Contents

Course 1: Intro Machine Learning

How Models Work

- Models identify previous patterns to make predictions on new data
- Simple Model Example: Decision Tree
    - Divides patterns by n # categories
    - Capture more factors with more "splits" or branches of branches
        - Adding splits = deeper trees
    - The category chain is called the "Path" and the bottom conclusion node is called the "Leaf"

Basic Data Exploration

- Use a library like Pandas to explore and manipulate data
- Pandas uses "dataframes" to store the data which are like SQL tables or excel sheets. Use .describe() to get a rough overview
    - The columns are the data categories in the dataset
    - The rows are the statistical measurements (std, mean, 25%, etc.)
- Coding Exercise Complete!

Your First Machine Learning Model

- Problem: Too many variables → Pick from intuition (IRL want to automate this)
- Pandas Dataframe Operations
    - Use .columns to identify the columns
    - Drop missing values with .dropna(axis=0)
    - Select variables with dot notation; ex. Y = melbourne_data.price
    - Use .head() to preview the first few rows in the actual dataframe
- Another name for inputted columns is "features"; how do we pick them?
    - Sometimes all columns will be features while other times you only want to select a few
    - X = features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']
- The Steps to Building a Model
    - Define: What type of model will it be?
    - Fit: Capture patterns.
    - Predict
    - Evaluate: How accurate are the predictions?
- Using Scikit Learn
    - Scikit-learn is a library with pretrained models; just import
    - Ex. my_model = DecisionTreeRegressor(random_state=1)
        - Random state is a parameter; it ensures replicable results
        - Some models require parameter(s) as well
    - Fit: my_model.fit(X, y)
        - X are the features
        - Y is the target
    - Predict: my_model.predict(X)
    - First Machine Learning Model Exercise Complete!

Model Validation

- Model Validation is how well your model predicts what actually happens
    - One such metric is the Mean Absolute Error (MAE)
        - error = actual - predicted
        - MAE = avg(|error|)
- Validation Data
    - The value of a prediction lies in how well it can predict on new data = no point predicting on training data
    - If so, can find false patterns
    - Validation Data! → Set aside a portion of the data to validate on so it is "new"
        - This can be done automatically through scikit-learn
        - Train_x, val_x, train_y, val_y = train_test_split(X, y, random_state=0)
            - Make the model and predict after with these new variables
- Exercise Complete!

Underfitting and Overfitting

- Other models exist, see scikit-learn's documentation page for more than you need
- Overfitting is when we divide the decision tree too much
    - This fits really well for sample data but…
    - Miscategorizes for new data = inaccurate
- Underfitting is when we divide the tree too little
    - Ex. Housing price factors are not just whether a house has a garage or not
    - Doesn't capture all the trends aka "full story"
- There is a "sweet spot" for tree depth that takes trial and error
    - MAE is one such measurement
    - Other Common strategies include looping through different depths
- After finding the best depth, specify the best max_leaf_nodes for the model and fit the model on ALL of the data for best accuracy!
- Exercise Complete!

Random Forests

- Alternative model
- Addresses the tension be overfitting and underfitting for decision trees
- Uses many trees and makes a prediction by averaging the predictions of each component tree
- Exercise Complete!

Course 2: Intermediate Machine Learning

Exercise submitted!

Missing Values

- This is very common, ex. 2 bedroom house won't have a value for a third bedroom's price
- Option 1: Drop the problematic column
  - Problem: Loses important information!
  - ```
    # Get names of columns with missing values
    cols_with_missing = [col for col in X_train.columns
                         if X_train[col].isnull().any()]

    # Drop columns in training and validation data
    reduced_X_train = X_train.drop(cols_with_missing, axis=1)
    reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)
    ```
- Option 2: Imputation (Fill in missing values with some number)
  - Ex. mean value in each column becomes the fill value
  - Better than dropping columns
  - Problem: Won't be exactly right and predicting based on estimated values
  - ```
    from sklearn.impute import SimpleImputer

    # Fill in the lines below: imputation
    imputer = SimpleImputer() # Your code here
    imputed_X_train = pd.DataFrame(imputer.fit_transform(X_train))
    imputed_X_valid = pd.DataFrame(imputer.transform(X_valid))

    # Fill in the lines below: imputation removed column names; put them back
    imputed_X_train.columns = X_train.columns
    imputed_X_valid.columns = X_valid.columns
    ```
  - With the imported imputer, you need to add back the column names
- Option 3: An Extension to Imputation
  - Add an additional column saying with a boolean True for rows that were imputed
- *Note: Drop the same columns in your training and validation sets!
- Exercise Complete!

Categorical Variables

- Variables that don't have specific values, i.e. limited # inputs
- Problem: The nature of these variables creates errors in most models
- Solution: Encoding
  - Option 1: Drop Categorical Variables
    - Only works if the columns contained irrelevant info

```
# Fill in the lines below: drop columns in training and validation data
s = (X_train.dtypes == 'object')
print(s)
object_cols = list(s[s].index)
drop_X_train = X_train.drop(object_cols, axis=1)
drop_X_valid = X_valid.drop(object_cols, axis=1)
```

- Option 2: Ordinal Encoding
  - If the variables have a clear order (ordinal), assign a number to each category
  - Convert the categories to numbers
  - The encoder will throw an error if the training data has values that aren't in the validation data, be careful!

```
from sklearn.preprocessing import OrdinalEncoder

# Drop categorical columns that will not be encoded
label_X_train = X_train.drop(bad_label_cols, axis=1)
label_X_valid = X_valid.drop(bad_label_cols, axis=1)

# Apply ordinal encoder
ordinal_encoder = OrdinalEncoder() # Your code here
label_X_train[good_label_cols] = ordinal_encoder.fit_transform(X_train[good_label_cols])
label_X_valid[good_label_cols] = ordinal_encoder.transform(X_valid[good_label_cols])
```

