebook using markdown.

**Part III**

* + Explore your telco data to discover drivers of churn
  + Determine if certain columns should be dropped
  + Determine if it would be valuable to bin some numeric columns
  + Determine if it would be valuable to combine multiple columns into one.

What are your drivers of churn?

Does it make sense to combine any features?

Do you find any surprises?

Document any and all findings and takeaways in your notebook using markdown.

* Evaluation

**Classification Model Evaluation**

In this lesson, we will discuss common ways of evaluating a classification model's performance.

**Note** To simplify things, this lesson uses a single dataframe and does not perform data splitting. The methods used in this section along with the classification\_report function, which you'll see in the next lesson, are used to evaluate model performance on train to see in-sample performance, on validate to see out-of-sample performance and allow us to tune hyper-parameters, and ultimately on the test dataset.

**The Confusion Matrix**

A **confusion matrix** is a cross-tabulation of our model's predictions against the actual values. We will discuss the different evaluation metrics in this lesson.

As a simple example, imagine we are predicting whether or not someone likes coffee. Our data and predictions might look like this:

import pandas as pd

from sklearn.metrics import confusion\_matrix

df = pd.DataFrame({

'actual': ['coffee', 'no coffee', 'no coffee', 'coffee', 'coffee', 'coffee', 'no coffee', 'coffee'],

'prediction': ['no coffee', 'no coffee', 'coffee', 'coffee', 'coffee', 'coffee', 'no coffee', 'no coffee'],

})

df

|  | **actual** | **prediction** |
| --- | --- | --- |
| 0 | coffee | no coffee |
| 1 | no coffee | no coffee |
| 2 | no coffee | coffee |
| 3 | coffee | coffee |
| 4 | coffee | coffee |
| 5 | coffee | coffee |
| 6 | no coffee | no coffee |
| 7 | coffee | no coffee |

The crosstab matrix of 'actual' and 'prediction' columns would look like this

pd.crosstab(df.actual, df.prediction)

| **prediction** | **coffee** | **no coffee** |
| --- | --- | --- |
| actual |  |  |
| coffee | 3 | 2 |
| no coffee | 1 | 2 |

The sklearn confusion matrix would look like this:

confusion\_matrix(df.actual, df.prediction,

labels = ('no coffee', 'coffee'))

array([[2, 1],

[2, 3]])

The matrix here represent the 4 possible outcomes of our classification task:

* + c[0,0]: There are 2 **True Negatives**, where we predicted the people don't like cofee and they really don't
  + c[0:1]: There is 1 **False Positive**, where we predicted the person likes coffee but they really don't
  + c[1,0]: There are 2 **False Negatives**, where we predicted those people don't like coffee, but they really do
  + c[1,1]: There are 3 **True Positives**, that is for 4 people they really do like coffee and we predicted they do

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Positives and Negatives**

Here we are treating liking coffee as the positive case and not liking coffee as the negative case. This choice is arbitrary and we could have chosen not liking coffee as the positive case and liking cofee as the negative case.

Either way, when discussing classification model performance, you'll see one outcome classified as positive and the other as negative.

</aside>

**Baseline**

For a classification problem, a common choice for the baseline model is a model that simply predicts the most common class every single time.

df.actual.value\_counts()

coffee 5

no coffee 3

Name: actual, dtype: int64

In our example, there are 5 coffee drinkers and 3 non-coffee drinkers, so our baseline model would be to predict that someone likes coffee every single time.

df['baseline\_prediction'] = 'coffee'

**Common Evaluation Metrics**

Now that we have introduced the idea of a confusion matrix, we can discuss some metrics that are derived from it.

**Accuracy**

Accuracy is the number of times we predicted correctly divided by the total number of observations. Put another way:

$$ \frac{TP + TN}{TP + TN + FP + FN} $$

In our example above, this would be

$$ \frac{3+2}{3+2+1+2}=\frac{5}{8}=0.625 $$

So our model's overall accuracy is 62.5%.

Accuracy is a good, easy to understand metric, but can fail to capture the whole picture when the classes in the original dataset are not evenly distributed.

**Precision**

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Positives and Negatives**

While the overall accuracy will remain the same no matter which outcome we designate as the positive and the negative, because of their definition, precision and recall **are** affected by these choices.

</aside>

Precision is the percentage of positive predictions that we made that are correct. Precision tells us how good our model's positive predictions are, and does not take into account false negatives or true negatives. More formally:

$$ \frac{TP}{TP + FP} $$

In our example:

$$ \frac{3}{3+1}=0.75 $$

That is, 75% of the time that we predicted someone likes coffee, we were right.

We might choose to optimize for precision when the cost of acting on a positive prediction is high. With precision as a metric, false negatives are "free", but false positives are costly. For example we might optimize for precision when predicting whether or not an email message is spam, as it is better to send a spam message to a user's inbox than it is to send a real message to the spam folder.

**Recall**

Recall is the percentage of positive cases that we accurately predicted. Recall tells us how well our model does at "capturing" the actually positve cases. Recall does not take into account false positives or True negatives.

$$ \frac{TP}{TP+FN} $$

In our example:

$$ \frac{3}{3+2}=0.6 $$

We predicted 60% of the people that like coffee correctly.

We might choose to optimize for recall when the cost of missing out on a positive case is high, or when it is better to act on a predicted positive than not to. With recall as a metric, false positives are "free", but false negatives are costly. For example, we might optimize for recall when trying to flag fradulent bank transactions, as it is better to flag is non-fraudulent transaction for review than it is to miss out on an actually fraudulent transaction.

**Other Metrics**

While the metrics above are some of the most common, they are not by far an exhaustive list. Here is an overview of several other common metrics:

* + **Misclassification Rate**: 1 - accuracy; how often does the model get it wrong?
  + **Sensitivity**: aka *True Positive Rate*; how good is our model when the actual value is positive? recall for the positive class
  + **Specificity**: How good is our model when the actual value is negative? Recall for the negative class
  + **False positive rate**: How likely is it we get a false positive when the actual value is negative?
  + **F1 Score**: the harmonic mean of precision and recall
  + **Area Under ROC Curve**: A way to measure overall model performance for models that predict not just a class, but a probability as well.

**Evaluation**

Now we will put the metrics we've discussed into practice with python code.

First we can calculate accuracy. Accuracy is simply the number of times where we got the prediction right:

model\_accuracy = (df.prediction == df.actual).mean()

baseline\_accuracy = (df.baseline\_prediction == df.actual).mean()

print(f' model accuracy: {model\_accuracy:.2%}')

print(f'baseline accuracy: {baseline\_accuracy:.2%}')

model accuracy: 62.50%

baseline accuracy: 62.50%

Recall is how well we do on actually positive cases. Here we'll define positive as preferring coffe.

First we'll subset the dataframe so that we are only looking at the rows where we have the positive case. Then we'll evaluate how well our model's predictions do.

subset = df[df.actual == 'coffee']

model\_recall = (subset.prediction == subset.actual).mean()

baseline\_recall = (subset.baseline\_prediction == subset.actual).mean()

print(f' model recall: {model\_recall:.2%}')

print(f'baseline recall: {baseline\_recall:.2%}')

model recall: 60.00%

baseline recall: 100.00%

Notice here that our baseline model has 100% recall. This is because the baseline is to always predict the person prefers coffee, so we'll never miss an actually positive case.

Next we'll calculate precision. Precision is based on just the times that the model predicts the positive class. Because the predictions for our model and the baseline differ, we'll need to create 2 seperate subsets here.

subset = df[df.prediction == 'coffee']

model\_precision = (subset.prediction == subset.actual).mean()

subset = df[df.baseline\_prediction == 'coffee']

baseline\_precision = (subset.baseline\_prediction == subset.actual).mean()

print(f'model precision: {model\_precision:.2%}')

print(f'baseline precision: {baseline\_precision:.2%}')

model precision: 75.00%

baseline precision: 62.50%

Notice that the baseline model's precision is the same as it's accuracy. This is because the baseline model always predicts the positive case, so the subset of the data used for the precision calculation is the entire dataset.

**Multi-Class Classification**

All of the above metrics can be applied to a multi-class classfication problems as well. Overall, we treat the multiclass classification performance evaluation as a sequence of binary classification performance evaluations, one for each class. This approach is sometimes referred to as **one-vs-rest**.

The steps for doing so are:

* + Look at one class individually. Treat correctly identifying the class as the positive case.
  + Compute performance metrics for this class.
  + Repeat for every other classes.
  + Average the performance metrics together.

The average calculation can be performed by simply averaging all the metrics together and dividing by the number of data points (a **macro average**), or by a weighted average. For the weighted average, we weight each metric by the number of data points in the class and divide by the total number of data points. This process is referred to as a **micro average**.

One way to think about this is that the macro average weighs each class equally, while the micro average weighs each observation equally.

**Further Reading**

* + [Wikipedia: Confusion Matrix](https://en.wikipedia.org/wiki/Confusion_matrix)
  + [Wikipedia: Receiver Operating Characteristic](https://en.wikipedia.org/wiki/Receiver_operating_characteristic)
  + [A helpful notebook on classification model evaluation](https://www.ritchieng.com/machine-learning-evaluate-classification-model/)

**Exercises**

* + Create a new file named model\_evaluation.py or model\_evaluation.ipynb for these exercises.
  + Given the following confusion matrix, evaluate (by hand) the model's performance.
  + | | pred dog | pred cat |
  + |:------------ |-----------:|-----------:|
  + | actual dog | 46 | 7 |
  + | actual cat | 13 | 34 |
    - In the context of this problem, what is a false positive?
    - In the context of this problem, what is a false negative?
    - How would you describe this model?
  + You are working as a datascientist working for Codeup Cody Creator (C3 for short), a rubber-duck manufacturing plant.

Unfortunately, some of the rubber ducks that are produced will have defects. Your team has built several models that try to predict those defects, and the data from their predictions [can be found here](https://ds.codeup.com/data/c3.csv).

Use the predictions dataset and pandas to help answer the following questions:

* + - An internal team wants to investigate the cause of the manufacturing defects. They tell you that they want to identify as many of the ducks that have a defect as possible. Which evaluation metric would be appropriate here? Which model would be the best fit for this use case?
    - Recently several stories in the local news have come out highlighting customers who received a rubber duck with a defect, and portraying C3 in a bad light. The PR team has decided to launch a program that gives customers with a defective duck a vacation to Hawaii. They need you to predict which ducks will have defects, but tell you the really don't want to accidentally give out a vacation package when the duck really doesn't have a defect. Which evaluation metric would be appropriate here? Which model would be the best fit for this use case?
  + You are working as a data scientist for Gives You Paws ™, a subscription based service that shows you cute pictures of dogs or cats (or both for an additional fee).

At Gives You Paws, anyone can upload pictures of their cats or dogs. The photos are then put through a two step process. First an automated algorithm tags pictures as either a cat or a dog (Phase I). Next, the photos that have been initially identified are put through another round of review, possibly with some human oversight, before being presented to the users (Phase II).

Several models have already been developed with the data, and [you can find their results here](https://ds.codeup.com/data/gives_you_paws.csv).

Given this dataset, use pandas to create a baseline model (i.e. a model that just predicts the most common class) and answer the following questions:

* + - In terms of accuracy, how do the various models compare to the baseline model? Are any of the models better than the baseline?
    - Suppose you are working on a team that solely deals with dog pictures. Which of these models would you recommend?
    - Suppose you are working on a team that solely deals with cat pictures. Which of these models would you recommend?
  + Follow the links below to read the documentation about each function, then apply those functions to the data from the previous problem.
    - [sklearn.metrics.accuracy\_score](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html)
    - [sklearn.metrics.precision\_score](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html)
    - [sklearn.metrics.recall\_score](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html)
    - [sklearn.metrics.classification\_report](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)
* Preprocessing

**Preprocessing**

This stage is focused on ensuring our data is ready to be fed into a model.

* + Columns: The train, validate, and test dataset should have the same columns. If any modifications were made to train, they should be executed on validate and test, as well.
  + Datatypes: All datatypes that will be fed into model need to be numeric (dummy vars, factor vars, manual encoding)
  + Scale numeric data: If an algorithm will be used that will be affected by the differing weights, this ensures that continuous variables have the same weight and are on the same units.
    - This will be talked about in a future module
  + Tidy data: Getting your data in the shape it needs to be for modeling and exploring. Every row should be an observation and every column should be a feature/attribute/variable. We want 1 observation per row, and 1 row per observation. If you want to predict a customer churn, each row should be a customer and each customer should be on only 1 row. (address duplicates, aggregate, melt, reshape, ...)
    - this will be talked about in a future module

We will be focused on cleaning up datatypes for modeling. This is known as encoding, where we turn all string values into numeric values.

We will continue working through the titanic dataset.

`import pandas as pd import numpy as np

from sklearn.model\_selection import train\_test\_split

import acquire import prepare`

`# Get Titanic data

df = acquire.get\_titanic\_data() df.head()`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **age** | **sibsp** | **parch** | **fare** | **embarked** | **class** | **deck** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | None | Southampton | 0 |
| 1 | 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | C | Cherbourg | 0 |
| 2 | 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | None | Southampton | 1 |
| 3 | 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | C | Southampton | 0 |
| 4 | 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | None | Southampton | 1 |

`# Prepare Titanic data

train, validate, test = prepare.prep\_titanic\_data(df) train.head()`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **sibsp** | **parch** | **fare** | **embark\_town** | **alone** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 583 | 583 | 0 | 1 | male | 0 | 0 | 40.1250 | Cherbourg | 1 |
| 165 | 165 | 1 | 3 | male | 0 | 2 | 20.5250 | Southampton | 0 |
| 50 | 50 | 0 | 3 | male | 4 | 1 | 39.6875 | Southampton | 0 |
| 259 | 259 | 1 | 2 | female | 0 | 1 | 26.0000 | Southampton | 0 |
| 306 | 306 | 1 | 1 | female | 0 | 0 | 110.8833 | Cherbourg | 1 |

Get dummy vars for sex and embark\_town

* + dummy\_na: create a dummy var for na values, also?
  + drop\_first: drop first dummy var (since we know if they do not belong to any of the vars listed, then they must belong to the first one that is not listed).

`# Using drop\_first leaves sex\_male, embark\_town\_Queenstown, and embark\_town\_Southampton.

dummy\_train = pd.get\_dummies(train[['sex','embark\_town']], dummy\_na=False, drop\_first=[True, True]) dummy\_train.head()`

|  | **sex\_male** | **embark\_town\_Queenstown** | **embark\_town\_Southampton** |
| --- | --- | --- | --- |
| 583 | 1 | 0 | 0 |
| 165 | 1 | 0 | 1 |
| 50 | 1 | 0 | 1 |
| 259 | 0 | 0 | 1 |
| 306 | 0 | 0 | 0 |

`# Concatenate the dummy\_df dataframe above with the original df and verify.

train = pd.concat([train, dummy\_train], axis=1) train.head()`

|  | **passenger\_id** | **survived** | **pclass** | **sex** | **sibsp** | **parch** | **fare** | **embark\_town** | **alone** | **sex\_male** | **embark\_town\_Queenstown** | **embark\_town\_Southampton** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 583 | 583 | 0 | 1 | male | 0 | 0 | 40.1250 | Cherbourg | 1 | 1 | 0 | 0 |
| 165 | 165 | 1 | 3 | male | 0 | 2 | 20.5250 | Southampton | 0 | 1 | 0 | 1 |
| 50 | 50 | 0 | 3 | male | 4 | 1 | 39.6875 | Southampton | 0 | 1 | 0 | 1 |
| 259 | 259 | 1 | 2 | female | 0 | 1 | 26.0000 | Southampton | 0 | 0 | 0 | 1 |
| 306 | 306 | 1 | 1 | female | 0 | 0 | 110.8833 | Cherbourg | 1 | 0 | 0 | 0 |

`# Drop string values that have been replaced with encoded values.

train = train.drop(columns=['sex', 'embark\_town']) train.head()`

|  | **passenger\_id** | **survived** | **pclass** | **sibsp** | **parch** | **fare** | **alone** | **sex\_male** | **embark\_town\_Queenstown** | **embark\_town\_Southampton** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 583 | 583 | 0 | 1 | 0 | 0 | 40.1250 | 1 | 1 | 0 | 0 |
| 165 | 165 | 1 | 3 | 0 | 2 | 20.5250 | 0 | 1 | 0 | 1 |
| 50 | 50 | 0 | 3 | 4 | 1 | 39.6875 | 0 | 1 | 0 | 1 |
| 259 | 259 | 1 | 2 | 0 | 1 | 26.0000 | 0 | 0 | 0 | 1 |
| 306 | 306 | 1 | 1 | 0 | 0 | 110.8833 | 1 | 0 | 0 | 0 |

This dataframe is now ready for modeling.

We need to follow the same steps for the validation and test dataframes, so they will also work with the algorithms.

`# Using drop\_first leaves sex\_male, embark\_town\_Queenstown, and embark\_town\_Southampton.

dummy\_val = pd.get\_dummies(validate[['sex','embark\_town']], dummy\_na=False, drop\_first=[True, True]) dummy\_test = pd.get\_dummies(test[['sex','embark\_town']], dummy\_na=False, drop\_first=[True, True])

**Concatenate the dummy\_df dataframe above with the original df.**

validate = pd.concat([validate, dummy\_val], axis=1) test = pd.concat([test, dummy\_test], axis=1)

**Drop string values that have been replaced with encoded values.**

validate = validate.drop(columns=['sex', 'embark\_town']) test = test.drop(columns=['sex', 'embark\_town'])`

**Exercises**

Do these exercises in a notebook called modeling.ipynb first, then transfer the final functions to the model.py file.

This work should all be saved in your local classification-exercises repo. Add, commit, and push your changes.

Using the Titanic dataset

* + Use the function defined in acquire.py to load the Titanic data.
  + Use the function defined in prepare.py to prepare the titanic data.
  + Encode the categorical columns on train dataset. Create dummy variables of the categorical columns and concatenate them onto the dataframe. Remove the columns they are replacing. Repeat on validate and test.
  + Create a function named preprocess\_titanic that accepts the train, validate, and test titanic data, and returns the dataframes ready for modeling.

Using the Telco dataset

* + Use the function defined in acquire.py to load the Telco data.
  + Use the function defined in prepare.py to prepare the Telco data.
  + Encode the categorical columns on train.
    - Encode at least one column using .replace
    - Encode at least one column using .map
    - Encode the rest of the columns by creating dummy variables and concatenating them onto the dataframe.
  + Repeat the same steps on validate and test.
  + Create a function named prep\_telco that accepts the train, validate, and test telco data, and returns the dataframes ready for modeling.
* Modeling
  + Overview

**Modeling Overview**

Planning - Acquisition - Preparation - Exploratory Analysis - **Modeling** - Product Delivery

**Decision Tree (CART: Classification And Regression Trees)**

[sklearn.tree.DecisionTreeClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier)

* + - A sequence of rules used to classify 2 or more classes.
    - Each node represents a single input variable (x) and a split point or class of that variable.
    - The leaf nodes of the tree contain an output variable (y) which is used to make a prediction.
    - Predictions are made by walking the splits of the tree until arriving at a leaf node and output the class value at that leaf node.

| **Pros** | **Cons** |
| --- | --- |
| Simple to understand, visualize & explain. | Risk of Overfitting: Can create complex trees that do not generalize well. |
| Requires little data preparation. | Can be unstable because small variations in the data might lead to overfitting. |
| Can handle both numerical and categorical data. |  |
| Performs well for a broad range of problems. |  |

Example below: If an observation has a length of 45, blue eyes, and 2 legs, it's going to be classified as red.

**Random Forest**

[sklearn.ensemble.RandomForestClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier)

Random forest is an implementation of bootstrap aggregation, aka bagging, which is an ensemble algorithm.

Bootstrapping is a statistical method for estimating a quantity from a data sample, e.g. mean. You take lots of samples of your data, calculate the mean, then average all of your mean values to give you a better estimation of the true mean value. In bootstrap aggregation, or bagging, the same approach is used for estimating entire statistical models, such as decision trees. Multiple samples of your training data are taken and models are constructed for each sample set.

When you need to make a prediction for new data, each model makes a prediction and the predictions are averaged to give a better estimation of the true output value.

Random forest is a tweak on this approach of bootstrapping, where decision trees are created so that rather than selecting optimal split points, suboptimal splits are made by introducing randomness. The models created for each sample of the data are therefore more different than they otherwise would be, in normal bootstrapping, but still accurate in their unique and different ways. This combines their prediction results in a better estimate of the true underlying output value.

If you get good results with an algorithm with high variance (like decision trees), you can often get better results by bagging that algorithm, e.g. using a random forest.

| **Pros** | **Cons** |
| --- | --- |
| Less risk of overfitting than with a decision tree. | High demand on computational resources. |
| More accurate than decision trees in most cases. | Difficult to implement. |
|  | Somewhat of a blackbox model, difficult to explain. |

**Logistic Regression**

[sklearn.linear\_model.LogisticRegression](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)

* + - Technically a regression algorithm (Goal is to find the values for the coefficients that weight each input variable).
    - Used to predict binary outcomes.
    - The output is a value between 0 and 1 that represents the probability of one class over the other.

| **Pros** | **Cons** |
| --- | --- |
| Interpretable: Good for understanding the influence of several independent variables on a single outcome variable. | Need to remove attributes which are either unrelated to the output variable or correlated to other attributes. |
| Flexible: We can choose to ‘snap’ predictions to 0 and 1 via a rule (such as if < .5, 0 else 1) OR we can choose to use the output as is, which is a probability of being class 1. | Not one of the top performing classification algorithms. |
| Easy to implement, meaning it is good to use for creating a benchmark. |  |
| Very efficient and does not require many computational resources. |  |

**Linear vs Logistic Regression**

**Logistic Regression**

**K-Nearest Neighbor**

**Description**

[sklearn.neighbors.KNeighborsClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html)

K-Nearest Neighbor (KNN) makes predictions based on how close a new data point is to known data points.

It is considered "lazy" as it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the K nearest neighbors of each point.

Predictions are made for a new data point by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances. For regression problems, this might be the mean output variable. For classification problems, this might be the mode (or most common) class value.

It is important to define a metric to measure the similarity between data instances. Euclidean distance can be used if attributes are all on the same scale (or you convert them to the same scale).

| **Pros** | **Cons** |
| --- | --- |
| Simple to implement. | Need to determine the value of K. |
| Robust to noise. | High Computational Cost: It has to compute the distance of each instance to all the training samples...you have to hang on to your entire training dataset. |
| Performs calculations "just in time", i.e. when a prediction is needed (as opposed to ahead of time) | "Curse of dimensionality": Distance can break down in very high dimensions, negatively affecting the performance. |
| Training instances can be updated and curated over time to keep predictions accurate. |  |

* + - [Tutorial on an implementation of KNN in python](https://www.kdnuggets.com/2016/01/implementing-your-own-knn-using-python.html)

**Support Vector Machine**

A technique that uses higher dimensions to best separate data points into two classes.

Support Vector Machines select hyperplanes (a line that splits the input variable space) to best separate the points in the input variable space by their class, either class 0 or class 1. In two-dimensions, you can visualize this as a line.

An optimization algorithm is used to find the values for the coefficients that maximizes the margin. The distance between the hyperplane and the closest data points is referred to as the **margin**. The best or optimal hyperplane that can separate the two classes is the line that has the largest margin. Only these points, called the **support vectors**, are relevant in defining the hyperplane and in the construction of the classifier.

| **Pros** | **Cons** |
| --- | --- |
| Effective in high dimensional spaces | Does not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation. |
| Memory efficient: Uses a subset of training points in the decision function |  |
| Highly successful classifier |  |

**Naïve Bayes**

Naive Bayes is based on Bayes’ theorem that assumes independence between every pair of features.

It is comprised of two types of probabilities that can be calculated directly from your training data:

* + - The probability of each class
    - The conditional probability for each class given each x value

Once calculated, the probability model can be used to make predictions for new data using Bayes Theorem. When your data is real-valued it is common to assume a Gaussian distribution (bell curve) so that you can easily estimate these probabilities. (so normalize your data!)

It assumes that each input variable is independent (which is often not the case), thus it is called "naive". This is a strong assumption and unrealistic for real data, nevertheless, the technique is very effective on a large range of complex problems, including document classification and spam filtering.

| **Pros** | **Cons** |
| --- | --- |
| Works with a smaller sample size of training data than other classifiers | Can be a bad estimator if used in less than ideal problems. |
| Extremely fast compared to more sophisticated methods. |  |
| Simple & Powerful |  |

Use cases:

* + - Based on their purchase and browsing history, what promos should I offer to my customers?
    - Learn from IB to develop methods for prospecting new customers
  + Decision Tree

**Decision Tree**

aka **CART:** Classification And Regression Trees

**What are Decision Trees?**

* + - Remember that classification is a **supervised** machine learning process. That means we train on labeled data.
    - In Decision Trees, we use the training data to train the tree to find a decision boundary to use as a **decision rule** for future data.
    - Decision trees are like playing "20 Questions" with your features used to predict the target. Each question is a "Yes" or a "No". That number of questions is the **depth of your tree**.
    - Given enough depth, decision trees are **overfitting** machines.

**About Decision Trees**

* + - A sequence of rules that can be used to classify 2 or more classes
    - Each node represents a single input variable (x) and a split point or class of that variable
    - The leaf nodes of the tree contain an output variable (y) which is used to make a prediction.
    - Predictions are made by walking the splits of the tree until arriving at a leaf node and output the class value at that leaf node.

**A Sample Titanic Decision Tree**

**Pros of Decision Trees**

* + - Simple to understand
    - Simple to visualize
    - Simple to explain the output
    - Requires little data preparation
    - We don't need to encode our target variable
    - Perform well for a broad range of problems

**Cons of Decision Trees**

* + - Can create complex trees that do not generalise well.
    - Can be unstable because small variations in the data might lead to overfitting.

**How it works**

* + - Classification algorithms use training data to measure the distance between points or the distance around boundaries between points.
    - By "learning" the pattern recognition around sets of labeled points, the classifier produces a **decision rule** to use to apply to classify new incoming data.

**Example**

Iris dataset: Identifing the iris species based on 4 measurements.

This algorithm is quite simple. The idea is to take the observations and split them into subcategories until we have a tree showing us how to classify something.

!<https://github.com/ryanorsinger/machine-learning-classification-workshop/blob/master/decision_tree_diagram.png?raw=true>

`# ignore warnings import warnings warnings.filterwarnings("ignore")

import numpy as np

from pydataset import data

from sklearn.model\_selection import train\_test\_split from sklearn.tree import DecisionTreeClassifier, plot\_tree from sklearn.metrics import classification\_report from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt import seaborn as sns

**read Iris data from pydatset**

df = data('iris')

df.head()`

|  | **Sepal.Length** | **Sepal.Width** | **Petal.Length** | **Petal.Width** | **Species** |
| --- | --- | --- | --- | --- | --- |
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

Let's cleanup the column names a bit:

# convert column names to lowercase, replace '.' in column names with '\_' df.columns = [col.lower().replace('.', '\_') for col in df]

**Train Validate Test**

Now we'll do our train/validate/test split: - We'll do exploration and train our model on the train data - We tune our model on validate, since it will be out-of-sample until we use it. - And keep the test nice and safe and separate, for our final out-of-sample dataset, to see how well our tuned model performs on new data.

`from sklearn.model\_selection import train\_test\_split

def train\_validate\_test\_split(df, target, seed=123): ''' This function takes in a dataframe, the name of the target variable (for stratification purposes), and an integer for a setting a seed and splits the data into train, validate and test. Test is 20% of the original dataset, validate is .30\*.80= 24% of the original dataset, and train is .70\*.80= 56% of the original dataset. The function returns, in this order, train, validate and test dataframes. ''' train\_validate, test = train\_test\_split(df, test\_size=0.2, random\_state=seed, stratify=df[target]) train, validate = train\_test\_split(train\_validate, test\_size=0.3, random\_state=seed, stratify=train\_validate[target]) return train, validate, test`

`# split into train, validate, test train, validate, test = train\_validate\_test\_split(df, target='species', seed=123)

**create X & y version of train, where y is a series with just the target variable and X are all the features.**

X\_train = train.drop(columns=['species']) y\_train = train.species

X\_validate = validate.drop(columns=['species']) y\_validate = validate.species

X\_test = test.drop(columns=['species']) y\_test = test.species`

| **Object** | **Description** | **Purpose** |
| --- | --- | --- |
| 1. df | Dataframe, Features and target | New features, additional cleaning needed, etc. |
| 2. X\_train | Dataframe, Features | Feature selection, fit models, make predictions |
| 3. y\_train | Series, Target | Feature selection, evaluate model predictions |
| 4. X\_validate | Dataframe, Features | Make predictions using top models |
| 5. y\_validate | Series, Target | Evaluate model predictions made from X\_validate to assess overfitting |
| 6. X\_test | Dataframe, Features | Make predictions using best model |
| 7. y\_test | Series, Target | Evaluate model predictions made from X\_test to estimate future performance on new data |

**Train Model**

**Create the object**

Create the Decision Tree object with desired hyper-parameters.

`# for classification you can change the algorithm to gini or entropy (information gain).

**Default is gini.**

clf = DecisionTreeClassifier(max\_depth=3, random\_state=123)`

**Fit the model**

Fit the random forest algorithm to the training data.

`# model.fit(X, y)

clf = clf.fit(X\_train, y\_train)`

**Visualize a Decision Tree**

By visualizing the decision tree, we can see the relative importance of different features. The most important features will be the first splits, and we can see how many observations in each category remain after each split.

plt.figure(figsize=(13, 7)) plot\_tree(clf, feature\_names=X\_train.columns, class\_names=clf.classes\_, rounded=True)

[Text(0.3333333333333333, 0.875, 'petal\_length <= 2.6\\ngini = 0.667\\nsamples = 84\\nvalue = [28, 28, 28]\\nclass = setosa'),

Text(0.16666666666666666, 0.625, 'gini = 0.0\\nsamples = 28\\nvalue = [28, 0, 0]\\nclass = setosa'),

Text(0.5, 0.625, 'petal\_length <= 4.75\\ngini = 0.5\\nsamples = 56\\nvalue = [0, 28, 28]\\nclass = versicolor'),

Text(0.3333333333333333, 0.375, 'gini = 0.0\\nsamples = 25\\nvalue = [0, 25, 0]\\nclass = versicolor'),

Text(0.6666666666666666, 0.375, 'petal\_length <= 4.95\\ngini = 0.175\\nsamples = 31\\nvalue = [0, 3, 28]\\nclass = virginica'),

Text(0.5, 0.125, 'gini = 0.5\\nsamples = 4\\nvalue = [0, 2, 2]\\nclass = versicolor'),

Text(0.8333333333333334, 0.125, 'gini = 0.071\\nsamples = 27\\nvalue = [0, 1, 26]\\nclass = virginica')]

**Make Predictions**

Classify each flower by its estimated species.

`# make prediction on train obeservations

y\_pred = clf.predict(X\_train) y\_pred[0:5]`

array(['versicolor', 'setosa', 'virginica', 'versicolor', 'setosa'],

dtype=object)

**Estimate Probability**

Estimate the probability of each species, using the training data.

y\_pred\_proba = clf.predict\_proba(X\_train) y\_pred\_proba[0:5]

array([[0. , 1. , 0. ],

[1. , 0. , 0. ],

[0. , 0.03703704, 0.96296296],

[0. , 1. , 0. ],

[1. , 0. , 0. ]])

**Evaluate Model**

**Compute the Accuracy**

Remember that *Accuracy* is the number of correct predictions over the number of total instances that have been evaluated.

print('Accuracy of Decision Tree classifier on training set: {:.2f}' .format(clf.score(X\_train, y\_train)))

Accuracy of Decision Tree classifier on training set: 0.96

**Create a confusion matrix**

* + - **True Positive:** number of occurrences where y is true and y is predicted true.
    - **True Negative:** number of occurrences where y is false and y is predicted false.
    - **False Positive:** number of occurrences where y is false and y is predicted true.
    - **False Negative:** number of occurrences where y is true and y is predicted false.

`# confusion matrix

confusion\_matrix(y\_train, y\_pred)`

array([[28, 0, 0],

[ 0, 27, 1],

[ 0, 2, 26]])

y\_train.value\_counts()

versicolor 28

setosa 28

virginica 28

Name: species, dtype: int64

`import pandas as pd

labels = sorted(y\_train.unique())

pd.DataFrame(confusion\_matrix(y\_train, y\_pred), index=labels, columns=labels)`

|  | **setosa** | **versicolor** | **virginica** |
| --- | --- | --- | --- |
| setosa | 28 | 0 | 0 |
| versicolor | 0 | 27 | 1 |
| virginica | 0 | 2 | 26 |

**Create a classificaiton report**

* + - **Precision:** the higher this number is, the more you were able to pinpoint all positives correctly. If this is a low score, you predicted a lot of positives where there were none. $\frac{TP}{(TP+FP)}$
    - **Recall:** if this score is high, you didn’t miss a lot of positives. But as it gets lower, you are not predicting the positives that are actually there. $\frac{TP}{(TP+FN)}$
    - **f1-score:** The balanced harmonic mean of Recall and Precision, giving both metrics equal weight. The higher the F-Measure is, the better. $\in[0,1]$
    - **Support:** number of occurrences of each class in where y is true.

print(classification\_report(y\_train, y\_pred))

precision recall f1-score support

setosa 1.00 1.00 1.00 28

versicolor 0.93 0.96 0.95 28

virginica 0.96 0.93 0.95 28

accuracy 0.96 84

macro avg 0.96 0.96 0.96 84

weighted avg 0.96 0.96 0.96 84

**Evaluate the Model with our Validate dataset**

**Compute the accuracy of the model when run on the validate data**

* + - Because we didn't do exploration or train on validate, it is "out-of-sample".
    - Evaluating on validate means we're learning how well the model does on new data.

print('Accuracy of Decision Tree classifier on validate set: {:.2f}' .format(clf.score(X\_validate, y\_validate)))

Accuracy of Decision Tree classifier on validate set: 0.94

`# And since accuracy isn't everything

**Produce y\_predictions that come from the X\_validate**

y\_pred = clf.predict(X\_validate)

**Compare actual y values (from validate) to predicted y\_values from the model run on X\_validate**

print(classification\_report(y\_validate, y\_pred))`

precision recall f1-score support

setosa 1.00 1.00 1.00 12

versicolor 0.86 1.00 0.92 12

virginica 1.00 0.83 0.91 12

accuracy 0.94 36

macro avg 0.95 0.94 0.94 36

weighted avg 0.95 0.94 0.94 36

**Avoid Overfitting**

* + - Obtain more training data (this is always the first answer)
    - Feature engineering
    - The simplest approach is to set your max\_depth argument to a lower integer)
    - Some other concepts:
      * Use a validate split to ensure we're not overfitting hyperparameters on train
      * Tree pruning
        + Pre-modeling - setting max depth before training your model (or training your model then cranking down the max\_depth)
        + Post modeling pruning - go for totally overfit, then then prune that overfit tree. [sklearn goes into more detail here](https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html#sphx-glr-auto-examples-tree-plot-cost-complexity-pruning-py)
    - For more on validate and test sets, see <https://machinelearningmastery.com/difference-test-validation-datasets/>

**Further Resources**

* + - For more detail on decision trees, see <https://scikit-learn.org/stable/modules/tree.html>

**Exercises**

Using the titanic data, in your classification-exercises repository, create a notebook, decision\_tree.ipynb where you will do the following:

* + - What is your baseline prediction? What is your baseline accuracy? *remember: your baseline prediction for a classification problem is predicting the most prevelant class in the training dataset (the mode). When you make those predictions, what is your accuracy? This is your baseline accuracy.*
    - Fit the decision tree classifier to your training sample and transform (i.e. make predictions on the training sample)
    - Evaluate your in-sample results using the model score, confusion matrix, and classification report.
    - Compute: Accuracy, true positive rate, false positive rate, true negative rate, false negative rate, precision, recall, f1-score, and support.
    - Run through steps 2-4 using a different max\_depth value.
    - Which model performs better on your in-sample data?
    - Which model performs best on your out-of-sample data, the validate set?
    - Work through these same exercises using the Telco dataset.
    - Experiment with this model on other datasets with a higher number of output classes.
  + Random Forest

**Random Forest**

**What is a Random Forest?**

* + - Random Forest is a type **Ensemble** Machine Learning algorithm called Bootstrap Aggregation or bagging.

**How does it work?**

* + - **Bootstrapping** is a statistical method for estimating a quantity from a data sample, e.g. mean. You take lots of samples of your data, calculate the mean, then average all of your mean values to give you a better estimation of the true mean value. In bagging, the same approach is used for estimating entire statistical models, such as decision trees. Multiple samples of your training data are taken and models are constructed for each sample set. When you need to make a prediction for new data, each model makes a prediction and the **predictions are averaged** to give a better estimate of the true output value.
    - Random forest is a tweak on this approach where decision trees are created so that rather than selecting optimal split points, suboptimal splits are made by introducing randomness. The models created for each sample of the data are therefore more different than they otherwise would be, but still accurate in their unique and different ways. Combining their predictions results in a better estimate of the true underlying output value.
    - In the example below, "Will I exercise", "I" am a single observation. Each person is an observation. The model takes each observation through the forest and votes on the most frequent class for that observation to get a final prediction.

**Pros**

* + - Reduction in over-fitting
    - More accurate than decision trees in most cases
    - Naturally performs feature selection

**Cons**

* + - Slow real time prediction
    - Difficult to implement
    - Complex algorithm so difficult to explain

# ignore warnings

import warnings

warnings.filterwarnings("ignore")

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

%matplotlib inline

import seaborn as sns

from pydataset import data

# read Iris data from pydatset

df = data('iris')

# convert column names to lowercase, replace '.' in column names with '\_'

df.columns = [col.lower().replace('.', '\_') for col in df]

df.head()

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **species** |
| --- | --- | --- | --- | --- | --- |
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

**Train Validate Test**

Now we'll do our train/validate/test split:

* + - We'll do exploration and train our model on the train data
    - We tune our model on validate, since it will be out-of-sample until we use it.
    - And keep the test nice and safe and separate, for our final out-of-sample dataset, to see how well our tuned model performs on new data.

from sklearn.model\_selection import train\_test\_split

def train\_validate\_test\_split(df, target, seed=123):

'''

This function takes in a dataframe, the name of the target variable

(for stratification purposes), and an integer for a setting a seed

and splits the data into train, validate and test.

Test is 20% of the original dataset, validate is .30\*.80= 24% of the

original dataset, and train is .70\*.80= 56% of the original dataset.

The function returns, in this order, train, validate and test dataframes.

'''

train\_validate, test = train\_test\_split(df, test\_size=0.2,

random\_state=seed,

stratify=df[target])

train, validate = train\_test\_split(train\_validate, test\_size=0.3,

random\_state=seed,

stratify=train\_validate[target])

return train, validate, test

# split into train, validate, test

train, validate, test = train\_validate\_test\_split(df, target='species', seed=123)

# create X & y version of train, where y is a series with just the target variable and X are all the features.

X\_train = train.drop(columns=['species'])

y\_train = train.species

X\_validate = validate.drop(columns=['species'])

y\_validate = validate.species

X\_test = test.drop(columns=['species'])

y\_test = test.species

**Train Model**

**Create the object**

Create the Random Forest object with desired hyper-parameters.

Model parameters and default values for class sklearn.ensemble.RandomForestClassifier

* + - n\_estimators=’warn’
    - criterion=’gini’
    - max\_depth=None
    - min\_samples\_split=2
    - min\_samples\_leaf=1
    - min\_weight\_fraction\_leaf=0.0
    - max\_features=’auto’
    - max\_leaf\_nodes=None
    - min\_impurity\_decrease=0.0
    - min\_impurity\_split=None
    - bootstrap=True
    - oob\_score=False
    - n\_jobs=None
    - random\_state=None
    - verbose=0
    - warm\_start=False
    - class\_weight=None

All of these can be passed as key word arguments to RandomForestClassifier

rf = RandomForestClassifier(bootstrap=True,

class\_weight=None,

criterion='gini',

min\_samples\_leaf=3,

n\_estimators=100,

max\_depth=3,

random\_state=123)

**Fit the model**

Fit the random forest algorithm to the training data.

rf.fit(X\_train, y\_train)

RandomForestClassifier(max\_depth=3, min\_samples\_leaf=3, random\_state=123)

**Feature Importance**

Evaluate importance, or weight, of each feature.

print(rf.feature\_importances\_)

[0.08209193 0.02845967 0.47781398 0.41163442]

**Make Predictions**

Classify each flower by its estimated species.

y\_pred = rf.predict(X\_train)

**Estimate Probability**

Estimate the probability of each species, using the training data.

y\_pred\_proba = rf.predict\_proba(X\_train)

**Evaluate Model**

**Compute the Accuracy**

print('Accuracy of random forest classifier on training set: {:.2f}'

.format(rf.score(X\_train, y\_train)))

Accuracy of random forest classifier on training set: 0.98

**Create a confusion matrix**

print(confusion\_matrix(y\_train, y\_pred))

[[28 0 0]

[ 0 26 2]

[ 0 0 28]]

**Create a classificaiton report**

**Precision:** $\frac{TP}{(TP+FP)}$

**Recall:** $\frac{TP}{(TP+FN)}$

**F1-Score:** A measure of accuracy. The harmonic mean of precision & recall. The harmonic mean is the reciprocal of the arithmetic mean of the reciprocals.

$$ \text{F1} \in[0,1] $$

$$ ⁍ $$

**Support:** number of occurrences of each class.

print(classification\_report(y\_train, y\_pred))

precision recall f1-score support

setosa 1.00 1.00 1.00 28

versicolor 1.00 0.93 0.96 28

virginica 0.93 1.00 0.97 28

accuracy 0.98 84

macro avg 0.98 0.98 0.98 84

weighted avg 0.98 0.98 0.98 84

**Validate Model**

**Evaluate on Out-of-Sample data**

Compute the accuracy of the model when run on the validate dataset.

print('Accuracy of random forest classifier on test set: {:.2f}'

.format(rf.score(X\_validate, y\_validate)))

Accuracy of random forest classifier on test set: 0.97

**Exercises**

Create a new notebook, random\_forests, and work with titanic data to do the following:

* + - Fit the Random Forest classifier to your training sample and transform (i.e. make predictions on the training sample) setting the random\_state accordingly and setting min\_samples\_leaf = 1 and max\_depth = 10.
    - Evaluate your results using the model score, confusion matrix, and classification report.
    - Print and clearly label the following: Accuracy, true positive rate, false positive rate, true negative rate, false negative rate, precision, recall, f1-score, and support.
    - Run through steps increasing your min\_samples\_leaf and decreasing your max\_depth.
    - What are the differences in the evaluation metrics? Which performs better on your in-sample data? Why?

After making a few models, which one has the best performance (or closest metrics) on both train and validate?

* + KNN

**K-Nearest Neighbor**

**What is KNN?**

* + - Supervised Algorithm
    - Makes predictions based on how close a new data point is to known data points.
    - Considered a **lazy algorithm** in that it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the k nearest neighbors of each point.
    - Predictions are made for a new data point by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances. For regression problems, this might be the mean output variable. For classification problems, this might be the mode (or most common) class value.
    - It is important to define a metric to measure how similar data instances are. Euclidean distance can be used if attributes are all on the same scale (or you convert them to the same scale).

Reference: [Cambridge Coding Academy](https://cambridgecoding.wordpress.com/2016/01/16/machine-learning-under-the-hood-writing-your-own-k-nearest-neighbour-algorithm/)

**Pros**

* + - Simple to implement
    - Performs calculations "just in time", i.e. when a prediction is needed (as opposed to ahead of time)
    - Training instances can be updated and curated over time to keep predictions accurate.

**Cons**

* + - Need to determine the value of K
    - The computation cost is high as it needs to compute the distance of each instance to all the training samples...you need to hang on to your entire training dataset. Therefore, not ideal for use on big data.
    - Distance can break down in very high dimensions, negatively affecting the performance. This is know as the "Curse of dimensionality". To alleviate, only use those input variables that are most relevant to predicting the output variable.

# ignore warnings

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

import numpy as np

%matplotlib inline

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

from pydataset import data

# read Iris data from pydatset

df = data('iris')

# convert column names to lowercase, replace '.' in column names with '\_'

df.columns = [col.lower().replace('.', '\_') for col in df]

df.head()

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **species** |
| --- | --- | --- | --- | --- | --- |
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

**Train Validate Test**

Now we'll do our train/validate/test split:

* + - We'll do exploration and train our model on the train data
    - We tune our model on validate, since it will be out-of-sample until we use it.
    - And keep the test nice and safe and separate, for our final out-of-sample dataset, to see how well our tuned model performs on new data.

from sklearn.model\_selection import train\_test\_split

def train\_validate\_test\_split(df, target, seed=123):

'''

This function takes in a dataframe, the name of the target variable

(for stratification purposes), and an integer for a setting a seed

and splits the data into train, validate and test.

Test is 20% of the original dataset, validate is .30\*.80= 24% of the

original dataset, and train is .70\*.80= 56% of the original dataset.

The function returns, in this order, train, validate and test dataframes.

'''

train\_validate, test = train\_test\_split(df, test\_size=0.2,

random\_state=seed,

stratify=df[target])

train, validate = train\_test\_split(train\_validate, test\_size=0.3,

random\_state=seed,

stratify=train\_validate[target])

return train, validate, test

# split into train, validate, test

train, validate, test = train\_validate\_test\_split(df, target='species', seed=123)

# create X & y version of train, where y is a series with just the target variable and X are all the features.

X\_train = train.drop(columns=['species'])

y\_train = train.species

X\_validate = validate.drop(columns=['species'])

y\_validate = validate.species

X\_test = test.drop(columns=['species'])

y\_test = test.species

**Train Model**

**Create KNN Object**

# weights = ['uniform', 'distance'] knn = KNeighborsClassifier(n\_neighbors=5, weights='uniform')

**Fit the model**

Fit the model to the training data.

knn.fit(X\_train, y\_train)

* + - KNeighborsClassifier

KNeighborsClassifier()

**Make Predictions**

Classify each flower by its estimated species.

y\_pred = knn.predict(X\_train)

**Estimate Probability**

Estimate the probability of each species, using the training data.

y\_pred\_proba = knn.predict\_proba(X\_train)

**Evaluate Model**

**Compute the Accuracy**

print('Accuracy of KNN classifier on training set: {:.2f}' .format(knn.score(X\_train, y\_train)))

Accuracy of KNN classifier on training set: 0.99

**Create a confusion matrix**

print(confusion\_matrix(y\_train, y\_pred))

[[28 0 0]

[ 0 27 1]

[ 0 0 28]]

**Create a classification report**

**Precision:** $\frac{TP}{(TP+FP)}$

**Recall:** $\frac{TP}{(TP+FN)}$

**F1-Score:** A measure of accuracy. The harmonic mean of precision & recall. The harmonic mean is the reciprocal of the arithmetic mean of the reciprocals.

$$ \text{F1} \in[0,1] $$

$$ \text{F1}-\text{score} = \text{harmonic mean} = \frac{2}{\frac{1}{\text{precision}}+\frac{1}{\text{recall}}} $$

**Support:** number of occurrences of each class.

print(classification\_report(y\_train, y\_pred))

precision recall f1-score support

setosa 1.00 1.00 1.00 28

versicolor 1.00 0.96 0.98 28

virginica 0.97 1.00 0.98 28

accuracy 0.99 84

macro avg 0.99 0.99 0.99 84

weighted avg 0.99 0.99 0.99 84

Let's look at how this works manually and through simple visual classification. We start with 4 *labeled* samples here.