- Option 3: One-hot Encoding
  - Creates new columns in the presence/absence of variables to convey info
  - Ex. Column values are "red, green, blue" → red column

```
# Apply one-hot encoder to each column with categorical data
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[low_cardinality_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[low_cardinality_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

# Ensure all columns have string type
OH_X_train.columns = OH_X_train.columns.astype(str)
OH_X_valid.columns = OH_X_valid.columns.astype(str)
```

    - Train the one-hot encoded columns on low cardinality object columns
    - Add back in the index

        - Remove the categorical columns in the datasets (to be replaced with OneHotCoding columns)
        - Rejoin the numerical and OneHotCoding (instead of categorical) columns
        - Make all columns have a string type so they qualify as feature names (needed to do stuff like calculate MAE)
- (Optional) Submitting to Kaggle Competitions

Pipelines

- What it is: A Bundle that combines the preprocessing and modeling steps
- What it does: Keeps data preprocessing code and modeling organized
- Benefits:
    - Cleaner Code: no need to account for data at each preprocessing step
    - Fewer Bugs: less chance to miss/fumble a step
    - Streamlined Production
    - More Model Validation options
- Constructing a Pipeline
    - Step 1: Define Preprocessing steps
        - Ex. Use the "Column Transformer" to bundle the preprocessing packages

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

        -
        - You can change the "strategy" parameters for better results!
    - Step 2: Define the model
        - Ex. Use Random Forest

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)
```

- Step 3: Create and evaluate Pipeline

```
from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model)
                             ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

- Exercise complete!

Cross-Validation

- Noise = Randomness from model predictions due to variations in validation set selection
- More validation data = Less Noise = Good = Less training Data = weaker models = Bad → Problem!
- Solution: Run model on different subsets of data → Cross-Validation!
  - Divide data into sections (aka "folds)
  - Each fold rotates being the "holdout fold"; the fold that isn't used in the training set
  - The model thus has each fold as the validation set once and we've used 100% of the folds even if not simultaneously
- When to use?
  - More accurate estimation of model quality which is important for modeling decisions, however, it takes a lot of time to run
  - Good for small datasets (when cross-validation isn't such a big burden) bad for large datasets; single validation in that case is sufficient
- How
  - Preprocess the data first and set up the model (with Pipeline)

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                              ('model', RandomForestRegressor(n_estimators=50,
                                                              random_state=0))
                             ])
```

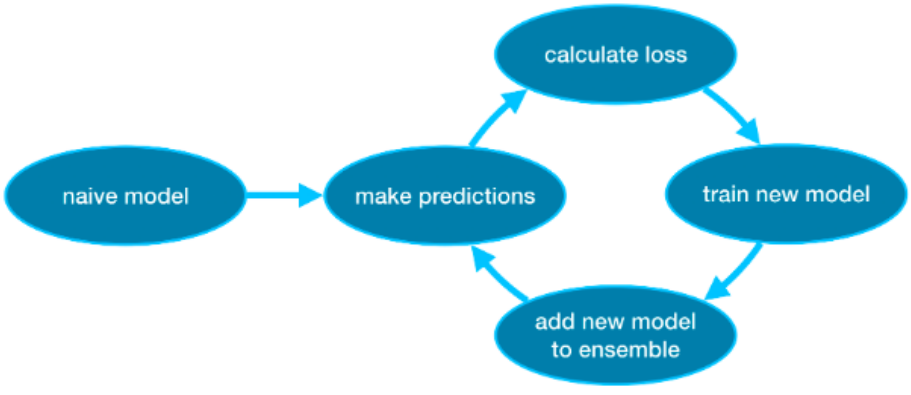
    - Import cross-val score from scikit-learn to calculate and perform cross-validation
        - Multiply scores by -1 since SKLearn calculates negative MAE
        - List the # folds you desire in parameter

```
from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates *negative* MAE
scores = -1 * cross_val_score(my_pipeline, X, y,
                              cv=5,
                              scoring='neg_mean_absolute_error')
```

- Exercise Complete!

<u>XGBoost</u>

- A model that uses gradient boosting; These models achieve the best results usually
    - XGBoost = “Extreme gradient boosting”
- An ensemble (combination) method like RF
- What:
    - Train one model (with very naive predictions)
    - Use the predictions to calculate a loss function (like mean squared error)
    - Use loss function to fit a new model to add to the ensemble (the parameters are determined based on the previous prediction results
        - The gradient is short for “gradient descent” which is what we do here
    - Add the new model to the ensemble and repeat



- How:
    - Import SKLearn API

```
from xgboost import XGBRegressor

my_model = XGBRegressor()
my_model.fit(X_train, y_train)
```

- Make Predictions and evaluate model like other SKLearn models

```
from sklearn.metrics import mean_absolute_error

predictions = my_model.predict(X_valid)
print("Mean Absolute Error: " + str(mean_absolute_error(predictions, y_valid)))
```

- Parameter Tuning !!!
  - N_estimators = how many times to cycle
    - Too low = underfitting
    - Too high = overfitting
    - Typical values range from 100 to 1000

```
my_model = XGBRegressor(n_estimators=500)
my_model.fit(X_train, y_train)
```

  - Early_stopping_rounds
    - Causes model to stop iterating when validation scores stop improving
    - Way to automatically find # estimators
    - Specify # rounds of straight deterioration before stopping (to prevent it from stopping by chance); 5 good #
    - Eval_set = data set aside to calculate validation scores

```
my_model = XGBRegressor(n_estimators=500)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

  - Learning_rate
    - A number multiplied on the predictions of each model to scale the weight of each model down
    - The alternative is directly adding the models to the ensemble
    - Default value is 0.1

```
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

  - N_jobs
    - # jobs to run in parallel

- Good for large datasets but small datasets useless
- Set to # of cores in PC
-

```
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=4)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

- Exercise Complete!

Data Leakage

- Occurs when information exists in your training datasets but not in the real dataset resulting in inaccurate predictions
- Two types: Target Leakage and Train-Test contamination
- Target Leakage
  - Occurs when there's data that won't be available at the time of your predictions
  - Ex. "Taken Antibiotics" and "Pneumonia", Can be some ppl with disease who haven't taken antibiotics when making prediction, model not trained on that
  - Solution: Drop any variables created/edited after target variable at prediction time
- Train-Test Contamination
  - When your validation dataset is trained on or corrupted in some way such that it's no longer an accurate measure of the model's performance after deployment
    - Ex. Use of preprocessing techniques like imputers on both the training and validation data before calling "train_test_split"
- One way to detect leakage:
  - Display + Analyze each variable description
  - Which variables are ambiguous and/or have data after the prediction?
    - Ex. Credit Card Applications → "Expenditure" might mean expenditure on card after getting accepted = bad
    - Ex. Housing Prices → "Average Neighborhood Price" might be leakage if the current house's price is included in the average
    - Ex. Disease Prediction → "Surgeon Infection Rate" Might be train-test contamination if the current infection rate for the surgery is factored into the average
- Exercise Complete!

Course 3: Intro Programming

Notes to Self:

- Trivial but I did the exercises for practice & to make sure I didn't miss anything
- Familiarized myself with Kaggle submissions and did some ML models from scratch on established data like the Titanic
- Brushed up on python; Diff data types can be multiplied by booleans! True=1, False=0

Practice questions I thought were worth reviewing are listed below for reference:

- Arithmetic + Variables

[ ]:

```
# Load the data from the titanic competition
import pandas as pd
titanic_data = pd.read_csv("../input/titanic/train.csv")

# Show the first five rows of the data
titanic_data.head()
```

The data has a different row for each passenger.

The next code cell defines and prints the values of three variables:

- `total` = total number of passengers who boarded the ship
- `survived` = number of passengers who survived the shipwreck
- `minors` = number of passengers under 18 years of age

Run the code cell without changes. (Don't worry about the details of how these variables are calculated for now. You can learn more about how to calculate these values in the Pandas course.)

+ Code + Markdown

[ ]:

```
# Number of total passengers
total = len(titanic_data)
print(total)

# Number of passengers who survived
survived = (titanic_data.Survived == 1).sum()
print(survived)

# Number of passengers under 18
minors = (titanic_data.Age < 18).sum()
print(minors)
```

- Functions

Use the next code cell to define the function `get_actual_cost()`. You'll need to use the `math.ceil()` function to do this.

When answering this question, note that it's completely valid to define a function that makes use of another function. For instance, we can define a function `round_up_and_divide_by_three` that makes use of the `math.ceil` function:

```
def round_up_and_divide_by_three(num):
    new_value = math.ceil(num)
    final_value = new_value / 3
    return final_value
```

+ Code + Markdown

```
def get_actual_cost(sqft_walls, sqft_ceiling, sqft_per_gallon, cost_per_gallon):
    cost = math.ceil((sqft_walls + sqft_ceiling)/sqft_per_gallon) * cost_per_gallon
    return cost

# Check your answer
q5.check()
```

Correct

- Data Types

[5]:
```
# Uncomment and run this code to get started!
print(3 * True)
print(-3.1 * True)
print(type("abc" * False))
print(len("abc" * False))
```

```
3
-3.1
<class 'str'>
0
```

🌶️ Question 5

You own an online shop where you sell rings with custom engravings. You offer both gold plated and solid gold rings.

- Gold plated rings have a base cost of $50, and you charge $7 per engraved unit.
- Solid gold rings have a base cost of $100, and you charge $10 per engraved unit.
- Spaces and punctuation are counted as engraved units.

Write a function `cost_of_project()` that takes two arguments:

- `engraving` - a Python string with the text of the engraving
- `solid_gold` - a Boolean that indicates whether the ring is solid gold

It should return the cost of the project. This question should be fairly challenging, and you may need a hint.

]:
```
def cost_of_project(engraving, solid_gold):
    cost = 50 * (1+solid_gold) + (7+solid_gold*3)*(len(engraving))
    return cost

# Check your answer
q5.check()
```

Correct

- Conditionals

In this question, you'll work with a function `get_labels()` that takes the nutritional details about a food item and prints the needed warning labels. This function takes several inputs:

- `food_type` = one of `"solid"` or `"liquid"`
- `serving_size` = size of one serving (if solid, in grams; if liquid, in milliliters)
- `calories_per_serving` = calories in one serving
- `saturated_fat_g` = grams of saturated fat in one serving
- `trans_fat_g` = grams of trans fat in one serving
- `sodium_mg` = mg of sodium in one serving
- `sugars_g` = grams of sugar in one serving

Note that some of the code here should feel unfamiliar to you, since we have not shared the details of how some of the functions like `excess_sugar()` or `excess_saturated_fat()` work. But at a high level, these are functions that return a value of `True` if the food is deemed to have an excess of sugar or saturated fat, respectively. These functions are used within the `get_labels()` function, and whenever there is an excess (of sugar or saturated fat, but also of trans fat, sodium, or calories), it prints the corresponding label.

```
# import functions needed to make get_labels work
from learntools.intro_to_programming.ex4q5 import excess_sugar, excess_saturated_fat, excess_trans_fat, excess_sodium, excess_cal

def get_labels(food_type, serving_size, calories_per_serving, saturated_fat_g, trans_fat_g, sodium_mg, sugars_g):
    # Print messages based on findings
    if excess_sugar(sugars_g, calories_per_serving) == True:
        print("EXCESO AZÚCARES / EXCESS SUGAR")
    if excess_saturated_fat(saturated_fat_g, calories_per_serving) == True:
        print("EXCESO GRASAS SATURADAS / EXCESS SATURATED FAT")
    if excess_trans_fat(trans_fat_g, calories_per_serving) == True:
        print("EXCESO GRASAS TRANS / EXCESS TRANS FAT")
    if excess_sodium(calories_per_serving, sodium_mg) == True:
        print("EXCESO SODIO / EXCESS SODIUM")
    if excess_calories(food_type, calories_per_serving, serving_size) == True:
        print("EXCESO CALORÍAS / EXCESS CALORIES")
```

- Lists

🌶️ Question 5

Say you're doing analytics for a website. You need to write a function that returns the percentage growth in the total number of users relative to a specified number of years ago.