# first\_nearest\_neighbor

import pandas as pd

samples = pd.DataFrame({'a': [5.7, 5.5, 6.3],

'b': [2.6, 3.5, 2.8],

'c': [3.5, 1.3, 5.1],

'd': [1.0, 0.2, 1.5],

'target': ['versicolor', 'setosa', 'virginica']

})

samples

|  | **a** | **b** | **c** | **d** | **target** |
| --- | --- | --- | --- | --- | --- |
| 0 | 5.7 | 2.6 | 3.5 | 1.0 | versicolor |
| 1 | 5.5 | 3.5 | 1.3 | 0.2 | setosa |
| 2 | 6.3 | 2.8 | 5.1 | 1.5 | virginica |

We then add 3 new unlabeled observations. For each observation, we look to the labeled samples to see which one it is closer to, in order to find the "1st-Nearest Neighbor".

new\_obs = pd.DataFrame([[6.3, 2.8, 5.1, 1.4],

[6.25, 2.77, 5.09, 1.35],

[5.5, 3.5, 1.29, 0.3]],

columns = ['a', 'b', 'c', 'd'])

new\_obs

|  | **a** | **b** | **c** | **d** |
| --- | --- | --- | --- | --- |
| 0 | 6.30 | 2.80 | 5.10 | 1.40 |
| 1 | 6.25 | 2.77 | 5.09 | 1.35 |
| 2 | 5.50 | 3.50 | 1.29 | 0.30 |

It is pretty clear which samples each new observation corresponds to. So we add those predictions.

pred\_target = pd.DataFrame(['virginica', 'virginica', 'setosa'], columns=['pred\_target'])

pd.concat([new\_obs, pred\_target], axis=1)

|  | **a** | **b** | **c** | **d** | **pred\_target** |
| --- | --- | --- | --- | --- | --- |
| 0 | 6.30 | 2.80 | 5.10 | 1.40 | virginica |
| 1 | 6.25 | 2.77 | 5.09 | 1.35 | virginica |
| 2 | 5.50 | 3.50 | 1.29 | 0.30 | setosa |

This is what K-Nearest Neighbors is doing for us. Except it's using the distance formula to actually compute the distance and find the K sample/labeled observations with the shortest or minimum distances. Of those K samples, which species is most common? i.e. what is the mode of those neighbors?

**Validate Model**

**Evaluate on Out-of-Sample data**

Compute the accuracy of the model when run on the validate dataset.

print('Accuracy of KNN classifier on test set: {:.2f}'

.format(knn.score(X\_validate, y\_validate)))

Accuracy of KNN classifier on test set: 0.97

**Visualize Model**

import matplotlib.pyplot as plt

k\_range = range(1, 20)

scores = []

for k in k\_range:

knn = KNeighborsClassifier(n\_neighbors = k)

knn.fit(X\_train, y\_train)

scores.append(knn.score(X\_test, y\_test))

plt.figure()

plt.xlabel('k')

plt.ylabel('accuracy')

plt.scatter(k\_range, scores)

plt.xticks([0,5,10,15,20])

plt.show()

**Exercises**

Create a new notebook, knn\_model, and work with the titanic dataset to answer the following:

* + - Fit a K-Nearest Neighbors classifier to your training sample and transform (i.e. make predictions on the training sample)
    - Evaluate your results using the model score, confusion matrix, and classification report.
    - Print and clearly label the following: Accuracy, true positive rate, false positive rate, true negative rate, false negative rate, precision, recall, f1-score, and support.
    - Run through steps 1-3 setting k to 10
    - Run through steps 1-3 setting k to 20
    - What are the differences in the evaluation metrics? Which performs better on your in-sample data? Why?
    - Which model performs best on our out-of-sample data from validate?
  + Logistic Regression

**Logistic Regression**

**What is Logistic Regression?**

* + - Technically a regression algorithm (goal is to find the values for the coefficients that weight each input variable)
    - Used for predicting discrete outcomes (binomial and multinomial)
    - Because the prediction for the output is transformed using the logistic function, a non-linear function, it is a classification algorithm.
    - The output is a value between 0 and 1 that represents the probability of one class over the other.
    - Like linear regression, logistic regression works better when you remove attributes that are either unrelated to the output variable or correlated to other attributes.

**Linear vs Logistic Regression**

**Logistic Regression**

**Pros**

* + - High interpretabability. It's explainable to others, i.e. it's useful for understanding the influence of several independent variables on a single outcome variable.
    - We can choose to ‘snap’ predictions to 0 and 1 via a rule (such as if < .5, 0 else 1) OR we can choose to use the output as is, which is a probability of being class 1.
    - It’s a fast model and is a good place to start with a benchmark for comparing with other classification algorithms.
    - Very efficient and does not require many computational resources. Runs fast.
    - Outputs clear predicted probabilities.

**Cons**

* + - Assumes all predictors are independent of each other.
    - Missing values must be dealt with prior to fitting the model.
    - We can’t solve non-linear problems with logistic regression since it’s decision surface is linear.
    - Not always as accurate as other classification algorithms.

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

# ignore warnings

import warnings

warnings.filterwarnings("ignore")

%matplotlib inline

import matplotlib.pyplot as plt

import seaborn as sns

from pydataset import data

# read Iris data from pydatset

df = data('iris')

# convert column names to lowercase, replace '.' in column names with '\_'

df.columns = [col.lower().replace('.', '\_') for col in df]

# we will have 2 different target variables

dummies = pd.get\_dummies(df['species'], drop\_first=True)

df = pd.concat([df, dummies], axis=1).drop(columns=['species'])

df.head()

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **versicolor** | **virginica** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | 0 | 0 |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | 0 | 0 |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | 0 | 0 |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | 0 | 0 |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | 0 | 0 |

**Train Validate Test**

Now we'll do our train/validate/test split:

* + - We will walk through the lesson aiming to predict versicolor.
    - We'll do exploration and train our model on the train data
    - We tune our model on validate, since it will be out-of-sample until we use it.
    - And keep the test nice and safe and separate, for our final out-of-sample dataset, to see how well our tuned model performs on new data.

from sklearn.model\_selection import train\_test\_split

def train\_validate\_test\_split(df, target, seed=123):

'''

This function takes in a dataframe, the name of the target variable

(for stratification purposes), and an integer for a setting a seed

and splits the data into train, validate and test.

Test is 20% of the original dataset, validate is .30\*.80= 24% of the

original dataset, and train is .70\*.80= 56% of the original dataset.

The function returns, in this order, train, validate and test dataframes.

'''

train\_validate, test = train\_test\_split(df, test\_size=0.2,

random\_state=seed,

stratify=df[target])

train, validate = train\_test\_split(train\_validate, test\_size=0.3,

random\_state=seed,

stratify=train\_validate[target])

return train, validate, test

# split into train, validate, test

train, validate, test = train\_validate\_test\_split(df, target='versicolor', seed=123)

# create X & y version of train, where y is a series with just the target variable and X are all the features.

X\_train = train.drop(columns=['versicolor','virginica'])

y\_train = train.versicolor

X\_validate = validate.drop(columns=['versicolor','virginica'])

y\_validate = validate.versicolor

X\_test = test.drop(columns=['versicolor','virginica'])

y\_test = test.versicolor

We'll use the [LogisticRegression](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) class from the sklearn.linear\_model module to implement our model.

**Arguments**

**penalty:** str, 'l1' or 'l2', default: 'l2', Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. We will discuss l1 & l2 penalties & regularization

**C:** float, default: 1.0, Inverse of regularization strength; must be a positive float.

**class\_weight:** dict or 'balanced', default: None, Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y)). Note that these weights will be multiplied with sample\_weight (passed through the fit method) if sample\_weight is specified.

**random\_state:** set the seed for reproducibility.

**intercept\_scaling:** float, default 1. Useful only when the solver 'liblinear' is used and self.fit\_intercept is set to True and you have not already scaled your data.

**solver:** {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default: 'liblinear', but will change to 'lbfgs' in v 0.22. The solver is the algorithm to use in the optimization problem. For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones. For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes. 'newton-cg', 'lbfgs' and 'sag' only handle L2 penalty, whereas 'liblinear' and 'saga' handle L1 penalty. *If using sag and saga solvers, make sure the features are on a similar scale.*

**max\_iter:** int, default: 100, Useful only for the newton-cg, sag and lbfgs solvers, Maximum number of iterations taken for the solvers to converge.

**multi\_class:** I recommend using other algorithms for multiclass or one-vs-rest if you want to use logistic regression. options: {'ovr', 'multinomial', 'auto'}, default: 'ovr' (one-versus-rest). If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

**Model 1**

For the first model, we will set the solver to be lbfgs.

**Make the Model**

**Create the object**

# from sklearn.linear\_model import LogisticRegression logit = LogisticRegression(C=1, class\_weight={0:1, 1:99}, random\_state=123, intercept\_scaling=1, solver='lbfgs')

**Fit the model**

Fit the logistic regression algorithm to the training data.

logit.fit(X\_train, y\_train)

LogisticRegression(C=1, class\_weight={0: 1, 1: 99}, random\_state=123)

**Feature Importance**

Evaluate importance, or weight, of each feature, using the coefficients.

Evaluate the intercept of the model.

print('Coefficient: \\n', logit.coef\_) print('Intercept: \\n', logit.intercept\_)

Coefficient:

[[-0.45745489 -4.33000304 2.00440881 -2.79033335]]

Intercept:

[14.54733857]

**Make Predictions**

Estimate whether or not the species is versicolor for each observation, using the training data.

y\_pred = logit.predict(X\_train)

**Estimate Probability**

Estimate the probability of species being versicolor for each observation, using the training data.

y\_pred\_proba = logit.predict\_proba(X\_train)

**Evaluate Model**

**Compute the Accuracy**

print('Accuracy of Logistic Regression classifier on training set: {:.2f}' .format(logit.score(X\_train, y\_train)))

Accuracy of Logistic Regression classifier on training set: 0.55

**Create a confusion matrix**

print(confusion\_matrix(y\_train, y\_pred))

[[18 38]

[ 0 28]]

**Create a classificaiton report**

print(classification\_report(y\_train, y\_pred))

precision recall f1-score support

0 1.00 0.32 0.49 56

1 0.42 1.00 0.60 28

accuracy 0.55 84

macro avg 0.71 0.66 0.54 84

weighted avg 0.81 0.55 0.52 84

**Model 2**

We can create new models by changing features we feed the algorithm, hyperparameters, and/or the alogrithm itself. For this second model, we will adjust our hyperparameter, C.

**Make the Model**

**Create the object**

logit2 = LogisticRegression(C=.1, class\_weight={0:1, 1:99}, random\_state=123, intercept\_scaling=1, solver='lbfgs')

**Fit the model**

Fit the logistic regreession algorithm to the training data.

logit2.fit(X\_train, y\_train)

LogisticRegression(C=0.1, class\_weight={0: 1, 1: 99}, random\_state=123)

**Feature Importance**

Evaluate importance, or weight, of each feature, using the coefficients.

Evaluate the intercept of the model.

print('Coefficient: \\n', logit2.coef\_) print('Intercept: \\n', logit2.intercept\_)

Coefficient:

[[-0.20880009 -1.67727952 1.00954681 -0.25663236]]

Intercept:

[6.02992374]

**Make Predictions**

Estimate whether or not the species is versicolor for each observation, using the training data.

y\_pred2 = logit2.predict(X\_train)

**Estimate Probability**

Estimate the probability of species being versicolor for each observation, using the training data.

y\_pred\_proba2 = logit2.predict\_proba(X\_train)

**Evaluate Model**

**Compute the Accuracy**

print('Accuracy of Logistic Regression classifier on training set: {:.2f}' .format(logit2.score(X\_train, y\_train)))

Accuracy of Logistic Regression classifier on training set: 0.40

**Create a confusion matrix**

print(confusion\_matrix(y\_train, y\_pred2))

[[ 6 50]

[ 0 28]]

**Create a classificaiton report**

print(classification\_report(y\_train, y\_pred2))

precision recall f1-score support

0 1.00 0.11 0.19 56

1 0.36 1.00 0.53 28

accuracy 0.40 84

macro avg 0.68 0.55 0.36 84

weighted avg 0.79 0.40 0.31 84

**Precision:** $\frac{TP}{(TP+FP)}$

**Recall:** $\frac{TP}{(TP+FN)}$

**F1-Score:** A measure of accuracy. The harmonic mean of precision & recall. The harmonic mean is the reciprocal of the arithmetic mean of the reciprocals.

$$ \text{F1}\in[0,1] $$

$$ {\text{F1}}-{\text{score}} = {\text{harmonic mean}} = \frac{2}{\frac1{\text{precision}}+\frac1{\text{recall}}} $$

**Support:** number of occurrences of each class.

**Validate Models**

**Evaluate on Out-of-Sample data**

Are either overfitting? Let's validate on unseen data, X\_validate. This means we use logit & logit2 to predict on X\_validate.

# make predictions

y\_pred1 = logit.predict(X\_validate)

y\_pred2 = logit2.predict(X\_validate)

print("Model 1: solver = lbfgs, c = 1")

# accuracy of model 1

print('Accuracy: {:.2f}'.format(logit.score(X\_validate, y\_validate)))

# confusion matrix of model 1

print(confusion\_matrix(y\_validate, y\_pred1))

# classification report of model 1

print(classification\_report(y\_validate, y\_pred1))

print("Model 2: solver = lbfgs, c = .1")

# accuracy of model 2

print('Accuracy: {:.2f}'.format(logit2.score(X\_validate, y\_validate)))

# confusion matrix of model 2

print(confusion\_matrix(y\_validate, y\_pred2))

# classification report of model 2

print(classification\_report(y\_validate, y\_pred2))

Model 1: solver = lbfgs, c = 1

Accuracy: 0.53

[[ 7 17]

[ 0 12]]

precision recall f1-score support

0 1.00 0.29 0.45 24

1 0.41 1.00 0.59 12

accuracy 0.53 36

macro avg 0.71 0.65 0.52 36

weighted avg 0.80 0.53 0.50 36

Model 2: solver = lbfgs, c = .1

Accuracy: 0.33

[[ 0 24]

[ 0 12]]

precision recall f1-score support

0 0.00 0.00 0.00 24

1 0.33 1.00 0.50 12

accuracy 0.33 36

macro avg 0.17 0.50 0.25 36

weighted avg 0.11 0.33 0.17 36

**Test Model**

Using the best model, compute the accuracy of the model when run on the test data.

y\_pred = logit.predict(X\_test)

y\_pred\_proba = logit.predict\_proba(X\_test)

print("Model 1: solver = lbfgs, c = 1")

print('Accuracy: {:.2f}'.format(logit.score(X\_test, y\_test)))

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

Model 1: solver = lbfgs, c = 1

Accuracy: 0.47

[[ 4 16]

[ 0 10]]

precision recall f1-score support

0 1.00 0.20 0.33 20

1 0.38 1.00 0.56 10

accuracy 0.47 30

macro avg 0.69 0.60 0.44 30

weighted avg 0.79 0.47 0.41 30

**Visualize Model**

# create array of probabilities of being versicolor (versicolor == 1)

y\_pred\_proba = np.array([i[1] for i in y\_pred\_proba])

fig = plt.figure()

ax = fig.add\_subplot(111)

# scatter plot where x is the probabilities and y is the class (0, 1)

ax.scatter(y\_pred\_proba, y\_pred)

<matplotlib.collections.PathCollection at 0x127b10610>

**Additional Example Logistic Regressions**

[Binary Logistic Regression](https://towardsdatascience.com/building-a-logistic-regression-in-python-step-by-step-becd4d56c9c8) predicting likelihood of bank customers to sign up for a new product.

[Multinomial Logistic Regression](https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html) predicting species from the [Iris Dataset](https://archive.ics.uci.edu/ml/datasets/iris)

**Exercises**

In these exercises, we'll continue working with the titanic dataset and building logistic regression models. Throughout this exercise, be sure you are training, evaluation, and comparing models on the train and validate datasets. The test dataset should only be used for your final model.

For all of the models you create, choose a threshold that optimizes for accuracy.

Create a new notebook, logistic\_regression, use it to answer the following questions:

* + - Create a model that includes only age, fare, and pclass. Does this model perform better than your baseline?
    - Include sex in your model as well. Note that you'll need to encode or create a dummy variable of this feature before including it in a model.
    - Try out other combinations of features and models.
    - Use you best 3 models to predict and evaluate on your validate sample.
    - Choose you best model from the validation performation, and evaluate it on the test dataset. How do the performance metrics compare to validate? to train?

**Bonus1** How do different strategies for handling the missing values in the age column affect model performance?

**Bonus2**: How do different strategies for encoding sex affect model performance?

**Bonus3**: scikit-learn's LogisticRegression classifier is actually applying [a regularization penalty to the coefficients](https://en.wikipedia.org/wiki/Regularized_least_squares) by default. This penalty causes the magnitude of the coefficients in the resulting model to be smaller than they otherwise would be. This value can be modified with the C hyper parameter. Small values of C correspond to a larger penalty, and large values of C correspond to a smaller penalty.

Try out the following values for C and note how the coefficients and the model's performance on both the dataset it was trained on and on the validate split are affected.

$$ C=.01,.1,1,10,100,1000 $$

**Bonus Bonus**: how does scaling the data interact with your choice of C?

* User-Defined Functions

**User-Defined Functions**

This page shows any user-defined functions referenced but not defined in the lessons throughout this module.

**Acquire Lesson**

* + All functions are defined in the lesson.

**Prepare Lesson**

import pandas as pd

import env

def get\_connection(db, user=env.user, host=env.host, password=env.password):

return f'mysql+pymysql://{user}:{password}@{host}/{db}'

def get\_titanic\_data():

return pd.read\_sql('SELECT \* FROM passengers', get\_connection('titanic\_db'))

iris\_sql="SELECT measurements.measurement\_id, \\

measurements.sepal\_length, measurements.sepal\_width, measurements.petal\_length, \\

measurements.petal\_width, species.species\_name, species.species\_id \\

FROM measurements JOIN species ON(species.species\_id=measurements.species\_id)"

def get\_iris\_data():

return pd.read\_sql(iris\_sql,get\_connection('iris\_db'))

**Explore Lesson**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from scipy import stats

def train\_validate\_test\_split(df, target, seed=123):

'''

This function takes in a dataframe, the name of the target variable

(for stratification purposes), and an integer for a setting a seed

and splits the data into train, validate and test.

Test is 20% of the original dataset, validate is .30\*.80= 24% of the

original dataset, and train is .70\*.80= 56% of the original dataset.

The function returns, in this order, train, validate and test dataframes.

'''

train\_validate, test = train\_test\_split(df, test\_size=0.2,

random\_state=seed,

stratify=df[target])

train, validate = train\_test\_split(train\_validate, test\_size=0.3,

random\_state=seed,

stratify=train\_validate[target])

return train, validate, test

def explore\_univariate(train, cat\_vars, quant\_vars):

for var in cat\_vars:

explore\_univariate\_categorical(train, var)

print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

for col in quant\_vars:

p, descriptive\_stats = explore\_univariate\_quant(train, col)

plt.show(p)

print(descriptive\_stats)

def explore\_bivariate(train, target, cat\_vars, quant\_vars):

for cat in cat\_vars:

explore\_bivariate\_categorical(train, target, cat)

for quant in quant\_vars:

explore\_bivariate\_quant(train, target, quant)

def explore\_multivariate(train, target, cat\_vars, quant\_vars):

'''

'''

plot\_swarm\_grid\_with\_color(train, target, cat\_vars, quant\_vars)

plt.show()

violin = plot\_violin\_grid\_with\_color(train, target, cat\_vars, quant\_vars)

plt.show()

pair = sns.pairplot(data=train, vars=quant\_vars, hue=target)

plt.show()

plot\_all\_continuous\_vars(train, target, quant\_vars)

plt.show()

### Univariate

def explore\_univariate\_categorical(train, cat\_var):

'''

takes in a dataframe and a categorical variable and returns

a frequency table and barplot of the frequencies.

'''

frequency\_table = freq\_table(train, cat\_var)

plt.figure(figsize=(2,2))

sns.barplot(x=cat\_var, y='Count', data=frequency\_table, color='lightseagreen')

plt.title(cat\_var)

plt.show()

print(frequency\_table)

def explore\_univariate\_quant(train, quant\_var):

'''

takes in a dataframe and a quantitative variable and returns

descriptive stats table, histogram, and boxplot of the distributions.

'''

descriptive\_stats = train[quant\_var].describe()

plt.figure(figsize=(8,2))

p = plt.subplot(1, 2, 1)

p = plt.hist(train[quant\_var], color='lightseagreen')

p = plt.title(quant\_var)

# second plot: box plot

p = plt.subplot(1, 2, 2)

p = plt.boxplot(train[quant\_var])

p = plt.title(quant\_var)

return p, descriptive\_stats

def freq\_table(train, cat\_var):

'''

for a given categorical variable, compute the frequency count and percent split

and return a dataframe of those values along with the different classes.

'''

class\_labels = list(train[cat\_var].unique())

frequency\_table = (

pd.DataFrame({cat\_var: class\_labels,

'Count': train[cat\_var].value\_counts(normalize=False),

'Percent': round(train[cat\_var].value\_counts(normalize=True)\*100,2)}

)

)

return frequency\_table

#### Bivariate

def explore\_bivariate\_categorical(train, target, cat\_var):

'''

takes in categorical variable and binary target variable,

returns a crosstab of frequencies

runs a chi-square test for the proportions

and creates a barplot, adding a horizontal line of the overall rate of the target.

'''

print(cat\_var, "\\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\n")

ct = pd.crosstab(train[cat\_var], train[target], margins=True)

chi2\_summary, observed, expected = run\_chi2(train, cat\_var, target)

p = plot\_cat\_by\_target(train, target, cat\_var)

print(chi2\_summary)

print("\\nobserved:\\n", ct)

print("\\nexpected:\\n", expected)

plt.show(p)

print("\\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\n")

def explore\_bivariate\_quant(train, target, quant\_var):

'''

descriptive stats by each target class.

compare means across 2 target groups

boxenplot of target x quant

swarmplot of target x quant

'''

print(quant\_var, "\\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\n")

descriptive\_stats = train.groupby(target)[quant\_var].describe()

average = train[quant\_var].mean()

mann\_whitney = compare\_means(train, target, quant\_var)

plt.figure(figsize=(4,4))

boxen = plot\_boxen(train, target, quant\_var)

swarm = plot\_swarm(train, target, quant\_var)

plt.show()

print(descriptive\_stats, "\\n")

print("\\nMann-Whitney Test:\\n", mann\_whitney)

print("\\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\n")

## Bivariate Categorical

def run\_chi2(train, cat\_var, target):

observed = pd.crosstab(train[cat\_var], train[target])

chi2, p, degf, expected = stats.chi2\_contingency(observed)

chi2\_summary = pd.DataFrame({'chi2': [chi2], 'p-value': [p],

'degrees of freedom': [degf]})

expected = pd.DataFrame(expected)

return chi2\_summary, observed, expected

def plot\_cat\_by\_target(train, target, cat\_var):

p = plt.figure(figsize=(2,2))

p = sns.barplot(cat\_var, target, data=train, alpha=.8, color='lightseagreen')

overall\_rate = train[target].mean()

p = plt.axhline(overall\_rate, ls='--', color='gray')

return p

## Bivariate Quant

def plot\_swarm(train, target, quant\_var):

average = train[quant\_var].mean()

p = sns.swarmplot(data=train, x=target, y=quant\_var, color='lightgray')

p = plt.title(quant\_var)

p = plt.axhline(average, ls='--', color='black')

return p

def plot\_boxen(train, target, quant\_var):

average = train[quant\_var].mean()

p = sns.boxenplot(data=train, x=target, y=quant\_var, color='lightseagreen')

p = plt.title(quant\_var)

p = plt.axhline(average, ls='--', color='black')

return p

# alt\_hyp = ‘two-sided’, ‘less’, ‘greater’

def compare\_means(train, target, quant\_var, alt\_hyp='two-sided'):

x = train[train[target]==0][quant\_var]

y = train[train[target]==1][quant\_var]

return stats.mannwhitneyu(x, y, use\_continuity=True, alternative=alt\_hyp)

### Multivariate

def plot\_all\_continuous\_vars(train, target, quant\_vars):

'''

Melt the dataset to "long-form" representation

boxenplot of measurement x value with color representing the target variable.