Your function `percentage_growth()` should take two arguments as input:

- `num_users` = Python list with the total number of users each year. So `num_users[0]` is the total number of users in the first year, `num_users[1]` is the total number of users in the second year, and so on. The final entry in the list gives the total number of users in the most recently completed year.
- `yrs_ago` = number of years to go back in time when calculating the growth percentage

For instance, say `num_users = [920344, 1043553, 1204334, 1458996, 1503323, 1593432, 1623463, 1843064, 1930992, 2001078]`.

- if `yrs_ago = 1`, we want the function to return a value of about `0.036`. This corresponds to a percentage growth of approximately 3.6%, calculated as (2001078 - 1930992)/1930992.
- if `years_ago = 7`, we would want to return approximately `0.66`. This corresponds to a percentage growth of approximately 66%, calculated as (2001078 - 1204334)/1204334.

Your coworker sent you a draft of a function, but it doesn't seem to be doing the correct calculation. Can you figure out what has gone wrong and make the needed changes?

```
# TODO: Edit the function
def percentage_growth(num_users, yrs_ago):
    growth = (num_users[len(num_users)-1] - num_users[len(num_users)-yrs_ago-1])/num_users[len(num_users)-yrs_ago-1]
    return growth

# Do not change: Variable for calculating some test examples
num_users_test = [920344, 1043553, 1204334, 1458996, 1503323, 1593432, 1623463, 1843064, 1930992, 2001078]

# Do not change: Should return .036
print(percentage_growth(num_users_test, 1))

# Do not change: Should return 0.66
print(percentage_growth(num_users_test, 7))
```

Course 4: Python

Notes to self:

- Did all the exercises/brain teasers
- Refreshed knowledge on optional args
- Improved list comprehension
- Improved mastery of strings and formatting
- Got tripped up on variable clobbering (same variable in loop was also used to reference dict)

Exercises worth review:

6.🌶️

We've seen that calling `bool()` on an integer returns `False` if it's equal to 0 and `True` otherwise. What happens if we call `int()` on a bool? Try it out in the notebook cell below.

Can you take advantage of this to write a succinct function that corresponds to the English sentence "does the customer want exactly one topping?"?

```
def exactly_one_topping(ketchup, mustard, onion):
    """Return whether the customer wants exactly one of the three available toppings
    on their hot dog.
    """
    return (int(ketchup) + int(mustard) + int(onion)) == 1
    pass

# Check your answer
q6.check()
```

Correct:

This condition would be pretty complicated to express using just `and`, `or` and `not`, but using boolean-to-integer conversion gives us this short solution:

```
return (int(ketchup) + int(mustard) + int(onion)) == 1
```

Fun fact: we don't technically need to call `int` on the arguments. Just by doing addition with booleans, Python implicitly does the integer conversion. So we could also write...

```
return (ketchup + mustard + onion) == 1
```

5.🌶️

We're using lists to record people who attended our party and what order they arrived in. For example, the following list represents a party with 7 guests, in which Adela showed up first and Ford was the last to arrive:

```
party_attendees = ['Adela', 'Fleda', 'Owen', 'May', 'Mona', 'Gilbert', 'Ford']
```

A guest is considered 'fashionably late' if they arrived after at least half of the party's guests. However, they must not be the very last guest (that's taking it too far). In the above example, Mona and Gilbert are the only guests who were fashionably late.

Complete the function below which takes a list of party attendees as well as a person, and tells us whether that person is fashionably late.

```
def fashionably_late(arrivals, name):
    """Given an ordered list of arrivals to the party and a name, return whether the guest with that
    name was fashionably late.
    """
    index = arrivals.index(name)
    if (index >= len(arrivals)/2) and (index < len(arrivals)-1):
        return True
    return False
    pass

# Check your answer
q5.check()
```

Correct

2.

A researcher has gathered thousands of news articles. But she wants to focus her attention on articles including a specific word. Complete the function below to help her filter her list of articles.

Your function should meet the following criteria:

- Do not include documents where the keyword string shows up only as a part of a larger word. For example, if she were looking for the keyword “closed”, you would not include the string “enclosed.”
- She does not want you to distinguish upper case from lower case letters. So the phrase “Closed the case.” would be included when the keyword is “closed”
- Do not let periods or commas affect what is matched. “It is closed.” would be included when the keyword is “closed”. But you can assume there are no other types of punctuation.

```
def word_search(doc_list, keyword):
    """
    Takes a list of documents (each document is a string) and a keyword.
    Returns list of the index values into the original list for all documents
    containing the keyword.

    Example:
    doc_list = ["The Learn Python Challenge Casino.", "They bought a car", "Casinoville"]
    >>> word_search(doc_list, 'casino')
    >>> [0]
    """
    indices = []
    wordList = ''
    for i in range(len(doc_list)):
        wordList = doc_list[i].lower().split(' ')
        for word in wordList: #because of punctuation
            if keyword in word and len(word)-len(keyword)<=1 and i not in indices:
                indices.append(i)
    return indices

# Check your answer
q2.check()
```

3.

Now the researcher wants to supply multiple keywords to search for. Complete the function below to help her.