'''

my\_vars = [item for sublist in [quant\_vars, [target]] for item in sublist]

sns.set(style="whitegrid", palette="muted")

melt = train[my\_vars].melt(id\_vars=target, var\_name="measurement")

plt.figure(figsize=(8,6))

p = sns.boxenplot(x="measurement", y="value", hue=target, data=melt)

p.set(yscale="log", xlabel='')

plt.show()

def plot\_violin\_grid\_with\_color(train, target, cat\_vars, quant\_vars):

cols = len(cat\_vars)

for quant in quant\_vars:

\_, ax = plt.subplots(nrows=1, ncols=cols, figsize=(16, 4), sharey=True)

for i, cat in enumerate(cat\_vars):

sns.violinplot(x=cat, y=quant, data=train, split=True,

ax=ax[i], hue=target, palette="Set2")

ax[i].set\_xlabel('')

ax[i].set\_ylabel(quant)

ax[i].set\_title(cat)

plt.show()

def plot\_swarm\_grid\_with\_color(train, target, cat\_vars, quant\_vars):

cols = len(cat\_vars)

for quant in quant\_vars:

\_, ax = plt.subplots(nrows=1, ncols=cols, figsize=(16, 4), sharey=True)

for i, cat in enumerate(cat\_vars):

sns.swarmplot(x=cat, y=quant, data=train, ax=ax[i], hue=target, palette="Set2")

ax[i].set\_xlabel('')

ax[i].set\_ylabel(quant)

ax[i].set\_title(cat)

plt.show()

* Project Part 1 - Report

**Classification Project**

**Business Goals**

* + Find drivers for customer churn at Telco. Why are customers churning?
  + Construct a ML classification model that accurately predicts customer churn
  + Present your process and findings to the lead data scientist

**Project Objectives**

* + Refine your work into a final report, in the form of a jupyter notebook, that shows the work you did, why, goals, what you found, your methodologies, and your conclusions
  + Document code, process (*data acquistion, preparation, exploratory data analysis and statistical testing, modeling, and model evaluation*), findings, and key takeaways in your report (notebook)
  + Create modules ([acquire.py](http://acquire.py), [prepare.py](http://prepare.py)) that make your process repeateable and your report (notebook) easy to read and follow
  + Ask exploratory questions of your data that will help you understand more about the attributes and drivers of customers churning (answer questions through charts and statistical tests)
  + Construct a model to predict customer churn using classification techniques, and make predictions for a group of customers
  + Walk through your report (notebook) in a 5 minute presentation with the lead data scientist
  + Be prepared to answer panel questions about your code, process, findings and key takeaways, and model

**Audience**

* + Your target audience for your notebook walkthrough is your lead data scientist. This should guide your language and level of explanations in your walkthrough.

**Deliverables**

**Github repo with the following:**

* + **Readme (.md)**
    - project description with goals
    - initial hypotheses and/or questions you have of the data, ideas
    - data dictionary
    - project planning (lay out your process through the data science pipeline)
    - instructions or an explanation of how someone else can reproduce your project and findings (What would someone need to be able to recreate your project on their own?)
    - key findings, recommendations, and takeaways from your project
  + **Final Report (.ipynb)**
    - A **Report** that has filtered out all the extraneous elements not necessary to include in the report.
    - Use markdown throughout the notebook to guide the audience. Assume the reader will not read your code blocks as you think about how much markdown guidance do you need.
    - Then, assume another reader will read ALL of your code, so make sure it is very very clearly commented. All cells with code need comments.
    - Your notebook should begin with a project overview and goals
    - Preparation should specifically call out any ways you changed the data (like handling nulls)
    - Provide the context of the target variable through a visualization (distribution of the values, e.g.)
    - Exploration should be refined in the report because now you know which visualizations and tests led to valuable outcomes.
    - Include at least 4 visualizations in the form of:
      * Question *in markdown* that you want to answer
      * Visualization
      * Statistical test (in at least 2 of your 4)
      * Provide your clear answer or takeaway *in markdown and natural language* to the question based on your exploration
    - Include your 3 best models in the final notebook to review. Show the steps and code you went through to fit the models, evaluate, and select.
    - On your best model, a chart visualizing how it performed on test would be valuable.
    - End with a conclusion that talks about your original goals and how you reached those (or didn't), the key findings, recommendations and next steps ("If I had more time, I would...")
  + **Acquire & Prepare Modules (.py)**
    - contains functions to acquire, prepare and split your data. You can have other .py files if you desire to abstract other code away from your final report.
    - Your work must be reproducible by someone with their own env.py file.
    - Functions to acquire and prepare your data should be imported and used in your final report.
    - Each of your functions are complimented with docstrings. If they are functions you borrowed from instructors, put those docstrings in your own words.
  + **Predictions (.csv)**.
    - 3 columns: customer\_id, probability of churn, and prediction of churn. (1=churn, 0=not\_churn).
    - These predictions should be from your best performing model ran on X\_test.
    - Note that the order of the y\_pred and y\_proba are numpy arrays coming from running the model on X\_test. The order of those values will match the order of the rows in X\_test, so you can obtain the customer\_id from X\_test and concatenate these values together into a dataframe to write to CSV.
  + **non-final Notebook(s) (.ipynb)**
    - there should be at least 1 non-final notebook
    - these were created while working on the project, containing exploration & modeling work (and other work), not shown in the final report

**Live Presentation**

* + A **live presentation** where you deliver the **final report** (.ipynb) (your jupyter notebook) and walk through it with the audience.
  + *If have content that you intend to skip in your presentation, it should not be included in your report, like scrolls and scrolls of visualizations.* Remember, this is a different artifact from the notebook you worked on that contains all your work. This serves a purpose of conveying information to others. And you will use it to give an overview of your project by walking through the main steps (what cleaning did you do and why, what insights did you find in exploration, what are 3 models you developed, how did the differ, and how did they compare in terms of performance? What was the best model and how do you expect it to perform in production on data it's never seen? And finally, wrap it all up in a conclusion. (5 minutes max). *You should be prepared to answer follow-up questions about your code, process, tests, model, and findings.*
  + You will have created multiple notebooks in your work. Do not be concerned about not showing all your work in your report. That is not intended. We can look back to see the work that let to your final notebook. But we want you to feel comfortable creating a report from your work that gives brief insight into the findings and how you got those findings. This is the first step of you practicing delivering a report that is abstracted away from all the details. It takes practice to feel comfortable not showing everything. It will get easier the more you practice.

**Tips**

**Sample questions**

* + Are customers with DSL more or less likely to churn?
  + What month are customers most likely to churn and does that depend on their contract type?
  + Is there a service that is associated with more churn than expected?
  + Do customers who churn have a higher average monthly spend than those who don't?

**Making recommendations**

Recommendations should be feasible. If you find that month-to-month customers churn more, we won't be surprised, but Telco is not getting rid of that plan. The fact that customers churn is not because they can; it's because they can *and* they are motivated to do so. We want your insights into *why* they are motivated to do so. We realize you will not be able to do a full causal experiment, but we would like to see some solid evidence of your conclusions.

**The pipeline**

To step through each step in the pipeline, refer to the lessons in classification. This project is intended to pull together what you have done in this module. So use those resources. If you need a refresher on the pipeline and the goals in each stage, refer to the lesson in fundamentals: [Data Science Pipeline](https://ds.codeup.com/fundamentals/data-science-pipeline/).

**Data Prep**

Prepare your data within the notebook first. Line by line, testing the functionality. Then define the function (with docstrings, of course), in the notebook. Run the function, in the notebook. Make sure it works as expected before sending it to the [prepare.py](http://prepare.py) module. If you run into errors later, bring it back into the notebook and work backwards...take it out of the function, then run it. Still issues, take out line by line to find where the error is. Don't put it back into the [prepare.py](http://prepare.py) until you know it is working! And remember, restart your kernel before trying again!

**Time management**

The hardest part of this first DS project is time management. *Don't skip planning*. And in planning, write the questions you intend to answer in exploration. This is where you can fall down the rabbit hole (exploration + coming up with new features). Complete an MVP before diving deeper. So write questions to answer before you even touch code. When you get to exploration, answer only those questions first, limit the time you allow yourself to create new features. Answer the minimum of 4 questions, create viz's, run tests, then move on. Fit 3 models with train, predict and evaluate with train. Evaluate with validate, select the best, and then evaluate on test. Make a copy of the notebook.

One will become your report. Add the markdown at the top of the report (intro, project overview, goals, etc.), add the markdown of what you are doing to prepare the data, make sure the exploration is well documented where you can walk through each question, viz/test, and answer individually. Add a summary of your exploration findings to the end of your explore section. Enhance the markdown around your modeling stage. Which eval metric did you use and why? Which performed the best and why? Add a visual of your best model showing how it performed. Add a solid conclusion with how you achieved the goals, key takeaways, next steps, how you expect the predictions you made on the customers (in the csv) to perform, and recommendations to help reduct churn and improve customer retention.

Once you have finished all that and made sure your code is commented and all deliverables are ready to go, then you can go back (in your original working notebook...NOT your report), and do more exploration or modeling! If you find something you want to add before you deliver it, cool. If you don't, no problem because you already have an MVP, Minimally Viable Product!

**Templates**

A sample project with full repo and analysis report:

[Chess Upsets Example Project](https://github.com/Johndsalas/chess_upsets_example_project)

For a basic template to follow in your final report:

[Analysis Report Template](https://github.com/Johndsalas/lesson_prep/blob/main/project_template.ipynb)

* Storytelling
  + About Storytelling

**Storytelling with Data**

With a story, what you do will be remembered.

What you recommend will be considered.

What you discover will be influential.

*"We can't make sense of statistics very well without a narrative. Our cognitive capacity shuts down in the absence of a story."* - Robyn Dawes, Psychologist specializing in human judgment

*"More parts of our brain are active when we're engaged with a narrative. Stories increase empathy, understanding and recall. Storytelling is persuasive."* - Scott Berinato, Good Charts (2016)

**Berinato's Quadrants of Visualizations**

**Examples**

"Predicting ICU Survival" full presentation [here.](https://youtu.be/Tbu2squWHgU)

**References**

Berinato, Scott. (2016). Good Charts: The HBR Guide to Making Smarter, More Persuasive Data Visualizations. Boston: Harvard Business Review Press.

Chow, Ho Ming, Raymond A. Mar, Yishheng Xu, Siyuan Liu, Suraji Wagage, and Allen R. Braun. (2015) Personal Experience with Narrated Events Modulates Functional Connectivity within Visual and Motor Systems During Story Comprehension.

Dawes, Robyn M. (1999). A Message from Pyschologists to Economists. [Journal of Economic Behavior & Organization 39](http://www.sciencedirect.com/science/article/pii/S0167268199000244)

* + Understand What Makes a Good Story

**Understand What Makes a Good Story**

* + - Understand HOW WE SEE
    - Evaluating stories through CRITIQUE

**Understand How We See**

Berinato calls out some key features in how our brains process visual information in his book, Good Charts.

* + - We DON'T consume from left to right, top to bottom.
    - We see first what stands out.
    - We only see a few things at once.
    - We seek meaning and make connections. If meaning is unclear, we will make our own.
    - We use experience and expectations to make cognitive shortcuts.
    - We process and remember pictures MUCH better than numbers. Yes, even you!

Other examples of cognitive shortcuts

* + - Red:Green ⇒ Negative:Postive
    - Red:Blue ⇒ Hot/Active:Cold/Inactive
    - Hierarchies: Top ⇒ Bottom
    - Time: Left ⇒ Right
    - Y-axes: starts at 0
    - Up:Down ⇒ North:South, Good:Bad
    - Lighter:Darker ⇒ Emptier:Fuller, Lower:Higher
    - Like Colors ⇒ Like Items
    - Connected data points ⇒ relationship from one value to the next, like time, but not categories
    - Categories arranged from Most Extreme ⇒ Least Extreme
    - Color Gradient/Saturation ⇒ Progression of Values
    - Distinct Colors ⇒ Distinct Categories

**Critiquing Stories**

Understand what makes a good story by systematically critiquing data stories and visualizations. The following 7 steps are recommended by Scott Berinato (Good Charts, 2016) to systematically critique charts.

Step 1

Step 2

Step 3

Step 4

Step 5

Step 6

Step 7

**Exercises**

**Understand Resource Map**

[Explore Understand](https://www.canva.com/design/DAEUkPLyRMc/5YaTvBZH72VV2PcAaxy_sQ/view?utm_content=DAEUkPLyRMc&utm_campaign=designshare&utm_medium=link&utm_source=sharebutton)

* + Creating the Story

**Create a Data Story**

* + - DRAW it out
    - TALK it through
    - DIGITIZE drawings
    - REFINE slides
    - PRACTICE presentation

**1. Draw It Out**

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Your Turn**

You just got word that your Senior Data Scientist loved your notebook walkthrough so much they are insisting that you present one of your recommendations for reducing churn to the telco Board! You are tasked with creating and delivering a data backed presentation that will persuade the board to act on your recommendation. To do that, you will start out by creating the notecards described above. Be sure to reference your final report from your earlier research. There will be a few notecards not directly related to slides, but after those are created (setting, goal, big idea), then you should have one notecard per slide you intend to have. Take a photo of your notecards, as they will be submitted as part of your project. Estimated Time: 1 hour.

</aside>

**Slide Structure**

* + - Title Slide - should include title, presenter’s name, and date
    - Agenda - A brief summary of what you will be presenting
    - Executive Summary - Overview of your presentation including projects goal, big Idea, key findings, and recommendations
      * Goal - What is the overall goal of the project for the stakeholder?
      * Big Idea - What is the main idea that your presentation is trying to get across?
      * Key findings - What findings from your report support your big idea?
      * Recommendations - Given that your big idea is true, what concrete actions should the company take to achieve its goal?
    - Findings - Observations from your analysis that support your big idea
    - Recommendations - Given your big idea, what concrete steps should your stakeholders take to achieve their goal? What are the expected results?
    - Conclusion (1) - give a brief summary of your report and leave the audience with your most important takeaway

**2. Talk It Through**

source: <https://chart.guide>

**3. Digitize Drawings**

Up next: Refining the slides and practicing the presentation.

**Create Resource Map**

[Explore Create](https://www.canva.com/design/DAEUkKAYm3A/QaLCO9Hz4eIPM_nIr9rGAw/view?utm_content=DAEUkKAYm3A&utm_campaign=designshare&utm_medium=link&utm_source=sharebutton)

* + Refine & Present

**Refine and Present**

**4. Refine Your Story**

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Your Turn**

Walk through the 5 steps of refining for each of your charts. Refine your slides using the refine steps as they apply. Estimated Time: 2 hours.

</aside>

**5. Practice**

Speak to *YOUR* audience

Script it out

* + - Write the script.
    - Speak the script.
    - Adjust the script to match the way you speak.
    - Repeat until it feels natural.
    - Practice until you no longer need to read.

Record yourself

* + - Pay attention to the time...If it's too long, don't talk faster, but instead remove content.
    - Listen for filler words.

<aside> <img src="/icons/pencil\_blue.svg" alt="/icons/pencil\_blue.svg" width="40px" /> **Your Turn**

You are preparing for a 5 minute presentation. Write out your script. Speak the script. Adjust the script. Speak the script. Adjust. Speak and time the script. Cut where needed. Practice with a partner. Record yourself and play back the recording. Make a note on your script where you hit the half-way point (2.5 minutes) and your time at the end of the presentation. If your presentation is longer than 5 minutes, then add a note of where you hit the 5 minute mark. Estimated Time: 3 hours.

</aside>

**Refine and Present Resource Maps**

[Explore Refine](https://www.canva.com/design/DAEUkK0f27U/W7AJj62rVSeNQNkNqNnSGg/view?utm_content=DAEUkK0f27U&utm_campaign=designshare&utm_medium=link&utm_source=sharebutton)

[Explore Present](https://www.canva.com/design/DAEUkGI4GrM/ZCKw2adyNVFZjL1FTmy9Tg/view?utm_content=DAEUkGI4GrM&utm_campaign=designshare&utm_medium=link&utm_source=sharebutton)

* Project Part 2 - Story

**Storytelling Project**

**Scenario:**

After seeing your notebook walkthrough, your senior data scientist is adamant that you present one of your business recommendation(s) to the Telco Board.

**Objective:**

Drawing on your Telco Classification analysis, choose one actionable recommendation for reducing churn. Package your recommendation into a presentation for senior level management explaining why they should act on your recommendation. Your recommendation must be an action that Telco could reasonably take to reduce churn, and must be supported by findings from your report.

**Stakeholders (Audience):**

Your stakeholders are the Telco board, a group of senior level managers, who have no data science experience. Be sure to package your information in a way that can be understood by general audiences. Make sure that your charts are simple, declarative, and focus exclusively on relaying the information you want to talk about.

**Deliverables:**

* + A slide deck for your presentation 2) A live presentation (**5 minute max**) of your slide deck delivered to an audience of senior level managers

**Slide Deck Contents:**

**1. Title Slide**

* + 1 slide including name, date, and descriptive title

**2. Agenda Slide**

* + 1 slide briefly explaining what you will be presenting

**3. Executive Summary Slide**

* + 1 slide that summarizes your presentation. Your executive summary slide must include the following:
    - Goal - What is the overall goal of the project for the stakeholder?
    - Big Idea - What is the main idea that your presentation is trying to get across?
    - Key findings - What findings from your report support your big idea?
    - Recommendations - Given that your big idea is true, what concrete actions should the company take to achieve its goal?

**4. Findings**

* + 1-2 slides that dive deeper into your key findings and support your **Big Idea**. This is where you will make a case for your **Big Idea**. Remember you need not put all of your findings in this section. Only include findings that are relevant your **Big Idea**.
  + This section should be supported by at least two visualizations.

**5. Recommendations**

* + 1-2 slides that dive deeper into your recommendations. This is where you will demonstrate the value of your recommendation by showing projected results of action and inaction. Use the data in your report to make a reasonable speculation about the effect of acting on your recommendation.
  + This section should be supported by at least one visualization.

**6. Conclusion**

* + 1 slide summarizing your report and leaving your audience with one major takeaway

**Scoring Criteria**

Your **visualizations** will be scored using **visualization best practices**. Consider the following best practices when producing your visualizations.

* + Useful & meaningful titles that guide us to the key message
  + Axes, labels & variables are names that mean something to the audience.
  + Font size/type appropriate for setting and audience
  + Colors and color types are appropriate
  + Minimal extraneous elements, eye travel is limited
  + Conventions are relied upon, not fought against
  + Chart types are appropriate for the purpose and data type
  + Charts are ethical in their construction and use

Your **presentation** will be scored using **presentation best practices**. Consider the following best practices when planning and delivering your presentation.

* + Speaker begins by introducing themselves and their project
  + Speaker presents information in a logical, step-by-step manner, and stays on topic
  + Speaker presents information in a way that is appropriate for their audience
  + Speaker gives a verbal conclusion highlighting one major takeaway
  + Speaker is concise (five min max)

Your **slide formatting** will be scored using **presentation best practices**. Consider the following best practices when planning and delivering your presentation.

* + Slides have a consistent theme throughout
  + Conventions (i.e. red & green = christmas) are leaned into where possible
  + Font on text is consistent
  + Text is free of spelling errors
  + Information is conveyed in talking points rather than full sentences (avoid the wall of words effect)
  + Formatting is done in a way that supports the audiences understanding

**Optional Appendix Slides**

You may add a number of appendix slides after your conclusion slide. These may contain such things as further breakdowns of your data, citations, or anything else relevant to your recommendation that you care to include. These slides may be referenced during your presentation. For example, “For a further breakdown of these numbers see appendix slide a.” ***Appendix slides will not be directly presented and will not be graded.***

* [Overview](https://ds.codeup.com/regression/overview/)

**REGRESSION**

**Predicting Continuous Outcomes**

In this module, we will analyze, visualize and model various *labeled* datasets that are being stored in *SQL* and have *continuous target variables*. This means we will do supervised machine learning (because the data is labeled) using regression (because the target variable we are analyzing is continuous) on structured data (because the data can be naturally stored in rows and columns).

* + *Labeled* → *Supervised Learning*
  + *SQL* → *Structured Data*
  + *continuous target* → *Regression*

**Module Goals**

**Acquisition**

* + **Acquire** structured data from SQL to Pandas
  + **Summarize** Summarize the data through aggregates, descriptive stats and distribution plots (histograms, density plots, boxplots, e.g.). (pandas: .value\_counts, .head, .shape, .describe, .info, matplotlib.pyplot.hist, seaborn.boxplot)

**Preparation**

* + **Clean** the data by converting datatypes and handle missing values. (pandas: .isnull, .value\_counts, .dropna, .replace)
  + **Split** our observations into 3 samples, Train, Validate, and Test. (sklearn.model\_selection.train\_test\_split).

**Exploration**

* + **Hypothesize**: We will discuss the meaning of "drivers", variables vs. features, and the target variable. We will discuss the importance of documenting questions and hypotheses, obtaining answers for those questions, and documenting takeaways and findings at each step of exploration.
  + **Visualize** the interaction of variables, especially independent variables with the dependent variable using charts such as scatterplots, jointplots, pairgrids, and heatmaps to identify drivers.
  + **Test Hypotheses** that involve a continuous variable using t-tests and correlation tests.

**Preprocess**

* + **Encode** any string variables to 0 and 1. Utilize one hot encoding or label encoding.
  + **Scale** our numeric data so that all variables are on the same scale, such as between 0 and 1. We will discuss the importance of "scaling", different methods for scaling data, and why to use one type over another. (sklearn.preprocessing: StandardScaler, QuantileTransformer, RobustScaler, MinMaxScaler)
  + **Feature Engineering**: We will learn ways to identify, select, and create features through feature importance. We will discuss the "Curse of Dimensionality." (sklearn.feature\_selection.f\_regression).

**Modeling**

* + **Establish Baseline**: We will learn about the importance of establishing a "baseline model" or baseline score and ways to complete this task. The baseline for regression is often computed by predicting each observation's value to be the mean or median of the dependent variable.
  + **Build Models**: We will build linear regression models. What does that mean? We will extract the patterns in the data using well established algorithms, so that we don't have to do that manually. An example of a regression algorithm is the glm (generalized linear model). The algorithm will return to us a mathematical model or function (e.g. y = 3x + 2). That function will be used to compute predictions for each observation. We will learn about the differences in the most common regression algorithms. (sklearn.linear\_model)
  + **Model Evaluation**: We will compare regression models by computing evaluation metrics, i.e. metrics that measure how well a model did at predicting the target variable. (sklearn.metrics, math.sqrt)
  + **Model Selection and Testing**: We will learn how to select a model, and we will test the model on the unseen data sample (the out-of-sample data in the validate and then test datasets).

**Building a Data Product**

We will end with an end-to-end project practicing steps of the data science pipeline from planning through model selection and delivery.

**About Regression**

**Regression** is a **supervised machine learning** technique used to model the relationship of one or more features or independent variables, (one = simple regression, more = multiple regression) to one or more target or dependent variables, (one = univariate regression, more = multivariate regression). The variables are represented by **continuous** data.

A regression algorithm attempts to find the function that best 'mimics' or 'models' the relationship between independent feature(s) and dependent target variable(s). The algorithm does this by finding the line (for simple regression) or plane (for multiple regression) that minimizes the errors in our predictions when compared to the labeled data. Once we acquire that function, it can be used to make predictions on new observations when they become available; we can simply run these new values of the independent variable(s) through the function for each observation to predict the dependent target variable(s).

The algorithm attempts to find the “best” choices of values for the parameters, which in a linear regression model are the coefficients, bi��, in order to make the formula as “accurate” as possible, i.e. minimize the error. There are different ways to define the error, but whichever evaluation metric we select, the algorithm finds the line of best fit by identifying the parameters that minimize that error.