(You're encouraged to use the `word_search` function you just wrote when implementing this function. Reusing code in this way makes your programs more robust and readable - and it saves typing!)

```
def multi_word_search(doc_list, keywords):
    """
    Takes list of documents (each document is a string) and a list of keywords.
    Returns a dictionary where each key is a keyword, and the value is a list of indices
    (from doc_list) of the documents containing that keyword

    >>> doc_list = ["The Learn Python Challenge Casino.", "They bought a car and a casino", "Casinoville"]
    >>> keywords = ['casino', 'they']
    >>> multi_word_search(doc_list, keywords)
    {'casino': [0, 1], 'they': [1]}
    """
    toRet = dict()
    for key in keywords:
        toRet[key] = word_search(doc_list, key)
    return toRet
    pass

# Check your answer
q3.check()
```

Correct

Luigi is trying to perform an analysis to determine the best items for winning races on the Mario Kart circuit. He has some data in the form of lists of dictionaries that look like...

```
[
    {'name': 'Peach', 'items': ['green shell', 'banana', 'green shell',], 'finish': 3},
    {'name': 'Bowser', 'items': ['green shell',], 'finish': 1},
    # Sometimes the racer's name wasn't recorded
    {'name': None, 'items': ['mushroom',], 'finish': 2},
    {'name': 'Toad', 'items': ['green shell', 'mushroom'], 'finish': 1},
]
```

`'items'` is a list of all the power-up items the racer picked up in that race, and `'finish'` was their placement in the race (1 for first place, 3 for third, etc.).

He wrote the function below to take a list like this and return a dictionary mapping each item to how many times it was picked up by first-place finishers.

[8]:

```python
def best_items(racers):
    """Given a list of racer dictionaries, return a dictionary mapping items to the number
    of times those items were picked up by racers who finished in first place.
    """
    winner_item_counts = {}
    for i in range(len(racers)):
        # The i'th racer dictionary
        racer = racers[i]
        # We're only interested in racers who finished in first
        if racer['finish'] == 1:
            for i in racer['items']:
                # Add one to the count for this item (adding it to the dict if necessary)
                if i not in winner_item_counts:
                    winner_item_counts[i] = 0
                winner_item_counts[i] += 1

        # Data quality issues :/ Print a warning about racers with no name set. We'll take care of it later.
        if racer['name'] is None:
            print("WARNING: Encountered racer with unknown name on iteration {}/{} (racer = {})".format(
                i+1, len(racers), racer['name'])
                 )
    return winner_item_counts
```

Course 5: Pandas

What is Pandas/How to Setup

- Pandas is a library, import it (import pandas)
- Pandas stores data in a structure called a dataframe which looks like this:
  - pd.DataFrame({'Yes': [50,21], 'No': [131,2]})
  -

	Yes	No
0	50	131
1	21	2

  - The "0, Yes" entry has a value of 50
  - Can also index the entries (instead of 0 and 1) with an extra parameter
- Panda Series; Dataframe : table :: Series : list

```
pd.Series([1, 2, 3, 4, 5])
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

  - A series is 1 column in a dataframe → can assign indices to each value in series
  - Other helpful parameters include "name" and "dtype"
- Preformatted file types ex. CSV "Comma Separated File"
  - Pd.read_csv to read the file into a dataframe
  - Df.shape = (#entries, #columns)
  - Writing to csv: myDf.to_csv("filepath.csv")

Indexing/Selecting/Assigning

- Access column via array OR dot notation
  - Reviews.country = reviews['country']
- Series = fancy dictionary
  - Reviews['country'][0] = first country in the reviews dataframe
- Pandas custom accessors (recommended for more complex operations)
  - iloc (Index based selection); first row in df = reviews.iloc[0]
    - First col in df = reviews.iloc[:, 0]
  - loc (label based selection); the data index value not its position matters
    - First item: reviews.loc[0, 'country']

```
reviews.loc[:, ['taster_name', 'taster_twitter_handle', 'points']]
```

	taster_name	taster_twitter_handle	points
0	Kerin O'Keefe	@kerinokeefe	87
1	Roger Voss	@vossroger	87
...	...	...	...
129969	Roger Voss	@vossroger	90
129970	Roger Voss	@vossroger	90

129971 rows × 3 columns

        -
        - Last index in range ex. [0:1000] is inclusive!, will return 1001 values
    - Row first Column Second, opposite of Python norm!
- Check documentation for modifications, many attributes are not immutable!
    - Ex. set_index makes one column the new index column
- Conditional selection
    - Reviews.loc[reviews.country == 'Italy']
    - Can combine: Reviews.loc[(reviews.country == 'Italy') && (reviews.points >= 90)]; use '|' for OR
    - .isin checks if a value is in a specific column
    - Isnull and notnull
- Assign values
    - Self explanatory: reviews['critic'] = 'everyone
    - Iterable: reviews['index_backwards'] = range(len(reviews), 0, -1)
- Exercise Complete!

<u>Summary Functions</u>

- describe() → summarizes key stat metrics of the data like max, min, and mean
    - Only makes sense for numerical data
- mean()
- unique(): lists unique values in the series
- value_counts(): lists unique values AND # times they occur
- Exercise Complete!

<u>Maps</u>

- A function that maps one set of values to another set of values
- map()
    - Parameter: Function that returns a single value from a series
        - Transforms each point in the series and returns the transformed series

```
review_points_mean = reviews.points.mean()
reviews.points.map(lambda p: p - review_points_mean)
```

    -
- apply()
    - Transforms whole dataframe by row

- Calls custom method
  - ```
    def remean_points(row):
        row.points = row.points - review_points_mean
        return row

    reviews.apply(remean_points, axis='columns')
    ```
- Pandas can auto map to series as well
  - Works with an operation between a series and a single value
  - OR series of equal length
- Exercise Complete!

Grouping

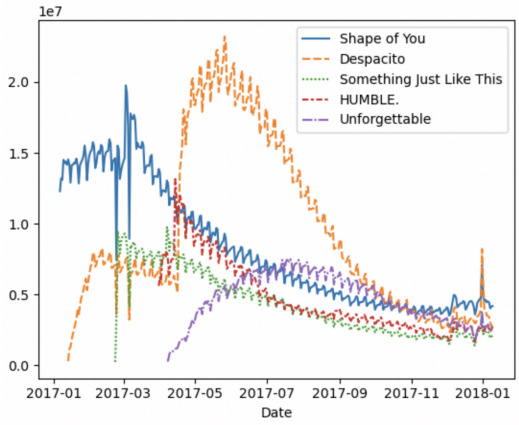
- Use case: sometimes want to group items not by column/row but by a set of points with something in common
- groupby(), groups dataframe entries by the criteria specified by parameter
  - I.e. long way to do value_counts() can be done by: reviews.groupby("points").points.count()
- A slice of the DataFrame containing only values that match
- Can group by more than one column for more control
  - Ex. best wine by country AND province
  - ```
    reviews.groupby(['country', 'province']).apply(lambda df: df.loc[df.points.idxmax()])
    ```
- agg(); a method to combine multiple functions to perform on the group
  - ```
    reviews.groupby(['country']).price.agg([len, min, max])
    ```
  - Does the length, min, and max function on the 'country' group
- Multi-indexes
  - Tiered set of indices created when using groupby() on multiple groups
  - Require multiple levels of indexing to access elements
  - reset_index() reverts to original single indexing form before the grouping
- Exercise Complete!

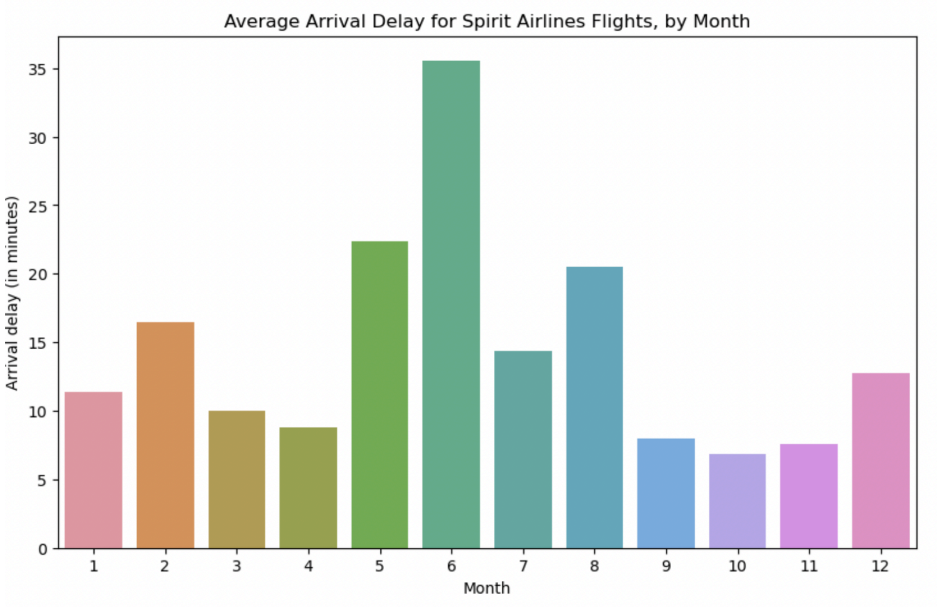
Sorting

- Problem: using group_by orders the values by index, what if we want values ordered another way? → Solution: sort ourselves
- sort_values()
  - Parameter is sorting criteria, ex. (by='len) = sort by length
  - Defaults to ascending; change by setting to false (ascending=False)
- To sort by index use: sort_index()
- Can also sort by multiple columns at a time
  - Ex. sort_values(by=['country', 'len'])
- Exercise Complete!

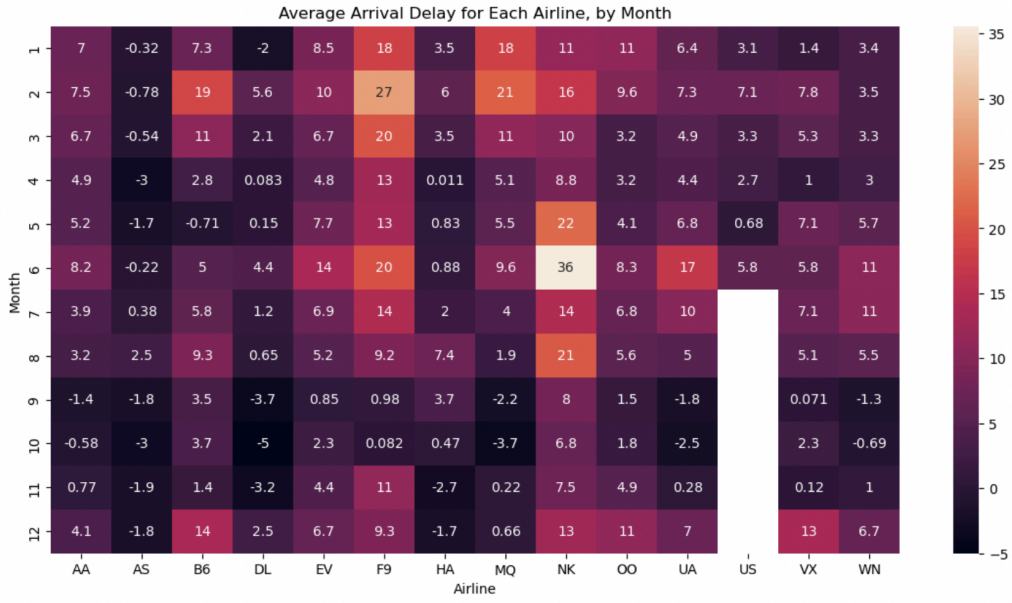
Datatypes and Missing Vals

- Use .dtype to find the data type of a dataframe
    ```

- Use .astype() to change the data type
- NaN = Not a Number = float type
- Use isnull or notnull() to find null values
    - Used weirdly:
    - n_missing_prices = len(reviews[reviews.price.isnull()])
- Use .fillna() to replace missing values
- Use .replace() to replace all entries of a certain value
    - First arg is to BE replaced
    - Second arg is to replace WITH
- Exercise Complete!