Once estimated, the parameters (intercept and coefficients) allow the value of the target variable to be obtained from the values of the feature variables.

**Simple Linear Regression**

$$ y = b\_0 + b\_1x + \epsilon $$

In the simple linear case, our feature is $x$ and our target is $y$. The algorithm finds the parameters that minimize the error between the actual values and the estimated values. The parameters the algorithm estimates are the slope, $b\_1$, and the y-intercept, $b\_0$. $\epsilon$ is the error term or the residual value. The residual is the difference of the actual value from the predicted value.

image source: <https://towardsdatascience.com/polynomial-regression-bbe8b9d97491>

**Multiple Linear Regression**

$$ y = b\_0 + b\_1x\_1 + b\_2x\_2 + ... b\_nx\_n + \epsilon $$

In a multiple linear regression case with n features, our features are $x\_1$ through $x\_n$ and our target is $y$. The algorithm finds the parameters that minimize the error between the actual values and the estimated values. The parameters the algorithm estimates are the coefficients of the features, $b\_1$ through $b\_n$, and the y-intercept, $b\_0$. $\epsilon$ is the error term or the residual value.

**Polynomial Regression**

$$ y = b\_0 + b\_1x + b\_2x^2 + ... + b\_nx^n + \epsilon $$

In the case we have a polynomial function, we still have a linear model due to the fact that $x^i$ is in fact a feature and the coefficient/weight associated with that feature is still linear. To convert the original features into their higher order terms, we will use the PolynomialFeatures class provided by scikit-learn. Then, we train the model using Linear Regression.

image source: <https://towardsdatascience.com/polynomial-regression-bbe8b9d97491>

**Notes**

Throughout the regression curriculum you will see a module named viz imported in the code examples. This module contains some complex matplotlib plotting code, and [is available for reference here](https://ds.codeup.com/regression/viz.py). The intent of putting the code in a seperate module is to not distract from the lesson at hand.

* [Acquisition and Prep](https://ds.codeup.com/regression/acquire-and-prep/)

**Acquire and Prep - Wrangle**

In the following lessons, we will walk through the data science pipeline using the following scenario:

I'm a university teacher, and I want to know when to worry about a student's progress. I want to be able to work with any students who are at high risk of failing the class, so that I can try to prevent that from happening. I have the grades of the three exams and the final grade from last semester's class. I'm hoping I can build a prediction model that will be able to use these exams to predict the final grade within 5 points average per student.

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import numpy as np

import env

**Acquire the Data**

Let's use pandas to read mySQL table into a pandas DataFrame.

# Read data from the student\_grades table in the school\_sample database on our mySQL server.

import os

def get\_connection(db, user=env.user, host=env.host, password=env.password):

return f'mysql+pymysql://{user}:{password}@{host}/{db}'

def get\_student\_data():

filename = "student\_grades.csv"

if os.path.isfile(filename):

return pd.read\_csv(filename)

else:

# Create the url

url = get\_connection('school\_sample')

# Read the SQL query into a dataframe

df = pd.read\_sql('SELECT \* FROM student\_grades', url)

# Write that dataframe to disk for later. Called "caching" the data for later.

df.to\_csv(filename)

# Return the dataframe to the calling code

return df

df = get\_student\_data()

**Sample and Summarize**

Let's take a look at the DataFrame we brought in and document our initial findings.

df.head()

|  | **Unnamed: 0** | **student\_id** | **exam1** | **exam2** | **exam3** | **final\_grade** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 100.0 | 90 | 95 | 96 |
| 1 | 1 | 2 | 98.0 | 93 | 96 | 95 |
| 2 | 2 | 3 | 85.0 | 83 | 87 | 87 |
| 3 | 3 | 4 | 83.0 | 80 | 86 | 85 |
| 4 | 4 | 5 | 93.0 | 90 | 96 | 97 |

`# 104 rows and 5 columns coming in.

df.shape`

(104, 6)

`# Display readable summary statistics for numeric columns. Why isn't exam3 showing up?

df.describe().T`

|  | **count** | **mean** | **std** | **min** | **25%** | **50%** | **75%** | **max** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Unnamed: 0 | 104.0 | 51.500000 | 30.166206 | 0.0 | 25.75 | 51.5 | 77.25 | 103.0 |
| student\_id | 104.0 | 52.500000 | 30.166206 | 1.0 | 26.75 | 52.5 | 78.25 | 104.0 |
| exam1 | 103.0 | 78.621359 | 14.260955 | 57.0 | 70.00 | 79.0 | 92.00 | 100.0 |
| exam2 | 104.0 | 77.307692 | 10.295703 | 65.0 | 70.00 | 75.0 | 89.00 | 93.0 |
| final\_grade | 104.0 | 81.692308 | 10.918122 | 65.0 | 72.00 | 81.0 | 93.00 | 97.0 |

`# Running .info() shows us that the exam3 column is not a numeric data type; it's an object.

[df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 104 entries, 0 to 103

Data columns (total 6 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Unnamed: 0 104 non-null int64

1 student\_id 104 non-null int64

2 exam1 103 non-null float64

3 exam2 104 non-null int64

4 exam3 104 non-null object

5 final\_grade 104 non-null int64

dtypes: float64(1), int64(4), object(1)

memory usage: 5.0+ KB

**Acquire and Summarize Takeaways**

* + exam1 has 1 Null value.
  + exam1 should likely be of type int64 once the Null value is addressed.
  + There is likely an odd value in exam3, as it should be of type int64 but was read in as an object. We need to find that value.
  + Given that there are limited attributes and limited observations with missing values, dropping the observations with missing values is probably a good way to go here.

**Prepare the Data**

**Finding Null Values**

Let's check out some other ways to find Null values when you are dealing with a larger dataframe, especially one with more attributes and more missing values.

* + np.nan values have a float data type. When a column you expect to have an integer data type reads in as a float, this may be signaling that there is one or more Null values present.

**.isnull().sum()**

`# Find the total number of Null values in each column of our DataFrame.

df.isnull().sum()`

Unnamed: 0 0

student\_id 0

exam1 1

exam2 0

exam3 0

final\_grade 0

dtype: int64

**.isnull().any()**

`# Check for any Null values in each column of our DataFrame.

df.isnull().any()`

Unnamed: 0 False

student\_id False

exam1 True

exam2 False

exam3 False

final\_grade False

dtype: bool

`# Return the names for any columns in our DataFrame with any Null values.

df.columns[df.isnull().any()]`

Index(['exam1'], dtype='object')

**Finding Odd Values**

Let's find the odd value in exam3 that is causing this numeric column to be coerced into an object data type.

`# Check out the values and their frequencies from exam3 column.

df.exam3.value\_counts(dropna=False, ascending=True)`

1

95 8

87 8

86 8

85 8

94 8

70 8

79 8

75 15

96 16

78 16

Name: exam3, dtype: int64

`# Replace a whitespace sequence or empty with a NaN value and reassign this manipulation to df.

df = df.replace(r'^\s\*$', np.nan, regex=True)`

Verify that our empty string has been replaced by a null

`# Now .info() shows us that exam3 has a Null value instead of a whitespace disguised as a non-null value.

[df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 104 entries, 0 to 103

Data columns (total 6 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Unnamed: 0 104 non-null int64

1 student\_id 104 non-null int64

2 exam1 103 non-null float64

3 exam2 104 non-null int64

4 exam3 103 non-null object

5 final\_grade 104 non-null int64

dtypes: float64(1), int64(4), object(1)

memory usage: 5.0+ KB

**Drop Null Values**

Let's drop observations that have any Null values; in this case, we have so few that we can simply drop rows instead of imputing values to save observations.

`# Drop all rows with any Null values, assign to df, and verify.

df = df.dropna() [df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

Int64Index: 102 entries, 0 to 102

Data columns (total 6 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Unnamed: 0 102 non-null int64

1 student\_id 102 non-null int64

2 exam1 102 non-null float64

3 exam2 102 non-null int64

4 exam3 102 non-null object

5 final\_grade 102 non-null int64

dtypes: float64(1), int64(4), object(1)

memory usage: 5.6+ KB

**Convert Data Types**

Let's convert any data types we need to at this point.

`# Change all column data tyes to int64, reassign to df, and verify.

df = df.astype('int') [df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

Int64Index: 102 entries, 0 to 102

Data columns (total 6 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Unnamed: 0 102 non-null int64

1 student\_id 102 non-null int64

2 exam1 102 non-null int64

3 exam2 102 non-null int64

4 exam3 102 non-null int64

5 final\_grade 102 non-null int64

dtypes: int64(6)

memory usage: 5.6 KB

You may want to fill your missing values with a value instead of dropping the rows. One way to do this is to apply the .fillna() method to your dataframe.

`# Default arguments for value and method parameters.

df.fillna(value=None, method=None)`

When running .describe(), we should now see exam3 listed since we have converted it to a numeric type.

df.describe().T

|  | **count** | **mean** | **std** | **min** | **25%** | **50%** | **75%** | **max** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Unnamed: 0 | 102.0 | 51.401961 | 29.743877 | 0.0 | 26.25 | 51.5 | 76.75 | 102.0 |
| student\_id | 102.0 | 53.313725 | 29.886909 | 1.0 | 28.25 | 53.5 | 78.75 | 104.0 |
| exam1 | 102.0 | 78.833333 | 14.167375 | 57.0 | 70.00 | 79.0 | 92.00 | 100.0 |
| exam2 | 102.0 | 77.500000 | 10.297313 | 65.0 | 70.00 | 75.0 | 89.00 | 93.0 |
| exam3 | 102.0 | 84.294118 | 8.736617 | 70.0 | 78.00 | 85.0 | 94.00 | 96.0 |
| final\_grade | 102.0 | 81.970588 | 10.833991 | 65.0 | 72.00 | 81.0 | 93.00 | 97.0 |

**Visualize Distributions**

We can plot histograms and/or boxplots to see the distributions of single variables and check for skewness, outliers, and unit scales. *Note, we don't have to split our data before exploring single variables. We DO have to split our data before performing bi- and multi-variate exploration.*

**sns.displot()**

We can use Seaborn's displot to display the binned values from a column.

`# The default is bins=10.

sns.displot(x='final\_grade', data=df)

plt.title('final\_grade') plt.show()`

**plt.subplot() & .hist()**

Here we'll loop through each of the numeric columns of interest and show the distribution of each on a separate subplot.

plt.figure(figsize=(16, 3))

# List of columns

cols = ['exam1', 'exam2', 'exam3', 'final\_grade']

for i, col in enumerate(cols):

# i starts at 0, but plot nos should start at 1

plot\_number = i + 1

# Create subplot.

plt.subplot(1,4, plot\_number)

# Title with column name.

plt.title(col)

# Display histogram for column.

df[col].hist(bins=5)

# Hide gridlines.

plt.grid(False)

**sns.boxplot()**

Seaborn's .boxplot will default to plotting *all* the numeric variables if we don't specify specific x and y values.

# We don't want to plot the `student\_id` column.

plt.figure(figsize=(8,4))

# Create boxplots for all but student\_id.

sns.boxplot(data=df.drop(columns=['student\_id']))

plt.show()

**Distribution Takeaways**

* + All of the column distributions are bimodal. There seem to be more students scoring on the upper and lower edges than in the middle 80s.
  + exam3 has the highest median score, exam1 and final\_grade look to have the same or very similar medians, and exam2 has the lowest median score.
  + exam2 has the least students scoring in the upper half of the grade range and the most scoring in the lower half.
  + exam1 and final\_grade distributions look very similar in these initial charts although exam1 has a larger range in scores than final\_grade.

**Pipeline Function**

We finalize these data wrangling steps (acquire and prepare) by writing a function that will reproduce the DataFrame with the necessary changes.

import os

def get\_connection(db, user=env.user, host=env.host, password=env.password):

return f'mysql+pymysql://{user}:{password}@{host}/{db}'

def get\_student\_data():

filename = "student\_grades.csv"

if os.path.isfile(filename):

return pd.read\_csv(filename, index\_col=0)

else:

# Create the url

url = get\_connection('school\_sample')

# Read the SQL query into a dataframe

df = pd.read\_sql('SELECT \* FROM student\_grades', url)

# Write that dataframe to disk for later. Called "caching" the data for later.

df.to\_csv(filename)

# Return the dataframe to the calling code

return df

def wrangle\_grades():

'''

Read student\_grades into a pandas DataFrame from mySQL,

drop student\_id column, replace whitespaces with NaN values,

drop any rows with Null values, convert all columns to int64,

return cleaned student grades DataFrame.

'''

# Acquire data

grades = get\_student\_data()

# Replace white space values with NaN values.

grades = grades.replace(r'^\\s\*$', np.nan, regex=True)

# Drop all rows with NaN values.

df = grades.dropna()

# Convert all columns to int64 data types.

df = df.astype('int')

return df

`# Let's test out or wrangle function from above.

df = wrangle\_grades() [df.info](http://df.info)()`

<class 'pandas.core.frame.DataFrame'>

Int64Index: 102 entries, 0 to 102

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 student\_id 102 non-null int64

1 exam1 102 non-null int64

2 exam2 102 non-null int64

3 exam3 102 non-null int64

4 final\_grade 102 non-null int64

dtypes: int64(5)

memory usage: 4.8 KB

**Exercises I**

Let's review the steps we take at the beginning of each new module.

* + Create a new repository named regression-exercises in your GitHub; all of your Regression work will be housed here.
  + Clone this repository within your local codeup-data-science directory.
  + Create a .gitignore and make sure your list of 'files to ignore' includes your env.py file.
  + Ceate a README.md file that outlines the contents and purpose of your repository.
  + Add, commit, and push these two files.
  + Now you can add your env.py file to this repository to access the Codeup database server.
  + For these exercises, you will create wrangle.ipynb and wrangle.py files to hold necessary functions.
  + As always, add, commit, and push your work often.

**Exercises II**

Let's set up an example scenario as perspective for our regression exercises using the Zillow dataset.

As a Codeup data science graduate, you want to show off your skills to the Zillow data science team in hopes of getting an interview for a position you saw pop up on LinkedIn. You thought it might look impressive to build an end-to-end project in which you use some of their Kaggle data to predict property values using some of their available features; who knows, you might even do some feature engineering to blow them away. Your goal is to predict the values of single unit properties using the observations from 2017.

In these exercises, you will complete the first step toward the above goal: acquire and prepare the necessary Zillow data from the zillow database in the Codeup database server.

* + Acquire bedroomcnt, bathroomcnt, calculatedfinishedsquarefeet, taxvaluedollarcnt, yearbuilt, taxamount, and fips from the zillow database for all 'Single Family Residential' properties.
  + Using your acquired Zillow data, walk through the summarization and cleaning steps in your wrangle.ipynb file like we did above. You may handle the missing values however you feel is appropriate and meaningful; remember to document your process and decisions using markdown and code commenting where helpful.
  + Write a function to split your data into train, validate, and test.
  + Store all of the necessary functions to automate your process from acquiring the data to returning a cleaned dataframe with no missing values in your wrangle.py file. Name your final function wrangle\_zillow.
* [Exploration](https://ds.codeup.com/regression/explore/)

**Exploration**

Let's explore the interactions of all attributes and target variable to help discover drivers of our target variable.

"Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis, and to check assumptions with the help of summary statistics and graphical representations." - Prasad Patil

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from scipy.stats import pearsonr, spearmanr

import env

import wrangle

**Acquire and Prepare Data**

We will use the function we created and stored in our wrangle file to quickly acquire and prepare our student grades data.

`# Use our function from wrangle to acquire and prepare our data.

df = wrangle.wrangle\_grades() df.head()`

|  | **student\_id** | **exam1** | **exam2** | **exam3** | **final\_grade** |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 100 | 90 | 95 | 96 |
| 1 | 2 | 98 | 93 | 96 | 95 |
| 2 | 3 | 85 | 83 | 87 | 87 |
| 3 | 4 | 83 | 80 | 86 | 85 |
| 4 | 5 | 93 | 90 | 96 | 97 |

df.info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 102 entries, 0 to 102

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 student\_id 102 non-null int64

1 exam1 102 non-null int64

2 exam2 102 non-null int64

3 exam3 102 non-null int64

4 final\_grade 102 non-null int64

dtypes: int64(5)

memory usage: 4.8 KB

**Main Stages in Exploration**

Let's check out the main stages in performing Exploratory Data Analysis, EDA, and run through some guidance for each stage.

“A hypothesis may be simply defined as a guess. A scientific hypothesis is an intelligent guess.” – Isaac Asimov

**Hypothesize:** Form and document your initial hypotheses about how the predictors (independent variables, features, or attributes) interact with the target (y-value or dependent variable).

* + Document your initial hypotheses. Write them down so they're concrete and not in your head.

**Visualize:** Use visualization techniques (scatterplot, jointplot, pairgrid, heatmap) to identify drivers. Sometimes a visualization is so powerful that it can suffice in identifying a driver all on its own. Other times, a visualization needs to be followed up with a statistical test. When in doubt, follow up with the appropriate test.

* + Plot out the distributions of each feature. *This is critical b/c many of our statisitical tools and machine learning algorithms assume certain distributions. If your data isn't remotely normally distributed, then avoid using any tools that assume normally distributed data.*
  + Plot out the interaction of 2 or more variables.
  + Plot out how subgroups compare to each-other and to the overall population.
  + Document any surprises you may find in visualizing. This means write down your takeaways; documenting your takeaways is a huge component of your final deliverable/analysis.
  + Identitfy features that correlate with each other. If feature A and feature B are each tightly correlated with the target variable, but they're also tightly correlated with each other, we should use one feature that correlates better, rather than use both.

**Test Your Hypotheses:** Analyze the drivers of a continuous target variable using the appropriate statistical tests (t-tests, correlation, and chi-squared hypothesis tests).

"Hypothesis generation is a process beginning with an educated guess whereas hypothesis testing is a process to conclude that the educated guess is true/false or the relationship between the variables is statistically significant or not." source

* + Document your hypothesis test results. That means writing up when the tests reject the null hypothesis or fail to reject your null hypothesis for each hypothesis you make.

**Types of Visualizations**

Here is a breakdown of visualization by type with some code useful snippets. Below, let's use the appropriate visualizations on our student grades dataset.

* + **Univariate Distributions**
    - Check out the distributions of a single variable at a time using pandas built-in plotting function to create a historgram or Seaborn displot, boxplot, or countplot; this can be done before splitting our data if we want.
    - Continuous variable distributions

df.[col].hist(grid=False, bins=10) sns.displot(x, data) sns.boxplot(data)

* + - Discrete variable distributions

sns.countplot(x='discrete\_var', data)

* + **Continuous with Continuous**
    - Seaborn pairplot to create a scatter matrix visualizing all continous variable relationships along with individiual distributions.

sns.pairplot(data)

* + - Seaborn relplot for a simple scatter plot of two continuous variables.

sns.relplot(x, y, data, kind=scatter)

* + - Seaborn lmplot for a simple scatter plot of two continous variables with a regression line. I can pass a discrete variable to col or hue to bring in another dimension, too.

sns.lmplot(x, y, data, scatter=True, hue=None, col=None)

* + - Seaborn jointplot for a simple scatter plot of two continuous variables with a regression line and the addition of a histogram for each variable.

sns.jointplot(x, y, data, kind=scatter)

* + - Seaborn heatmap of Correlation Coefficients for all numeric columns in a dataset.

sns.heatmap(train.corr())

* + **Discrete with Continuous**
    - Seaborn swarmplot or stripplot to examine a discrete variable by a continuous.

sns.swarmplot(x='discrete\_var', y='continuous\_var', data=train) sns.stripplot(x='discrete\_var', y='continuous\_var', data=train)

* + - Seaborn boxplot, violinplot, or barplot to show the distribution of a continuous variable by a discrete variable.

sns.boxplot(x='discrete\_var', y='continuous\_var', data=train) sns.violinplot(x='discrete\_var', y='continuous\_var', data=train) sns.barplot(x='discrete\_var', y='continuous\_var', data=train)

* + **Discrete with Discrete**
    - Seaborn heatmap with a pandas crosstab to examine discrete variables with discrete.

ctab = pd.crosstab(index, columns, values) sns.heatmap(ctab, annot=True)

Let's take a look at some common chart types by variable types.

**Split Data**

Before we explore bi- and multi-variate relationships, we *must* split our data to avoid leakage of unseen data.

`# Split into train, validate, and test sets; notice that we are keeping X and Y together so far.

train, validate, test = wrangle.split\_continuous(df)`

train -> (56, 5)

validate -> (25, 5)

test -> (21, 5)

**Goal**

Let's keep our goal from our student grades scenario in mind here.

I'm a university professor hoping I can build a prediction model that will be able to use these exams to predict the final grade within 5 points average per student.

Since my target variable is continuous, final\_grade, this is a regression problem. It's important to remember that Multiple linear regression analysis makes several key assumptions:

* + There must be a linear relationship between the outcome variable and the independent variables. *Scatterplots can show whether there is a linear or curvilinear relationship.*
  + No Multicollinearity—Multiple regression assumes that the independent variables are not highly correlated with each other.
  + Multivariate Normality–Multiple regression assumes that the residuals are normally distributed.

**Hypothesize**

* + My Null Hypothesis is that there is no correlation between the grades for exam1 and final grade.
  + My Alternative Hypothesis is that exam1is correlated with final grade.

**Visualize and Test**

* + To test my hypothesis, I'm going to create some visualizations and test statistics with my student grades data.
  + At the same time, I'll be checking that the key assumptions for multiple linear regression are met.

**sns.heatmap()**

Let's look at a heatmap of the correlation coefficients for a dataset. [Here](https://towardsdatascience.com/all-about-heatmaps-bb7d97f099d7) is an article with lots of heatmap customization options.

* + First, I need to calculate the correlation coefficient for each pair of variables.
  + Pandas .corr() method allows me to quickly create a correlation matrix by computing pairwise correlation of columns. By default, method=pearson.
  + I can change the .corr() argument to method=spearman if my variables are not normally distributed. Want to know more about the difference between pearson's r and spearman's rank? [This article](https://towardsdatascience.com/clearly-explained-pearson-v-s-spearman-correlation-coefficient-ada2f473b8) is short, sweet, and to the point.

`# Create the correlation matrix for all exams.

exam\_corr = train.drop(columns=['student\_id']).corr() exam\_corr`

|  | **exam1** | **exam2** | **exam3** | **final\_grade** |
| --- | --- | --- | --- | --- |
| exam1 | 1.000000 | 0.933110 | 0.940733 | 0.986033 |
| exam2 | 0.933110 | 1.000000 | 0.937750 | 0.930982 |
| exam3 | 0.940733 | 0.937750 | 1.000000 | 0.949790 |
| final\_grade | 0.986033 | 0.930982 | 0.949790 | 1.000000 |

* + Next, I pass my correlation matrix to Seaborn's heatmap along with any customization I want to perform.

`plt.figure(figsize=(8,6)) sns.heatmap(exam\_corr, cmap='Purples', annot=True, linewidth=0.5, mask= np.triu(exam\_corr)) plt.ylim(0, 4)

plt.show()`

# Pass my correlation matrix to Seaborn's heatmap.

kwargs = {'alpha':.9,'linewidth':3, 'linestyle':'-',

'linecolor':'k','rasterized':False, 'edgecolor':'w',

'capstyle':'projecting',}

plt.figure(figsize=(8,6))

sns.heatmap(exam\_corr, cmap='Purples', annot=True, mask= np.triu(exam\_corr), \*\*kwargs)

plt.ylim(0, 4)

plt.show()

# Use a scipy stats function pearsonr to calculate the correlation coefficient and the p-value.

r, p\_value = pearsonr(train.exam1, train.final\_grade)

print(f'Correlation Coefficient: {r}\\nP-value: {p\_value}')

Correlation Coefficient: 0.9860332649385699

P-value: 1.0065104366970098e-43

# Since my variables are not normally distributed, I might choose Spearman instead.

exam\_spearman = train.drop(columns=['student\_id']).corr(method='spearman')

exam\_spearman

|  | **exam1** | **exam2** | **exam3** | **final\_grade** |
| --- | --- | --- | --- | --- |
| exam1 | 1.000000 | 0.940463 | 0.918660 | 0.987083 |
| exam2 | 0.940463 | 1.000000 | 0.929203 | 0.910915 |
| exam3 | 0.918660 | 0.929203 | 1.000000 | 0.930853 |
| final\_grade | 0.987083 | 0.910915 | 0.930853 | 1.000000 |

* + My numbers come out pretty close to the same here.

plt.figure(figsize=(8,6))

sns.heatmap(exam\_spearman, cmap='Purples', annot=True, linewidth=0.5, mask= np.triu(exam\_corr))

plt.ylim(0, 4)

plt.show()

# Use a scipy stats function spearmanr to calculate the correlation coefficient and the p-value.

r, p\_value = spearmanr(train.exam1, train.final\_grade)

print(f'Correlation Coefficient: {r}\\nP-value: {p\_value}')

Correlation Coefficient: 0.9870830318356753

P-value: 1.2369246014330318e-44

**Heatmap Takeaways**

* + Although all of the exams have very high positive correlations with the target variable, exam1 and final\_grade are almost perfectly correlated. This looks to be the best predictor of our target variable.
  + Based on my correlation coefficient and my p-value, **I reject my Null hypothesis that there is no correlation between exam1 and final\_grade.**
  + Looking at the correlation between our independent variables, they also have high positive correlations with each other, multicollinearity. This informs me that I don't want to use all of them together in a linear regression model.
  + I will choose exam1 and perform a simple linear regression first. If I want to go back and do some feature engineering with my other independent variables, I could do that and see if I can improve on my accuracy.

**sns.relplot()**

Let's do a simple scatter plot of two continuous variables in our dataset.

sns.relplot(x="exam1", y="final\_grade", data=train)

plt.show()

**sns.lmplot()**

Let's make that simple scatter plot but add a regression line.

# I can really pop that line color if I want.

sns.lmplot(x="exam1", y="final\_grade", data=train, line\_kws={'color': 'red'})

plt.show()

**sns.jointplot()**

Let's use a sns.jointplot() with kind=reg to view individual variable distributions for our x and y along with a scatter plot with regression line.

sns.jointplot(x="exam1", y="final\_grade", data=train, kind='reg', height=5)

plt.show()

**Takeaways**

* + My relplot, lmplot, and joinplots charts show me that there is a linear relationship between exam1 and final\_grade.

**sns.pairplot()**

Let's use sns.pairplot() to view a scatter plot visualizing the relationships between all of the numeric columns in our dataset all at once as well as individual distributions for each individual column.

# We can drop the redundant information in the upper right half of the chart if we like.

sns.pairplot(train[['exam1', 'exam2', 'exam3', 'final\_grade']], corner=True)

plt.show()

**Additional thoughts**

* + Identify if there are logical/domain/cultural cutoffs in continuous variables that would allow us to treat them as categorial values. For example, 98.45 and 99.1 are both an A or an A+ grade in most scales.
  + If there's a logical cutoff point, like a grade of 70 or a voting age of 18, we can make a boolean to go along with a continuous value. This can allow us to gain additional insight in visualizing distributions between groups.

**Further Reading**

* + [Visualization with Seaborn Demos](https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html)
  + <https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15>
  + <https://www.itl.nist.gov/div898/handbook/index.htm>
  + <https://adataanalyst.com/data-analysis-resources/visualise-categorical-variables-in-python/>
  + Boxplot vs. Violin example <https://matplotlib.org/3.2.1/gallery/statistics/boxplot_vs_violin.html>
  + <https://datavizcatalogue.com/>

**Exercises**

Our Zillow scenario continues:

As a Codeup data science graduate, you want to show off your skills to the Zillow data science team in hopes of getting an interview for a position you saw pop up on LinkedIn. You thought it might look impressive to build an end-to-end project in which you use some of their Kaggle data to predict property values using some of their available features; who knows, you might even do some feature engineering to blow them away. Your goal is to predict the values of single unit properties using the observations from 2017.

In these exercises, you will run through the stages of exploration as you continue to work toward the above goal. Use **only** your train dataset to explore the relationships between independent variables with other independent variables or independent variables with your target variable.

* + Write a function named plot\_variable\_pairs that accepts a dataframe as input and plots all of the pairwise relationships along with the regression line for each pair.
  + Write a function named plot\_categorical\_and\_continuous\_vars that accepts your dataframe and the name of the columns that hold the continuous and categorical features and outputs 3 different plots for visualizing a categorical variable and a continuous variable.
  + Save the functions you have written to create visualizations in your explore.py file. Rewrite your notebook code so that you are using the functions imported from this file.
  + Use the functions you created above to explore your Zillow train dataset in your explore.ipynb notebook.
  + Come up with some initial hypotheses based on your goal of predicting property value.
  + Visualize all combinations of variables in some way.
  + Run the appropriate statistical tests where needed.
  + What independent variables are correlated with the dependent variable, home value?
  + Which independent variables are correlated with other independent variables (bedrooms, bathrooms, year built, square feet)?
  + Make sure to document your takeaways from visualizations and statistical tests as well as the decisions you make throughout your process.
  + Explore your dataset with any other visualizations you think will be helpful.

**Bonus Exercise**

* + In a separate notebook called explore\_mall, use the functions you have developed in this exercise with the mall\_customers dataset in the Codeup database server. You will need to write a sql query to acquire your data. Make spending\_score your target variable.
* [Scaling Numeric Data](https://ds.codeup.com/regression/split-and-scale/)

**Preprocessing: Data Scaling**

In this lesson we will discuss different ways to scale our data, that is, different ways to change the range of a feature or features.

**Background and Context**

Scaling is the process by which we normalize the numeric range of the atttributes of our data. There are several ways to do this, but as an example, we might adjust the scale of our data such that the highest value is 1 and the smallest value is 0 (this is referred to as min-max scaling).

**Why Scale Data?**

In general, working with the data in its original units is preferred. However, there are a number of occasions where we might wish to work with scaled data instead:

* + We might wish to visualize the combination of 2 variables with different scales.
  + Some statistical tests assume normality of the data. We could apply a **non-linear** scaler to a non-normally distributed feature so that this assumption is met.
  + If two features have very different scales the feature with the larger units will have a larger impact on some model types (anywhere distance is measured).
  + Sometimes scaling the data can provide a better interpretation than the original units. Times when a log scale is helpful are an example of this.
  + If we wish to combine features with different units it could be helpful to first scale both features to the same units.

**When do we scale data?**

In terms of a project lifecycle, data scaling is typically performed between the initial data exploration and feature engineering. Scaling data is an activity we will do in the prepare phase of the data science pipeline. We might wish to set up multiple datasets, one with scaled units and one without. Another option is to add additional columns to our original dataset with the scaled features.

It is important that data scaling happens **after** data splitting. We don't want to leak information from our test/validate splits by using those to calculate parameters for scaling.

**How do we scale the data?**

Data is scaled with either a **linear** or **non-linear** scaling method. A scaling method is typically a mathematical formula that can be used to transform a data point from the original dataset into a datapoint in the scaled dataset. A common convention is to notate the transformed data as $x′$ and the original as $x$.

Note that features are scaled independently. That is, the way we scale one feature does not impact the way we scale another feature. Put another way, imagine we have two features we wish to scale: x1 and x2. The mean of x1 plays no role in the way we scale x2.

Most scaling methods involve the calculation of a parameter from the dataset, for example the maximum data point, or the mean of the data. When calculating these parameters, it is important that they are derived from the training dataset. Once we know these parameters, we can apply the same scaling to every data point in the validate and test splits.

Most of the scaling methods we'll discuss have implementations in the sklearn.preprocessing module. The objects there are used like many other sklearn objects: first by fitting and then by transforming. A fit object can be used to either scale an unscaled feature or to transform a scaled feature back to the original units.

**Linear vs Non-linear Scalers**

The types of scaling we can perform on our data fall into two categories: **linear** and **non-linear**. In a linear scaling operation there is a linear correspondance between the original and the scaled value. A linear scaling operation maintains the "shape" of the distribution data and the space between data points is preserved. A non-linear scaling operation changes the shape of the data, and the distance between the points is *not* preserved.

Usually we will use a linear scaler, but non-linear scalers can be useful when changing the shape of the data is desireable, for example, when we wish to use a statistical test that assumes normality, but our data is not normally distributed.

**Setup**

For the remainder of this lesson we will demonstrate how to scale our data with various methods and what the results of scaling look like.

import matplotlib.pyplot as plt

import numpy as np

from scipy import stats

import sklearn.preprocessing

from sklearn.model\_selection import train\_test\_split

import pandas as pd

np.random.seed(123)

x = stats.skewnorm(7).rvs(1500) \* 10 + 100

x = x.reshape(-1, 1)

plt.hist(x, bins=25,ec='black')

print('Here is a histogram of the dataset we will be working with.')

Here is a histogram of the dataset we will be working with.

x\_train\_and\_validate, x\_test = train\_test\_split(x, random\_state=123)

x\_train, x\_validate = train\_test\_split(x\_train\_and\_validate)

**Min-Max Scaling**

Min-max scaling is a linear scaling method that transforms our features such that the range is between 0 and 1.

$$ x^′= \frac{x−min(x)} {max(x)−min(x)} $$

scaler = sklearn.preprocessing.MinMaxScaler()

# Note that we only call .fit with the training data,

# but we use .transform to apply the scaling to all the data splits.

scaler.fit(x\_train)

x\_train\_scaled = scaler.transform(x\_train)

x\_validate\_scaled = scaler.transform(x\_validate)

x\_test\_scaled = scaler.transform(x\_test)

plt.figure(figsize=(13, 6))

plt.subplot(121)

plt.hist(x\_train, bins=25, ec='black')

plt.title('Original')

plt.subplot(122)

plt.hist(x\_train\_scaled, bins=25, ec='black')

plt.title('Scaled')

Text(0.5, 1.0, 'Scaled')

**Standard Scaler**

**Standardization** is a linear transformation of our data such that is looks like the standard normal distribution. That is, it will have a mean of 0 and a standard deviation of 1.

$$ x^′=\frac{x−\bar{x}}{\sigma\_x} $$

Sometimes this is split into two operations:

* + **scaling** is dividing each data point by the standard deviation. This causes the resulting dataset to have a standard deviation of 1.
  + **centering** is subtracting the mean from each data point. This causes the resulting dataset to have a mean of 0.

scaler = sklearn.preprocessing.StandardScaler()

# Note that we only call .fit with the training data,

# but we use .transform to apply the scaling to all the data splits.

scaler.fit(x\_train)

x\_train\_scaled = scaler.transform(x\_train)

x\_validate\_scaled = scaler.transform(x\_validate)

x\_test\_scaled = scaler.transform(x\_test)

plt.figure(figsize=(13, 6))

plt.subplot(121)

plt.hist(x\_train, bins=25, ec='black')

plt.title('Original')

plt.subplot(122)

plt.hist(x\_train\_scaled, bins=25, ec='black')

plt.title('Scaled')

Text(0.5, 1.0, 'Scaled')

**RobustScaler**

A robust scaler is another linear transformation that follows the same idea as the standard scaler but uses parameters that are more robust to outliers.

$$ x^′=\frac{x−med(x)}{IQR\_x} $$

scaler = sklearn.preprocessing.RobustScaler()

# Note that we only call .fit with the training data,

# but we use .transform to apply the scaling to all the data splits.

scaler.fit(x\_train)

x\_train\_scaled = scaler.transform(x\_train)

x\_validate\_scaled = scaler.transform(x\_validate)

x\_test\_scaled = scaler.transform(x\_test)

plt.figure(figsize=(13, 6))

plt.subplot(121)

plt.hist(x\_train, bins=25, ec='black')

plt.title('Original')

plt.subplot(122)

plt.hist(x\_train\_scaled, bins=25, ec='black')

plt.title('Scaled')

Text(0.5, 1.0, 'Scaled')

**Exercises**

Do your work for these exercises in a jupyter notebook named scaling. Use the zillow dataset you acquired and prepped in previous lesson. Once you are finished, you may wish to repeat the exercises on another dataset for additional practice.

* + Apply the scalers we talked about in this lesson to your data and visualize the results for the unscaled and scaled distribution .
  + Apply the .inverse\_transform method to your scaled data. Is the resulting dataset the exact same as the original data?
  + Read the documentation for sklearn's QuantileTransformer. Use normal for the output\_distribution and apply this scaler to your data. Visualize the result of your data scaling.
  + Use the QuantileTransformer, but omit the output\_distribution argument. Visualize your results. What do you notice?
  + Based on the work you've done, choose a scaling method for your dataset. Write a function within your prepare.py that accepts as input the train, validate, and test data splits, and returns the scaled versions of each. Be sure to only learn the parameters for scaling from your training data!
* [Feature Engineering](https://ds.codeup.com/regression/feature-engineering/)

**Feature Engineering**

"Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data." Jason Brownlee, [Machine Learning Mastery](https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/)

You can construct new features out of existing features, select the best features, remove the worst features, penalize features by giving them no weight in the model, and transform features, as examples.

Some feature engineering methods include:

* + Construct new features: Use domain knowledge, create products of features, etc.
  + Use statistical tests to determine each feature's usefulness in predicting the target variable. Rank the features and then select the K best features (Select K Best).
  + Recursively remove attributes to meet the number of required features and then build a model on those attributes that remain, to see if you can match or improve performance with a smaller subset (Recursive Feature Elimination).
  + Recursively remove the worst-performing features one by one till the overall performance of the model comes in an acceptable range (Backward Elimination).
  + Incorporate features one by one, starting from the predictor that exhibits the highest correlation with the dependent variable. Variables of greater theoretical importance are entered first. Once in the equation, the variable remains there (Forward Selection).
  + many, many more...

**In general, we want to use scaled data for the methods discussed in this lesson.**

import pandas as pd

import numpy as np

import wrangle

import warnings

warnings.filterwarnings("ignore")

# Here's the source for the dataset and data dictionary <https://archive.ics.uci.edu/ml/datasets/student+performance>

path = "<https://gist.githubusercontent.com/ryanorsinger/55ccfd2f7820af169baea5aad3a9c60d/raw/da6c5a33307ed7ee207bd119d3361062a1d1c07e/student-mat.csv>"

df, X\_train\_explore, \\

X\_train\_scaled, y\_train, \\

X\_validate\_scaled, y\_validate, \\

X\_test\_scaled, y\_test = wrangle.wrangle\_student\_math(path)

**Select K Best**

The goal of filter methods, such as SelectKBest, is to keep the attributes with the highest correlation to the target variable and of those features, if two are highly correlated with each other, remove one of them. With filter methods, the model is built after selecting the features. These methods identify the relevant features and subset the data with only those features.

Select K Best is a filter method, meaning the goal is to find and keep the attributes with the highest correlation to the target variable, and of those features, if two are highly correlated with each other, remove one of them.

SelectKBest will identify the K most relevant features and subset the data with only those features. Relevancy is determined by the test statistic for the chosen function or test (Chi-squared, F-regression, etc.). For regression, we will use the f-regression test to score the individual effect of each of the features (aka regressors).

from sklearn.feature\_selection import SelectKBest, f\_regression

* + Initialize the f\_selector object, setting the parameters, or instructions for the method to follow: "use the f\_regression test for scoring the features, and return to me the top 10 features", for example.

f\_selector = SelectKBest(f\_regression, k=2)

* + Fit the object to our data. In doing this, our selector scores, ranks, and identifies the top k features.

f\_selector.fit(X\_train\_scaled, y\_train)

* + SelectKBest

SelectKBest(k=2, score\_func=<function f\_regression at 0x13c2d6ca0>)

* + Transform our dataset to reduce to the k best features.

X\_reduced = f\_selector.transform(X\_train\_scaled)

print(X\_train\_scaled.shape)

print(X\_reduced.shape)

(221, 41)

(221, 2)

We can simplify Steps 1-3 in the following way:

X\_reduced2 = SelectKBest(f\_regression, k=2).fit\_transform(X\_train\_scaled, y\_train)

print(X\_reduced2.shape)

(221, 2)

We can use the inverse\_transform function to return to the original variables.

Let's say we want a list of the features we have selected. Why? Maybe we want to run various feature selection methods and want to keep track of how many times each feature was selected. We could simply grab the column names from our new dataframe above that contains only the best 2 features. However, maybe we don't need the new dataframe yet, as we aren't quite sure which features we will finally decide to keep. In that case, we could use get\_feature\_names\_out() after our object is fit and transformed.

f\_selector.get\_feature\_names\_out()

array(['G1', 'G2'], dtype=object)

To summarize, we used the SelectKBest method to select the top k features which were scored and ranked using the f-regression statistical test.

**Recursive Feature Elimination**

Recursive Feature Elimination is a *wrapper* method for feature selection. This means that it works by using the output of a machine learning algorithm as the evaluation criteria for eliminating features; in the case of linear regression, it uses the resulting coefficients.

You feed all the features to the selected Machine Learning algorithm, and, based on the hyperparameters you have set, features are removed. **One word of caution...this is an iterative and computationally expensive process!** The pro is that it is *more accurate than SelectKBest*.

RFE recursively removes attributes and then builds a model on those attributes that remain. The RFE method takes the machine learning algorithm to be used and the number of required features as input. It returns the ranking of all the variables, 1 being the most important, along with its support: a list of boolean values, True indicating relevant features and False indicating irrelevant features.

These are the steps we will take to implement RFE:

* + Initialize the linear regression object. sklearn.linear\_model.LinearRegression
  + Initialize the RFE object. sklearn.feature\_selection.RFE
  + Fit the RFE object to our data. rfe.fit()
  + Transform our X dataframe to include only n number of features. rfe.transform()
  + Optional: Get a list of features selected.
  + Optional: Get the ranking of all variables.

from sklearn.linear\_model import LinearRegression from sklearn.feature\_selection import RFE

* + Initialize the linear regression object

lm = LinearRegression()

* + Initialize the RFE object, setting the hyperparameters to be our linear regression object created above (as the algorithm to test the features on) and the number of features to return to be 2.

rfe = RFE(lm, n\_features\_to\_select=2)

* + Fit the RFE object to our data. This means you'll create multiple linear regression models, find the one that performs best, and identify the features that are used in that model. Those will be the features we want.
  + Transform our X dataframe to include only those 2 n features. .transform() *or do both of those steps together with .fit\_transform()*

# Transforming data using RFE X\_rfe = rfe.fit\_transform(X\_train\_scaled,y\_train)

We would then use our new X dataframe as the one to move forward with for actual modeling. As a sneak peak...

#Fitting the data to model lm.fit(X\_rfe,y\_train)

* + LinearRegression

LinearRegression()

* + If we want a list of the features that remain, we can use get\_feature\_names\_out(), just like SelectKBest.

rfe\_features = rfe.get\_feature\_names\_out()

print(str(len(rfe\_features)), 'selected features') print(rfe\_features)

2 selected features

['G1' 'G2']

* + We can also get a ranking of the features using rfe.ranking\_. This will return a 1 for the features that were selected. So, since we said we wanted 2 features to remain, the top two features will have a rank of 1. The features that were eliminated will be ranked accordingly. In this case, the third feature will have a rank of 2. However, if we had more than 1 feature that was eliminated, they would all have different ranks.

var\_ranks = rfe.ranking\_

var\_names = X\_train\_scaled.columns.tolist()

pd.DataFrame({'Var': var\_names, 'Rank': var\_ranks})

|  | **Var** | **Rank** |
| --- | --- | --- |
| 0 | age | 3 |
| 1 | Medu | 13 |
| 2 | Fedu | 15 |
| 3 | traveltime | 5 |
| 4 | studytime | 34 |
| 5 | failures | 9 |
| 6 | famrel | 4 |
| 7 | freetime | 39 |
| 8 | goout | 18 |
| 9 | Dalc | 19 |
| 10 | Walc | 24 |
| 11 | health | 14 |
| 12 | absences | 2 |
| 13 | G1 | 1 |
| 14 | G2 | 1 |
| 15 | school\_MS | 25 |
| 16 | sex\_M | 40 |
| 17 | address\_U | 26 |
| 18 | famsize\_LE3 | 27 |
| 19 | Pstatus\_T | 17 |
| 20 | Mjob\_health | 6 |
| 21 | Mjob\_other | 8 |
| 22 | Mjob\_services | 7 |
| 23 | Mjob\_teacher | 38 |
| 24 | Fjob\_health | 30 |
| 25 | Fjob\_other | 28 |
| 26 | Fjob\_services | 29 |
| 27 | Fjob\_teacher | 21 |
| 28 | reason\_home | 32 |
| 29 | reason\_other | 23 |
| 30 | reason\_reputation | 35 |
| 31 | guardian\_mother | 22 |
| 32 | guardian\_other | 16 |
| 33 | schoolsup\_yes | 10 |
| 34 | famsup\_yes | 12 |
| 35 | paid\_yes | 37 |
| 36 | activities\_yes | 36 |
| 37 | nursery\_yes | 33 |
| 38 | higher\_yes | 20 |
| 39 | internet\_yes | 11 |
| 40 | romantic\_yes | 31 |

Here we took LinearRegression model with 2 features and RFE gave feature ranking as above, but the selection of number 2 was random.

**Summary**

**SelectKBest**

Select the K best features using a statistical test to compare each X with y and find which X's have the strongest relationship with y. For regression, we will use the correlation test (f-regression) to score the relationships.

* + **Initialize the f\_selector object**, setting the parameters, or instructions for the method to follow: "use the *f\_regression* test for scoring the features, and return to me the top *10* features", for example.
  + **Fit the object to our data.** That is, run a correlation test for every X variable with our y variable, and then rank the X variables based on how correlated they are with the y/target variable. Then give me the top *10* features.
  + **Use get\_feature\_names\_out()** to get the list of features, and save them to a variable that you can use to filter your dataframe in modeling.

from sklearn.feature\_selection import SelectKBest, f\_regression

# parameters: f\_regression stats test, give me 10 features

f\_selector = SelectKBest(f\_regression, k=10)

# find the top 10 X's correlated with y

f\_selector.fit(X\_train\_scaled, y\_train)

# save top 10 features

f\_features = f\_selector.get\_feature\_names\_out()

f\_features

array(['age', 'Medu', 'Fedu', 'traveltime', 'failures', 'G1', 'G2',

'sex\_M', 'guardian\_other', 'higher\_yes'], dtype=object)

**Recursive Feature Elimination**

Recursive Feature Elimination will create a model with all the features, evaluate the performance metrics, find the weakest feature, remove it, then create a new model with the remaining features, evaluate the performance metrics, find the weakest feature, remove it, and so on, until it gets down to the number of features you have indicated you want when creating the RFE object. You will also need to indicate which Machine Learning algorithm you want to use.

* + **Initialize the machine learning algorithm**, in this case, LinearRegression
  + **Initialize the RFE object**, and provide the ML algorithm object from Step 1
  + **Fit the RFE object to our data.** Doing this will provide us with a list of features (the number we asked for) as well as a ranking of all the features.
  + **Assign the list** of selected features to a variable.
  + **Optional:** Get a ranking of all variables (1 being the most important)

from sklearn.linear\_model import LinearRegression

from sklearn.feature\_selection import RFE

# initialize the ML algorithm

lm = LinearRegression()

# create the rfe object, indicating the ML object (lm) and the number of features I want to end up with.

rfe = RFE(lm, n\_features\_to\_select=2)

# fit the data using RFE

rfe.fit(X\_train\_scaled,y\_train)

# get list of the column names.

rfe\_feature = rfe.get\_feature\_names\_out()

rfe\_feature

array(['G1', 'G2'], dtype=object)

# view list of columns and their ranking

# get the ranks

var\_ranks = rfe.ranking\_

# get the variable names

var\_names = X\_train\_scaled.columns.tolist()

# combine ranks and names into a df for clean viewing

rfe\_ranks\_df = pd.DataFrame({'Var': var\_names, 'Rank': var\_ranks})

# sort the df by rank

rfe\_ranks\_df.sort\_values('Rank').head(10)

|  | **Var** | **Rank** |
| --- | --- | --- |
| 14 | G2 | 1 |
| 13 | G1 | 1 |
| 12 | absences | 2 |
| 0 | age | 3 |
| 6 | famrel | 4 |
| 3 | traveltime | 5 |
| 20 | Mjob\_health | 6 |
| 22 | Mjob\_services | 7 |
| 21 | Mjob\_other | 8 |
| 5 | failures | 9 |

**Exercises**

Do your work for this exercise in a jupyter notebook named feature\_engineering within the regression-exercises repo. Add, commit, and push your work.