Renaming and Combining

- rename();
    - First arg is to change; 2nd arg is what to change to
    - Can replace index or column
    - `reviews.rename(columns={'points': 'score'})`
    - `reviews.rename(index={0: 'firstEntry', 1: 'secondEntry'})`
- rename_axis(); renames the axis names
    - `reviews.rename_axis("wines", axis='rows').rename_axis("fields", axis='columns')`
- Pd.concat; combines two Dataframes along one axis
- .join
    - Joins two dataframes together but they must have same indices
    - left.join(right, lsuffix= '_CAN', rsuffix= '_UK')
        - Left and right are dataframe names
        - Suffix is added to each of their respective column headings (_CAN for left and _UK for right)
- Exercise Complete!

Course 6: Data Visualization

Setup (libraries and tools)

- Import Seaborn as sns
- Import matplotlib.pyplot as plt
- Import pandas as pd
- Read in File
  - ```
    # Path of the file to read
    spotify_filepath = "../input/spotify.csv"

    # Read the file into a variable spotify_data
    spotify_data = pd.read_csv(spotify_filepath, index_col="Date", parse_dates=True)
    ```
- Plot
  - The plot is the "empty canvas" of every chart/graph; need to define it before the graph content can be displayed
  - plt.figure(figsize=(14,6))
    - 14 is width
    - 6 is height
  - plt.title("Title of Graph")
  - plt.xlabel("x-axis title")
  - plt.ylabel("y-axis title")

Line chart