* + Load the tips dataset.
    - Create a column named price\_per\_person. This should be the total bill divided by the party size.
    - Before using any of the methods discussed in the lesson, which features do you think would be most important for predicting the tip amount?
    - Use Select K Best to select the top 2 features for predicting tip amount. What are they?
    - Use Recursive Feature Elimination to select the top 2 features for tip amount. What are they?
    - Why do you think Select K Best and Recursive Feature Elimination might give different answers for the top features? Does this change as you change the number of features you are selecting?
  + Write a function named select\_kbest that takes in the predictors (X), the target (y), and the number of features to select (k) and returns the names of the top k selected features based on the SelectKBest class. Test your function with the tips dataset. You should see the same results as when you did the process manually.
  + Write a function named rfe that takes in the predictors, the target, and the number of features to select. It should return the top n features based on the RFE class. Test your function with the tips dataset. You should see the same results as when you did the process manually.
  + Load the swiss dataset and use all the other features to predict Fertility. Find the top 3 features using both Select K Best and Recursive Feature Elimination (use the functions you just built to help you out).
* [Evaluating Regression Models](https://ds.codeup.com/regression/evaluate/)

**Evaluating Regression Models**

In this lesson, we will discuss how to evaluate a model's performance, i.e. measure how well a model predicts the target variable. Knowing how well a model performs requires establishing a baseline, computing metrics, and comparing those metrics across multiple models (including the baseline).

* + You will learn ways to define "good enough" and have a baseline.
  + You will learn about various regression evaluation metrics, how to compute them manually, how to compute them using sklearn.metrics, and why you would use one over another.
  + You will learn how to select your best model using these evaluation metrics.

**Sample Scenario**

My students have just completed the first exam in the spring semester literature class. I want to know if there is anyone I should be concerned about at this point so that we can get ahead of a problem before it forms.

I have the data from last semester's class. Neither the content nor the exams have changed in any way, so I'm hoping I can use that data to find the line that most closely predicts final grades. Once I find the "Line of Best Fit" or the "Regression Line", I can use the equation to calculate the current students' final grades using their first exam grades.

There are 3 questions we need to answer:

* + How do we know if our model is good enough?
  + How do we find the line of best fit?
  + How do we evaluate our model?
  + Are our features valuable?

First, we will generate the sample data.

import pandas as pd

import numpy as np

# generate our student grade data

df = pd.DataFrame(np.array([[100, 96], [93, 90], [84, 89], [80, 85], [76, 80], [70, 68], [79, 75]]), columns = ['x', 'y'])

x = df['x']

y = df['y']

df.head(3)

|  | **x** | **y** |
| --- | --- | --- |
| 0 | 100 | 96 |
| 1 | 93 | 90 |
| 2 | 84 | 89 |

**Line of Best Fit**

When we are evaluating linear models, we are comparing the predicted points that fall on the **Line of Best Fit** to the actual data points.

Building a linear model is basically finding this line of best fit. You are using a linear regression algorithm to return the slope and y-intercept of the line that most accurately predicts y, given the x and y you provided to the algorithm. The slope and y-intercept, when input into the linear function y=mx+b�=��+�, give you the **line of best fit**, or the **regression line**.

Establishing a baseline in a simple regression problem is setting that line of best fit to be a constant value. Take a look at the chart below. Which line is the line of best fit? Which is the baseline?

import viz

import seaborn as sns

import matplotlib.pyplot as plt

plt.rc("axes.spines", top=False, right=False)

viz.evaluation\_example1(df, x, y)

**Establish a Baseline**

The very basic baseline is one that uses no model at all. This means, predict the target variable without using any features. The simples way to do this in regression is to take mean or median value and predict all future values to be that constant value. In other words, any model we build should perform better than it would if we had no inputs (i.e. no data for exam 1). As you can see above, as exam 1 changes, the final grade predicted is not impacted in any way. That tells us that the function for that line does not include x, or exam 1, in it at all. (You could also think of it as having a coefficient, or slope, of 0.)

In our sample scenario, we could define our baseline by predicting all final grades for this semester to be the mean or the median final grade of those from last semester. Later we will evaluate both models and if our regression line does not perform better than our baseline, then we do not more forward with the model as is. Let's set the baseline predictions to be the mean of all final grades.

baseline predictions: $\^{y}=\mu\_y$

df['yhat\_baseline'] = df['y'].mean()

df.head(3)

|  | **x** | **y** | **yhat\_baseline** |
| --- | --- | --- | --- |
| 0 | 100 | 96 | 83.285714 |
| 1 | 93 | 90 | 83.285714 |
| 2 | 84 | 89 | 83.285714 |

**Build a Simple Model**

We want to find the regression line where exam 1 is the feature or single independent variable and final grade is our target or dependent variable. We have a labeled dataset, so this is supervised machine learning. In short, we are going to provide a regression algorithm with our labeled data (our x and y values) and get in return the parameters needed for the optimal regression line. The parameters needed in a simple regression problem is the y-intercept and the slope/coefficient.

We will go into this in more detail in the modeling lesson, but for now we will wave the magic wand to find that line.

from sklearn.linear\_model import LinearRegression

# generate parameters, i.e. create model

ols\_model = LinearRegression().fit(df[['x']], df.y)

# compute predictions and add to original dataframe

df['yhat'] = ols\_model.predict(df[['x']])

df.head(3)

|  | **x** | **y** | **yhat\_baseline** | **yhat** |
| --- | --- | --- | --- | --- |
| 0 | 100 | 96 | 83.285714 | 97.635214 |
| 1 | 93 | 90 | 83.285714 | 91.676524 |
| 2 | 84 | 89 | 83.285714 | 84.015350 |

Let's plot our line of best fit and compare visually to our baseline (and a couple other lines we "guessed")

viz.evaluation\_example2(df, x, y)

How could we determine which line is best, numerically? We will do that next.

**Evaluate Part 1: RMSE**

**Is it "good enough"**

Now we need to measure the performance of the baseline and the line created from our model. We need to determine if our model is "good enough". To determine this, we have to answer two questions.

* + Is the model better than having no model at all? In other words, is using a variable input better than just predicting based on the existing value of the dependent variable (such as the mean or previous value)? We will compute evaluation metrics, such as Mean Squared Error, Root Mean Squared Error, or Median Absolute Error, for the model and the baseline, and compare them to each other.
  + In the case of a simple regression model, "Is our coefficient of determination, or $R^2$, significant?" We will run an F-test and check the p-value of the test statistic to see if the p-value is significant, or less than our alpha (.05).

**Manually**

**Manually Compute Evaluation Metrics**.

In this lesson, we will manually compute two common evaluation metric for regression models, the Mean Squared Error (MSE) and the Root Mean Squared Error (RMSE). To do so, we take the following steps:

* + Compute the **residual**, or error, for each data point.
  + Compute the **SSE**, Sum of Squared Errors, a.k.a. **RSS**, Residual Sum of Squares. This is simply squaring each of the errors computed in step one and summing them all together.
  + Compute the **MSE**, Mean Squared Error. We arrive at this by dividing your SSE by the total number of data points, i.e. the average of your errors that have each been squared.
  + Compute the **RMSE**, Root Mean Squared Error. Simply take the square root of the MSE.

We will do this for both the predictions from the model and those from the baseline. We will then compare the final value of each. Whichever has the lower value is the better prediction.

**Residuals**

The thick green line is our regression line. It passes through the values for exam1 ($x$) and the predicted values for final grade ($\^y$)

The residual of an observed value is the difference between the observed value and the estimated value. Another way to state this is that it is the vertical distance from the original data point to the expected data point (which shares an x and sits on the regression line). In this example, the residual of an observation is the difference between that students actual final grade and the expected final grade.

The residuals are the foundation for evaluating the effectiveness of a regression model.

Residuals: for each data point, $(\^y−y)$

from scipy import stats

from sklearn.metrics import mean\_squared\_error, r2\_score, explained\_variance\_score

from sklearn.linear\_model import LinearRegression

from sklearn.feature\_selection import f\_regression

from math import sqrt

import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings('ignore')

# compute residuals

df['residual'] = df['yhat'] - df['y']

df['residual\_baseline'] = df['yhat\_baseline'] - df['y']

df.head()

|  | **x** | **y** | **yhat\_baseline** | **yhat** | **residual** | **residual\_baseline** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 100 | 96 | 83.285714 | 97.635214 | 1.635214 | -12.714286 |
| 1 | 93 | 90 | 83.285714 | 91.676524 | 1.676524 | -6.714286 |
| 2 | 84 | 89 | 83.285714 | 84.015350 | -4.984650 | -5.714286 |
| 3 | 80 | 85 | 83.285714 | 80.610384 | -4.389616 | -1.714286 |
| 4 | 76 | 80 | 83.285714 | 77.205418 | -2.794582 | 3.285714 |

viz.evaluation\_example5(df, x, df.residual\_baseline)

viz.evaluation\_example5(df, x, df.residual)

Notice the trend that exists in the plot of the baseline residuals the we do not see in the second plot, that of the model residuals. This pattern in the residuals of the baseline indicates there is more information we can use from exam 1 to predict our y, the final exam grade.

Now, we will use the residuals to compute the Sum of Squared Errors, aka Residual Sum of Squares.

**SSE**

Sometime the Sum of the Squared Errors (SSE, a.k.a RSS, Residual Sum of Squares) will be used as the final metric to evaluate. Most times, however, this is used as a stepping stone to the other metrics, such as MSE and RMSE. If outliers matter, this is a good metric to use.

The value of the SSE is derived by simply squaring each of the errors computed in step one and summing them all together.

$$ SSE=\stackrel{n}{\sum\_{i=1}}(\^y - y\_i)^2 $$

# square each residual value

df['residual^2'] = df.residual \*\* 2

df['residual\_baseline^2'] = df.residual\_baseline \*\* 2

df.head(3)

|  | **x** | **y** | **yhat\_baseline** | **yhat** | **residual** | **residual\_baseline** | **residual^2** | **residual\_baseline^2** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 100 | 96 | 83.285714 | 97.635214 | 1.635214 | -12.714286 | 2.673926 | 161.653061 |
| 1 | 93 | 90 | 83.285714 | 91.676524 | 1.676524 | -6.714286 | 2.810732 | 45.081633 |
| 2 | 84 | 89 | 83.285714 | 84.015350 | -4.984650 | -5.714286 | 24.846737 | 32.653061 |

# SSE

SSE = sum(df['residual^2'])

SSE\_baseline = sum(df['residual\_baseline^2'])

print("SSE = ", SSE)

print("SSE - baseline = ", SSE\_baseline)

SSE = 96.85259593679456

SSE - baseline = 555.4285714285714

At second glance (after visualizing the residual plot), it appears the model is performing better than the baseline. However, it could be do to a single outlier that drastically has increased the SSE. (Well, we know it's not becuase we plotted the residuals, but if we hadn't, that could be the case.)

**MSE**

Next, we can use the SSE to compute the Mean Squared Error, MSE. We arrive at this by dividing your SSE by the total number of data points, i.e. the average of your errors that have each been squared. If outliers don't matter as much, but cost exponential instead of linear, then this is a good metric to use. That means that a residual of 10 (the expected value is 10 units off the actual value) is greater than twice a residual of 5.

$$ MSE=\frac{1}{n} \stackrel{n}{\sum\_{i=1}} (\^y - y\_i)^2 $$

MSE = SSE/len(df)

MSE\_baseline = SSE\_baseline/len(df)

print("MSE = ", MSE)

print("MSE baseline = ", MSE\_baseline)

MSE = 13.836085133827794

MSE baseline = 79.34693877551021

It is clear the model is performing better than the baseline.

**RMSE**

Now, we can use the MSE to compute the Root Mean Squared Error, RMSE. Simply take the square root of the MSE. If you want to see the error in the actual units of the y variable, then this is a good metric to use. In this case, the units will be grade points.

$$ RMSE=\sqrt{ \frac{1}{n} \stackrel{n}{\sum\_{i=1}} (\^y - y\_i)^2 } $$

from math import sqrt

RMSE = sqrt(MSE)

RMSE\_baseline = sqrt(MSE\_baseline)

print("RMSE = ", RMSE)

print("RMSE baseline = ", RMSE\_baseline)

RMSE = 3.719688849060872

RMSE baseline = 8.907689867497083

We can see that the model is off by an average of 3.7 grade points, while the baseline is off by an average of 8.9 grade points. The model does better than having no model at all.

**Scikit-Learn**

**Use scikit-learn to compute evaluation metrics**

Now, instead of manually computing the SSE, MSE, and RMSE, we will use sklearn.metrics to do so.

**SSE**

To compute the SSE or RSS, we must use the mean\_squared\_error function, as sklearn.metrics does not have a function for SSE directly. All we need to do is multiply the MSE by n, the number of data points.

Take a look:

$$ MSE = \frac{1}{n} \sum {^{n}\_{i=1}} (\^y - y\_i)^2 $$

$$ n\*MSE = \sum{*{i=1}^{n}} (\hat{y}-y*{i})^2 = SSE $$

$$ \therefore SSE = n\*MSE $$

from sklearn.metrics import mean\_squared\_error

SSE2 = mean\_squared\_error(df.y, df.yhat)\*len(df)

SSE2\_baseline = mean\_squared\_error(df.y, df.yhat\_baseline)\*len(df)

print("SSE manual == SSE sklearn: ", SSE == SSE2)

print("SSE manual - baseline == SSE sklearn - baseline: ", SSE\_baseline == SSE2\_baseline)

SSE manual == SSE sklearn: True

SSE manual - baseline == SSE sklearn - baseline: True

**MSE**

Now, instead of manually computing the MSE, we will use sklearn.metrics.mean\_squared\_error to compute it.

MSE2 = mean\_squared\_error(df.y, df.yhat)

MSE2\_baseline = mean\_squared\_error(df.y, df.yhat\_baseline)

print("MSE manual == MSE sklearn: ", MSE == MSE2)

print("MSE manual - baseline == MSE sklearn - baseline: ", MSE\_baseline == MSE2\_baseline)

MSE manual == MSE sklearn: True

MSE manual - baseline == MSE sklearn - baseline: True

**RMSE**

Now, instead of manually computing the RMSE, we will use sklearn.metrics.mean\_squared\_error to compute it.

We have to use mean\_squared\_error because there is not a function for RMSE. But that's easy because RMSE is only the square root of MSE.

RMSE2 = sqrt(mean\_squared\_error(df.y, df.yhat))

RMSE2\_baseline = sqrt(mean\_squared\_error(df.y, df.yhat\_baseline))

print("RMSE manual == RMSE skearn: ", RMSE == RMSE2)

print("RMSE manual - baseline == RMSE skearn - baseline: ", RMSE\_baseline == RMSE2\_baseline)

RMSE manual == RMSE skearn: True

RMSE manual - baseline == RMSE skearn - baseline: True

df\_eval = pd.DataFrame(np.array(['SSE','MSE','RMSE']), columns=['metric'])

df\_baseline\_eval = pd.DataFrame(np.array(['SSE\_baseline','MSE\_baseline','RMSE\_baseline']), columns=['metric'])

df\_eval['model\_error'] = np.array([SSE, MSE, RMSE])

df\_baseline\_eval['model\_error'] = np.array([SSE\_baseline, MSE\_baseline, RMSE\_baseline])

print(df\_eval)

print(df\_baseline\_eval)

metric model\_error

0 SSE 96.852596

1 MSE 13.836085

2 RMSE 3.719689

metric model\_error

0 SSE\_baseline 555.428571

1 MSE\_baseline 79.346939

2 RMSE\_baseline 8.907690

**Draw Conclusions**

Now, we will use our results to select the best model.

We will compare each baseline metric with the respective metrics of the linear regression model to see if our model performs better.

df\_eval['error\_delta'] = df\_eval.model\_error - df\_baseline\_eval.model\_error df\_eval

|  | **metric** | **model\_error** | **error\_delta** |
| --- | --- | --- | --- |
| 0 | SSE | 96.852596 | -458.575975 |
| 1 | MSE | 13.836085 | -65.510854 |
| 2 | RMSE | 3.719689 | -5.188001 |

yhat = df.yhat viz.evaluation\_example3(df, x, y, yhat)

So, it is clear here that the model performs much better than our baseline.

We can also answer that question and understand the strength of the relationship between the model and the dependent variable using the Coefficient of Determination, the F-test to test the value's significance, and the resulting p-value (probability that the relationship is due to chance)

**Evaluate Part 2: Model Significance**

$R^2$ tells you how well your model fits the data by measuring the strength of the relationship between your model and the dependent variable. However, it is not a formal test for the relationship. The F-test of overall significance is the hypothesis test for this relationship. It indicates whether your linear regression model provides a better fit to the data than a model that contains no independent variables.

If the overall F-test is significant (evident through the p-value of the F statistic), you can conclude that $R^2$ does not equal zero, and the correlation between the model and dependent variable is statistically significant.

**Manually**

First, we will manually compute the coefficient of determination, $R^2$ in order to understand how it is derived. Remember, $R^2$ is the ratio of the explained sum of squares (ESS) to the total sum of squares (TSS). When turned into a percentage, $R^2x100$, it represents the percent of variance in y (target) explained by x (feature).

In order to compute $R^2$ we need to compute the Explained Sum of Squares (ESS) and the Total Sum of Squares (TSS):

$$ R^2=\frac{ESS}{TSS} $$

**ESS**

The ESS (Explained Sum of Squares) is the sum of the difference between the each predicted value (predicted final grade, in this example) and the mean of all actual values (again, final grades).

$$ ESS = \sum^{n}\_{i=1} (\^y\_i - \bar{y})^2 $$

# compute explained sum of squares ESS = sum((df.yhat - df.y.mean())\*\*2)

**TSS**

The TSS (Total Sum of Squares) is the sum of difference between the actual final grade and the mean of all final grades. It can also be derived by summing the ESS and SSE.

$$ TSS = \sum ^n \_{i=1} (y\_i - \bar{y})^2 = ESS + SSE $$

SSE = df\_eval[df\_eval.metric=='SSE']['model\_error']

TSS = ESS + SSE

print("ESS = ", ESS)

print("SSE = ", SSE[0])

print("TSS = ", TSS[0])

ESS = 458.57597549177694

SSE = 96.85259593679456

TSS = 555.4285714285716

Looking back at the TSS, does that value look familiar? Where have you seen that number before?

Look at the definition of Total Sum of Squares and think about how you originally created your baseline model.

Let's compare those side-by-side...

# add the ESS, SSE, TSS from the model to a dataframe

ss = pd.DataFrame(np.array(['SSE','ESS','TSS']), columns=['metric'])

ss['model\_values'] = np.array([SSE[0], ESS, TSS[0]])

# compute and add the baseline values to the dataframe

ESS\_baseline = sum((df.yhat\_baseline - df.y.mean())\*\*2)

SSE\_baseline = df\_baseline\_eval[df\_baseline\_eval.metric=='SSE\_baseline']['model\_error'][0]

TSS\_baseline = ESS\_baseline + SSE\_baseline

`ss['model\_values'] = np.array([SSE\_baseline, ESS\_baseline, TSS\_baseline])

ss`

|  | **metric** | **model\_values** |
| --- | --- | --- |
| 0 | SSE | 555.428571 |
| 1 | ESS | 0.000000 |
| 2 | TSS | 555.428571 |

**R-Squared**

$R^2$ is the ratio of the Explained Sum of Squares (ESS) to the Total Sum of Squares (TSS):

$$ R^2 = \frac{ESS}{TSS} $$

R2 = ESS/TSS

print('R-squared = ',round(R2,3))

print("Percent of variance in y explained by x = ", round(R2\*100,1), "%")

R-squared = 0 0.826

Name: model\_error, dtype: float64

Percent of variance in y explained by x = 0 82.6

Name: model\_error, dtype: float64 %

**Scikit-Learn**

Now we will compute $R^2$ using sklearn.metrics.explained\_variance\_score, which, as your remember, the coefficient of determination, $R^2$ is essentially the explained variance by its basic definition.

# sklearn.metrics.explained\_variance\_score

evs = explained\_variance\_score(df.y, df.yhat)

print('Explained Variance = ', round(evs,3))

Explained Variance = 0.826

**Further Reading**

* + [Multiple Regression with statsmodels](https://nbviewer.jupyter.org/urls/s3.amazonaws.com/datarobotblog/notebooks/multiple_regression_in_python.ipynb)
  + Using [scipy.stats.linregress](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html) to quickly , r, p-value and standard error.

**Exercises**

Do you work for this exercise in either a jupyter notebook named evaluate within your regression-exercises repository. By the end of this exercise you will also create a python script named evaluate, so the overall deliverables for this exercise are the python script and jupyter notebook both with the name evaluate within your regression-exercises repo.

* + Load your zillow dataset.
  + Fit a linear regression model (ordinary least squares) and compute yhat, predictions of taxvaluedollarcnt using only calculatedfinishedsqft.

Here is some sample code to get you started:

from sklearn.linear\_model import LinearRegression

# assuming X and y are already defined

model = LinearRegression().fit(X, y)

predictions = model.predict(X)

Modify and add to the code above as necessary for it to work with the zillow dataset.

* + Plot the residuals for the linear regression model that you made.
  + Calculate the sum of squared errors, explained sum of squares, total sum of squares, mean squared error, and root mean squared error for your model.
  + Calculate the sum of squared errors, mean squared error, and root mean squared error for the baseline model (i.e. a model that always predicts the average taxvaluedollarcnt amount).
  + Write python code that compares the sum of squared errors for your model against the sum of squared errors for the baseline model and outputs whether or not your model performs better than the baseline model.
  + What is the amount of variance explained in your model?
  + Is your model better than the baseline model?
  + Create a file named evaluate.py that contains the following functions.
    - plot\_residuals(y, yhat): creates a residual plot
    - regression\_errors(y, yhat): returns the following values:
      * sum of squared errors (SSE)
      * explained sum of squares (ESS)
      * total sum of squares (TSS)
      * mean squared error (MSE)
      * root mean squared error (RMSE)
    - baseline\_mean\_errors(y): computes the SSE, MSE, and RMSE for the baseline model
    - better\_than\_baseline(y, yhat): returns true if your model performs better than the baseline, otherwise false

**Bonus Exercises**

* + Load the tips dataset from either pydataset or seaborn. Fit a linear regression model (ordinary least squares) and compute yhat, predictions of tip using total\_bill.
  + Load the mpg dataset and fit a model that predicts highway mileage based on engine displacement. Take a look at all the regression evaluation metrics, and determine whether this model is better than the baseline model. Use the functions from your evaluate.py to help accomplish this.
* [Modeling](https://ds.codeup.com/regression/model/)

**Regression**

Regression is a **supervised** machine learning technique for predicting a **continuous** target variable.

**Common Regression Algorithms**

Overview of the most common regression algorithms, along with their scikit-learn implementation.

**Ordinary Least Squares**

The linear regression algorithm we have all grown to know and love. In sklearn, the method that runs the OLS algorithm is [LinearRegression](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression).

sklearn.linear\_model.LinearRegression()

**LASSO + LARS**

Performs both feature selection and noise reduction to avoid overfitting (through Regularization11) to improve prediction performance and interpretability. Y should be normally distributed. The method to run the Lasso Lars combination in sklearn is [LassoLars](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LassoLars.html).

LASSO = Least Absolute Shrinkage and Selection Operator

LARS = Least Angle Regression

sklearn.linear\_model.LassoLars(alpha=1.0)

About alpha

* + $\alpha=0$ is basically the OLS algorithm.
  + $\alpha \in [0,\infin]$
  + Higher $\alpha$ will be more robust to collinearity between features.
  + As $\alpha$ increases, it will reach a point where performance in the model no longer changes. The fewer the features, the lower the value of $\alpha$ will be when you reach that point.
  + $\alpha = 1$ is a good starting point.
  + This is NOT the same $\alpha$ as the one representing critical value in statistical tests!

**Polynomial Regression**

Just like an ordinary linear model, but where the features are polynomial. So we create polynomial features first using the [PolynomialFeatures](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html#sklearn.preprocessing.PolynomialFeatures.fit_transform) method, and then fit a model using our new transformed features using any of the linear models, such as [LinearRegression](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression)

Steps:

* + Transform the features, with the hyperparameter degree representing the expected shape of the relationship between the features (X) and the target variable (y). It will usually be a 2 or 3, meaning a 2nd or 3rd degree polynomial. Values over that are likely to lead to overfitting.

sklearn.preprocessing.PolynomialFeatures(degree=2)

* + Fit a regression model using the transformed features.

sklearn.linear\_model.LinearRegression()

**Generalized Linear Model**

The most flexible regression algorithm, the GLM allows for different distributions, beyond just the Normal Distribution for OLS (and other models based on OLS, like LASSO). We will use the [TweedieRegressor](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.TweedieRegressor.html)22 method to create a GLM using sklearn.

sklearn.linear\_model.TweedieRegressor(power=0, alpha=1.0)

About power

* + power = 0: Normal Distribution
  + power = 1: Poisson Distribution
  + power = (1,2): Compound Distribution
  + power = 2: Gamma Distribution
  + power = 3: Inverse Gaussian Distribution

**TL;DR**

First of all, it always helps to know the distribution of your target variable before modeling.

plt.hist(y\_train) to see the shape of your target (y).

Secondly, you could try them all...doesn't hurt usually.

Third, if you have a good feel for your data:

* + For a normally distributed y and a linear relationship: OLS, LassoLars, GLM(power=0) will work best.
  + For polynomial relationships, polynomial regression is best.
  + For poisson, gamma or inverse gaussian distributions, use the Generalize Linear Model.

**Prepare Data for Modeling**

**Wrangle and Preprocess**

* + Acquired data from student-mat.csv.
  + Split data
  + Encode variables
  + Scale data

wrangle.wrangle\_sutdent\_math(path) returns the following:

| **Variable** | **Object Returned** | **Description** | **Purpose** |
| --- | --- | --- | --- |
| 1. df | dataframe | Original, complete dataframe | Acquire and prepare |
| 2. X\_train\_exp | dataframe | Features only, Unscaled | Exploration & analysis |
| 3. X\_train | dataframe | Features only, Scaled, Encoded categorical variables | Feature selection, fit models, make predictions |
| 4. y\_train | series | Target variable only, Unscaled | Feature selection, evaluate model predictions |
| 5. X\_validate | dataframe | Features only, Scaled, Encoded categorical variables | Make predictions using top models |
| 6. y\_validate | series | Target variable only, Unscaled | Evaluate model predictions made from X\_validate to assess overfitting |
| 7. X\_test | dataframe | Features only, Scaled, Encoded categorical variables | Make predictions using best model |
| 8. y\_test | series | Target variable only, Unscaled | Evaluate model predictions made from X\_test to estimate future performance on new data |

import pandas as pd

import numpy as np

import wrangle

import matplotlib.pyplot as plt

# modeling methods

from sklearn.metrics import mean\_squared\_error

from sklearn.linear\_model import LinearRegression, LassoLars, TweedieRegressor

from sklearn.preprocessing import PolynomialFeatures

import warnings

warnings.filterwarnings("ignore")

path='<https://gist.githubusercontent.com/ryanorsinger/\\>

55ccfd2f7820af169baea5aad3a9c60d/raw/da6c5a33307ed7ee207bd119d3361062a1d1c07e/student-mat.csv'

df, X\_train\_exp, X\_train, y\_train, \\

X\_validate, y\_validate, \\

X\_test, y\_test = wrangle.wrangle\_student\_math(path)

**Target Variable/y**

This helps us determine which type of algorithm we may want to use.

plt.hist(y\_train)

plt.xlabel("Final Grade (G3)")

plt.ylabel("Number of Students")

plt.show()

**Modeling**

**Baseline**

About the initial baseline:

Before we begin making models, we need to know how well we can estimate (predict) the final grade (G3) without using any features. This is often done by predicting every observation's target value to be the mean or the median. E.g. we could predict every student's final grade to be the mean final grade of all the students in our training sample. We will try both the mean and the median, see which performs best, and set that evaluation metric value as our baseline performance to beat.

* + Predict all final grades to be 10.52, which is equal to the mean of G3 for the training sample. Store in y\_train['G3\_pred\_mean'].
  + Predict all final grades to be 11, which is equal to the median of G3 for the training sample. Store in y\_train['G3\_pred\_median'].
  + Compute the RMSE comparing actual final grade (G3) to G3\_pred\_mean.
  + Compute the RMSE comparing actual final grade (G3) to G3\_pred\_median.

# We need y\_train and y\_validate to be dataframes to append the new columns with predicted values.

y\_train = pd.DataFrame(y\_train)

y\_validate = pd.DataFrame(y\_validate)

# 1. Predict G3\_pred\_mean

G3\_pred\_mean = y\_train['G3'].mean()

y\_train['G3\_pred\_mean'] = G3\_pred\_mean

y\_validate['G3\_pred\_mean'] = G3\_pred\_mean

# 2. compute G3\_pred\_median

G3\_pred\_median = y\_train['G3'].median()

y\_train['G3\_pred\_median'] = G3\_pred\_median

y\_validate['G3\_pred\_median'] = G3\_pred\_median

# 3. RMSE of G3\_pred\_mean

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_mean)\*\*(1/2)

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_mean)\*\*(1/2)

print("RMSE using Mean\\nTrain/In-Sample: ", round(rmse\_train, 2),

"\\nValidate/Out-of-Sample: ", round(rmse\_validate, 2))

# 4. RMSE of G3\_pred\_median

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_median)\*\*(1/2)

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_median)\*\*(1/2)

print("RMSE using Median\\nTrain/In-Sample: ", round(rmse\_train, 2),

"\\nValidate/Out-of-Sample: ", round(rmse\_validate, 2))

RMSE using Mean

Train/In-Sample: 4.5

Validate/Out-of-Sample: 4.58

RMSE using Median

Train/In-Sample: 4.52

Validate/Out-of-Sample: 4.69

# plot to visualize actual vs predicted.

plt.hist(y\_train.G3, color='blue', alpha=.5, label="Actual Final Grades")

plt.hist(y\_train.G3\_pred\_mean, bins=1, color='red', alpha=.5, rwidth=100, label="Predicted Final Grades - Mean")

plt.hist(y\_train.G3\_pred\_median, bins=1, color='orange', alpha=.5, rwidth=100, label="Predicted Final Grades - Median")

plt.xlabel("Final Grade (G3)")

plt.ylabel("Number of Students")

plt.legend()

plt.show()

**LinearRegression (OLS)**

* + Fit the model using X\_train\_scaled and the labels from y\_train.
  + Predict final grade for students in training sample using our model (lm).
  + Evaluate using RMSE
  + Repeat predictions and evaluation for validation.
  + Compare RMSE train vs. validation. Overfitting?

# create the model object

lm = LinearRegression()

# fit the model to our training data. We must specify the column in y\_train,

# since we have converted it to a dataframe from a series!

lm.fit(X\_train, y\_train.G3)

# predict train

y\_train['G3\_pred\_lm'] = lm.predict(X\_train)

# evaluate: rmse

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_lm)\*\*(1/2)

# predict validate

y\_validate['G3\_pred\_lm'] = lm.predict(X\_validate)

# evaluate: rmse

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_lm)\*\*(1/2)

print("RMSE for OLS using LinearRegression\\nTraining/In-Sample: ", rmse\_train,

"\\nValidation/Out-of-Sample: ", rmse\_validate)

RMSE for OLS using LinearRegression

Training/In-Sample: 1.6423518638932804

Validation/Out-of-Sample: 2.3167068610200023

**LassoLars**

* + Fit the model using X\_train\_scaled and the labels from y\_train.
  + Predict final grade for students in training sample using our model (lars).
  + Evaluate using RMSE
  + Repeat predictions and evaluation for validation.
  + Compare RMSE train vs. validation. Overfitting?

# create the model object

lars = LassoLars(alpha=1.0)

# fit the model to our training data. We must specify the column in y\_train,

# since we have converted it to a dataframe from a series!

lars.fit(X\_train, y\_train.G3)

# predict train

y\_train['G3\_pred\_lars'] = lars.predict(X\_train)

# evaluate: rmse

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_lars)\*\*(1/2)

# predict validate

y\_validate['G3\_pred\_lars'] = lars.predict(X\_validate)

# evaluate: rmse

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_lars)\*\*(1/2)

print("RMSE for Lasso + Lars\\nTraining/In-Sample: ", rmse\_train,

"\\nValidation/Out-of-Sample: ", rmse\_validate)

RMSE for Lasso + Lars

Training/In-Sample: 4.498925523895268

Validation/Out-of-Sample: 4.578916932633144

**TweedieRegressor (GLM)**

* + Fit the model using X\_train\_scaled and the labels from y\_train.
  + Predict final grade for students in training sample using our model (glm).
  + Evaluate using RMSE
  + Repeat predictions and evaluation for validation.
  + Compare RMSE train vs. validation. Overfitting?

# create the model object

glm = TweedieRegressor(power=1, alpha=0)

# fit the model to our training data. We must specify the column in y\_train,

# since we have converted it to a dataframe from a series!

glm.fit(X\_train, y\_train.G3)

# predict train

y\_train['G3\_pred\_glm'] = glm.predict(X\_train)

# evaluate: rmse

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_glm)\*\*(1/2)

# predict validate

y\_validate['G3\_pred\_glm'] = glm.predict(X\_validate)

# evaluate: rmse

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_glm)\*\*(1/2)

print("RMSE for GLM using Tweedie, power=1 & alpha=0\\nTraining/In-Sample: ", rmse\_train,

"\\nValidation/Out-of-Sample: ", rmse\_validate)

RMSE for GLM using Tweedie, power=1 & alpha=0

Training/In-Sample: 2.1576940833167058

Validation/Out-of-Sample: 2.5002918114896353

**Polynomial Regression**

Using sklearn.preprocessing.PolynommialFeatures() + sklearn.linear\_model.LinearRegression()

* + Create the new features, based on value indicated for degree for train, validate & test.
  + Fit the Linear Regression model
  + Predict using the transformed (squared or cubed, e.g.) features
  + Evaluate using RMSE
  + Repeat predictions and evaluation for validation.
  + Compare RMSE train vs. validation. Overfitting?

**PolynomialFeatures**

# make the polynomial features to get a new set of features

pf = PolynomialFeatures(degree=2)

# fit and transform X\_train\_scaled

X\_train\_degree2 = pf.fit\_transform(X\_train)

# transform X\_validate\_scaled & X\_test\_scaled

X\_validate\_degree2 = pf.transform(X\_validate)

X\_test\_degree2 = pf.transform(X\_test)

**LinearRegression**

# create the model object

lm2 = LinearRegression()

# fit the model to our training data. We must specify the column in y\_train,

# since we have converted it to a dataframe from a series!

lm2.fit(X\_train\_degree2, y\_train.G3)

# predict train

y\_train['G3\_pred\_lm2'] = lm2.predict(X\_train\_degree2)

# evaluate: rmse

rmse\_train = mean\_squared\_error(y\_train.G3, y\_train.G3\_pred\_lm2)\*\*(1/2)

# predict validate

y\_validate['G3\_pred\_lm2'] = lm2.predict(X\_validate\_degree2)

# evaluate: rmse

rmse\_validate = mean\_squared\_error(y\_validate.G3, y\_validate.G3\_pred\_lm2)\*\*(1/2)

print("RMSE for Polynomial Model, degrees=2\\nTraining/In-Sample: ", rmse\_train,

"\\nValidation/Out-of-Sample: ", rmse\_validate)

RMSE for Polynomial Model, degrees=2

Training/In-Sample: 2.0949158390423483e-14

Validation/Out-of-Sample: 3.016775612592487

**Evaluate**

**Plotting Actual vs. Predicted Values**

# y\_validate.head()

plt.figure(figsize=(16,8))

plt.plot(y\_validate.G3, y\_validate.G3\_pred\_mean, alpha=.5, color="gray", label='\_nolegend\_')

plt.annotate("Baseline: Predict Using Mean", (16, 9.5))

plt.plot(y\_validate.G3, y\_validate.G3, alpha=.5, color="blue", label='\_nolegend\_')

plt.annotate("The Ideal Line: Predicted = Actual", (.5, 3.5), rotation=15.5)

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_lm,

alpha=.5, color="red", s=100, label="Model: LinearRegression")

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_glm,

alpha=.5, color="yellow", s=100, label="Model: TweedieRegressor")

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_lm2,

alpha=.5, color="green", s=100, label="Model 2nd degree Polynomial")

plt.legend()

plt.xlabel("Actual Final Grade")

plt.ylabel("Predicted Final Grade")

plt.title("Where are predictions more extreme? More modest?")

# plt.annotate("The polynomial model appears to overreact to noise", (2.0, -10))

# plt.annotate("The OLS model (LinearRegression)\\n appears to be most consistent", (15.5, 3))

plt.show()

**Residual Plots: Plotting the Errors in Predictions**

# y\_validate.head()

plt.figure(figsize=(16,8))

plt.axhline(label="No Error")

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_lm-y\_validate.G3,

alpha=.5, color="red", s=100, label="Model: LinearRegression")

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_glm-y\_validate.G3,

alpha=.5, color="yellow", s=100, label="Model: TweedieRegressor")

plt.scatter(y\_validate.G3, y\_validate.G3\_pred\_lm2-y\_validate.G3,

alpha=.5, color="green", s=100, label="Model 2nd degree Polynomial")

plt.legend()

plt.xlabel("Actual Final Grade")

plt.ylabel("Residual/Error: Predicted Grade - Actual Grade")

plt.title("Do the size of errors change as the actual value changes?")

plt.annotate("The polynomial model appears to overreact to noise", (2.0, -10))

plt.annotate("The OLS model (LinearRegression)\\n appears to be most consistent", (15.5, 3))

plt.show()

**Histograms**

# plot to visualize actual vs predicted.

plt.figure(figsize=(16,8))

plt.hist(y\_validate.G3, color='blue', alpha=.5, label="Actual Final Grades")

plt.hist(y\_validate.G3\_pred\_lm, color='red', alpha=.5, label="Model: LinearRegression")

plt.hist(y\_validate.G3\_pred\_glm, color='yellow', alpha=.5, label="Model: TweedieRegressor")

plt.hist(y\_validate.G3\_pred\_lm2, color='green', alpha=.5, label="Model 2nd degree Polynomial")

plt.xlabel("Final Grade (G3)")

plt.ylabel("Number of Students")

plt.title("Comparing the Distribution of Actual Grades to Distributions of Predicted Grades for the Top Models")

plt.legend()

plt.show()

**Model Selection & Out-of-Sample Evaluation**

Model selected: lm (using LinearRegression)

y\_test = pd.DataFrame(y\_test)

# predict on test

y\_test['G3\_pred\_lm'] = lm.predict(X\_test)

# evaluate: rmse

rmse\_test = mean\_squared\_error(y\_test.G3, y\_test.G3\_pred\_lm)\*\*(1/2)

print("RMSE for OLS Model using LinearRegression\\nOut-of-Sample Performance: ", rmse\_test)

RMSE for OLS Model using LinearRegression

Out-of-Sample Performance: 2.078371913347772

**Exercises**

Do your work for this exercise in a jupyter notebook named modeling within the regression-exercises repo. Add, commit, and push your work.

* + Select a dataset with a continuous target variable.
  + Be sure your data is prepared (no missing values, numeric datatypes) and split into samples.
  + Work through **all** of the steps outlined in the lesson, from setting the baseline to selected a model and evaluating the final model on your test data.

**Notes**

11 Regularization = "Regularizations are techniques used to reduce the error by fitting a function appropriately on the given training set and avoid overfitting." [Towards Data Science](https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea)

22 TweedieRegressor: Requires sklearn v0.23 or greater installed. To update, run in terminal: conda install scikit-learn=0.23

* [Project](https://ds.codeup.com/regression/project/)

**Regression Project: Estimating Home Value**

**Scenario**

You are a junior data scientist on the Zillow data science team and receive the following email in your inbox:

We want to be able to predict the property tax assessed values ('taxvaluedollarcnt') of Single Family Properties that had a transaction during 2017.

We have a model already, but we are hoping your insights can help us improve it. I need recommendations on a way to make a better model. Maybe you will create a new feature out of existing ones that works better, try a non-linear regression algorithm, or try to create a different model for each county. Whatever you find that works (or doesn't work) will be useful. Given you have just joined our team, we are excited to see your outside perspective.

One last thing, Maggie lost the email that told us where these properties were located. Ugh, Maggie :-/. Because property taxes are assessed at the county level, we would like to know what states and counties these are located in.

* + - The Zillow Data Science Team

**Business Goals**

* + Construct an ML Regression model that predicts propery tax assessed values ('taxvaluedollarcnt') of **Single Family Properties** using attributes of the properties.
  + Find the key drivers of property value for single family properties. Some questions that come to mind are:
    - Why do some properties have a much higher value than others when they are located so close to each other?
    - Why are some properties valued so differently from others when they have nearly the same physical attributes but only differ in location?
    - Is having 1 bathroom worse for property value than having 2 bedrooms?
  + Deliver a report that the data science team can read through and replicate, understand what steps were taken, why and what the outcome was.
  + Make recommendations on what works or doesn't work in predicting these homes' values.

**Project Objectives**

* + Document code, process (*data acquistion, preparation, exploratory data analysis and statistical testing, modeling, and model evaluation*), findings, and key takeaways in a jupyter notebook final report.
  + Create modules ([acquire.py](http://acquire.py), [prepare.py](http://prepare.py)) that make your process repeateable and your report (notebook) easier to read and follow.
  + Ask exploratory questions of your data that will help you understand more about the attributes and drivers of home value. Answer questions through charts and statistical tests.
  + Construct a model to predict assessed home value for single family properties using regression techniques.
  + Make recommendations to a data science team about how to improve predictions.
  + Refine your work into a report, in the form of a jupyter notebook, that you will walk through in a 5 minute presentation to a group of collegues and managers. Communicate your goals, the work you did and why, what you found, your methodologies, and your conclusions.
  + Be prepared to answer panel questions about your code, process, findings, key takeaways, and model.

**Audience**

Your customer/end user is the **Zillow Data Science Team**. In your deliverables, be sure to re-state your goals, as if you were delivering this to Zillow. They have asked for something from you, and you are basically communicating in a more concise way, and very clearly, the goals as you understand them and how you have acted upon them through your research.

**Deliverables**

A. Github repo with:

* + a complete [readme.md](http://readme.md)
  + acquire module (.py)
  + prepare module (.py)
  + a final report (.ipynb)
  + other supplemental artifacts created while working on the project (e.g. exploratory/modeling notebook(s)).

B. Live 5 minute (max) presentation of your final notebook

**A. Github repo with the following:**

* + **Readme (.md)**
    - Project goals
    - Project description
    - Project planning (lay out your process through the data science pipeline)
    - Initial hypotheses and/or questions you have of the data, ideas
    - Data dictionary
    - Instructions or an explanation of how someone else can reproduce your project and findings (What would someone need to be able to recreate your project on their own?)
    - Key findings, recommendations, and takeaways from your project.
  + **Acquire & Prepare Modules (.py)**
    - Contains functions to acquire, prepare and split your data. You can have other .py files if you desire to abstract other code away from your final report.
    - Each of your functions are accompanied by descriptive docstrings. If they are functions you borrowed from instructors, put those docstrings in your own words.
    - Functions to acquire and prepare your data should be imported and used in your final report.
    - Your work must be reproducible by someone with their own env.py file.
  + **Final Report (.ipynb)**
    - A **Report** that has filtered out all the extraneous elements not necessary to include in the report.
    - Use markdown throughout the notebook to guide the audience. Assume the reader will not read your code blocks as you think about how much markdown guidance do you need.
    - Then, assume another reader will read ALL of your code, so make sure it is clearly commented. All cells with code need comments.
    - Your notebook should begin with a project overview and goals
    - Exploration should be refined in the report because now you know which visualizations and tests led to valuable outcomes.
    - Include at least 4 visualizations in the form of:
      * Question *in markdown* that you want to answer
      * Visualization
      * Statistical test (in at least 2 of your 4)
      * Provide your clear answer or takeaway *in markdown and natural language* to the question based on your exploration.
    - Provide the context of the target variable through a visualization (distribution of the values, e.g.)
    - Include your 3 best models in the final notebook to review. Show the steps and code you went through to fit the models, evaluate, and select.
    - On your best model, a chart visualizing how it performed on test would be valuable.
    - End with a conclusion that talks about your original goals and how you reached those (or didn't), the key findings, recommendations and next steps ("If I had more time, I would...")
  + Additional non-final notebooks (.ipynb) may be created while working on the project, containing exploration, other work, or modeling work, but they will not be graded. All required elements must be in the final report notebook.

**B. Live Presentation**

* + A **live presentation** where you deliver the **final report** (.ipynb) and walk through it with the audience.
  + You have a time limit of 5 minutes to present.
  + *If have content that you intend to skip in your presentation, it should not be included in your report, like scrolls and scrolls of visualizations.* Remember, this is a different artifact from the notebook you worked on that contains all your work. This serves a purpose of conveying information to others. And you will use it to give an overview of your project by walking through the main steps - what cleaning did you do and why, what insights did you find in exploration, what are 3 models you developed, how did the differ, and how did they compare in terms of performance? What was the best model and how do you expect it to perform in production on data it's never seen? And finally, wrap it all up in a conclusion. (5 minutes max)
  + You should be prepared to answer follow-up questions about your code, process, tests, model, and findings.
  + You will have created multiple notebooks in your work. Do not be concerned about not showing all your work in your report. That is not intended. We can look back to see the work that led to your final notebook. But we want you to feel comfortable creating a report from your work that gives brief insight into the findings and how you got those findings. This is the first step in learning to deliver a report that is abstracted away from all the details. It takes practice to feel comfortable not showing everything. It will get easier the more you practice.

**Project Guidance**

* + Read the rubric for the project before beginning.
  + Reread your feedback/rubric from the classification project and any notes you took from feedback.
  + You are asked to use **properties that had a transaction** in 2017! You must figure out how to determine which properties those are and filter your data IN SQL before bringing it into your Python environment.
    - You will need to use the **properties\_2017**, **predictions\_2017**, and **propertylandusetype** tables.
  + For the first iteration of your model, use only square feet of the home, number of bedrooms, and number of bathrooms to estimate the property's assessed value, taxvaluedollarcnt. You can expand this to other fields after you have completed an MVP (minimally viable product).
  + Be sure and remove the fields that leak information about taxvaluedollarcnt. These are fields we would not know until we knew the assessed value, so using them would be "cheating". These fields are landtaxvaluedollarcnt, structuretaxvaluedollarcnt, and taxamount.
  + You will want to do some data validation or QA (quality assurance) to be sure the data you gather is what you think it is.
  + You will want to make sure you are using the best fields to represent square feet of home, number of bedrooms, and number of bathrooms. "Best" meaning the most accurate and available information. Here you will need to do some data investigation in the database and use your domain expertise to make some judgement calls.
  + You will want to read and re-read the requirements given by your stakeholders to be sure you are meeting all of their needs and representing it in your data, report, and model.

**Data Science Pipeline Guidance**

We highly recommend you re-read the data science pipeline lesson to refresh your memory on the purpose of each stage in the pipeline and how to get there.

<https://ds.codeup.com/fundamentals/data-science-pipeline/>