- sns.lineplot(data=spotify_data)



- Can add additional parameter to lineplot function
- If we want only one line (plot a single column) we can also specify that:
  - sns.lineplot(data=spotify_data['Shape of You'], label="Shape of You")
- Exercise Complete!

Bar Charts and Heatmaps

- Bar Chart: sns.barplot(x=flight_data.index, y=flight_data['NK'])
    ```



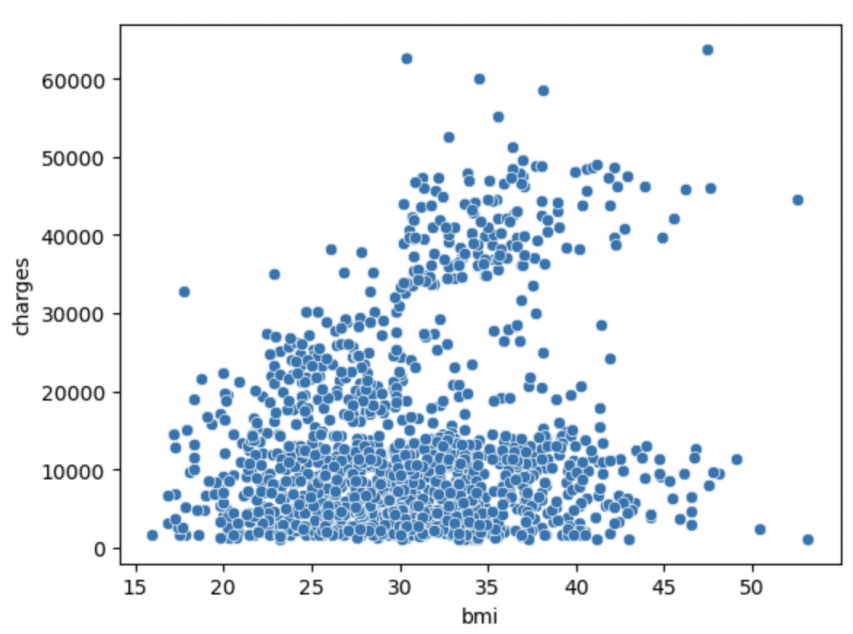
-
- The index column (month in the example) cannot be used to determine the height of the bars (y parameter)
- Heatmap: sns.heatmap(data = flightdata, annot=True)
    - Leaving out annot means the values in each cell will not show up



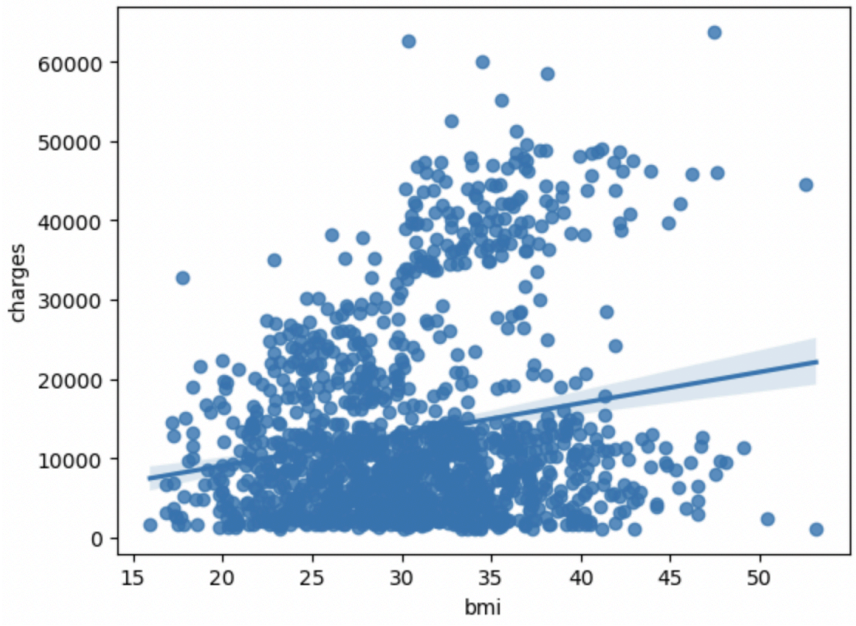
-
- Exercise Complete!

<u>Scatter Plots</u>

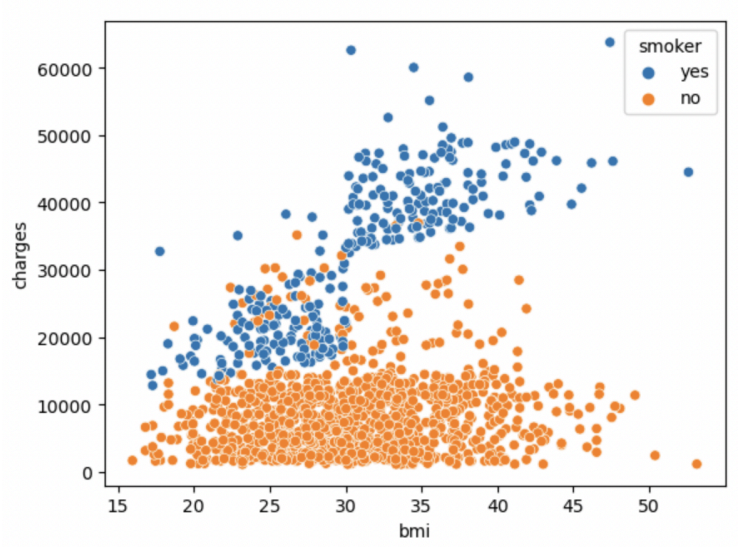
- sns.scatterplot(x=insurance_data['bmi'], y=insurance_data['charges'])
- X is the horizontal axis
- Y is the vertical axis



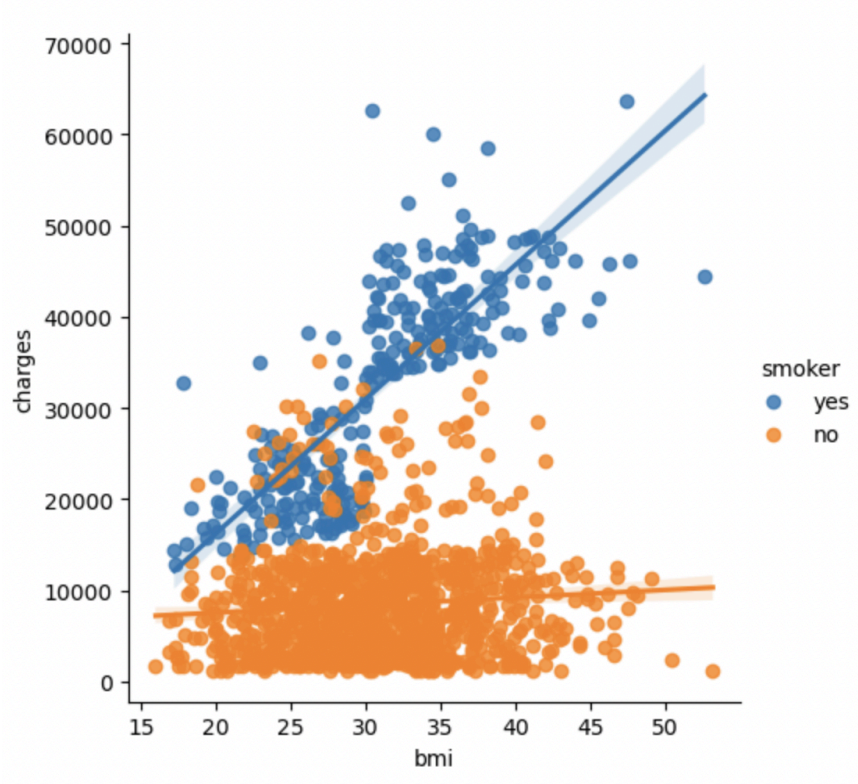
- Scatterplot with a regression line = sns.regplot



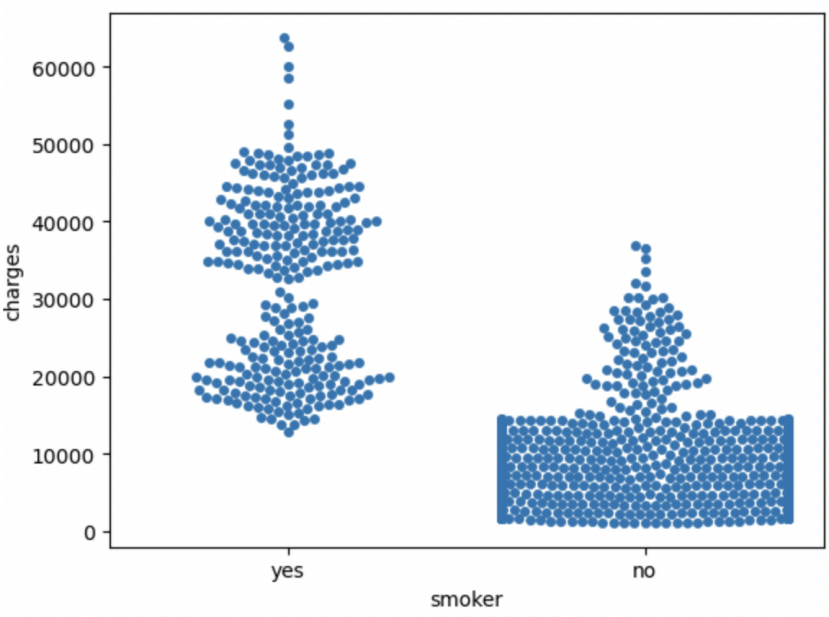
- Show 3 relationships w/ Scatterplot → Dot color as 3rd variable!
  - Add hue argument
  - sns.scatterplot(x=insurance_data['bmi'], y=insurance_data['charges'], hue = insurance_data[‘smoker’])



- Multiple regression lines for multiple relationships
  - sns.lmplot(x=”bmi”, y=”charges”, hue=”smoker”, data=insurance_data)
  - No longer need entire data column as args; Just pass in dataframe and column names



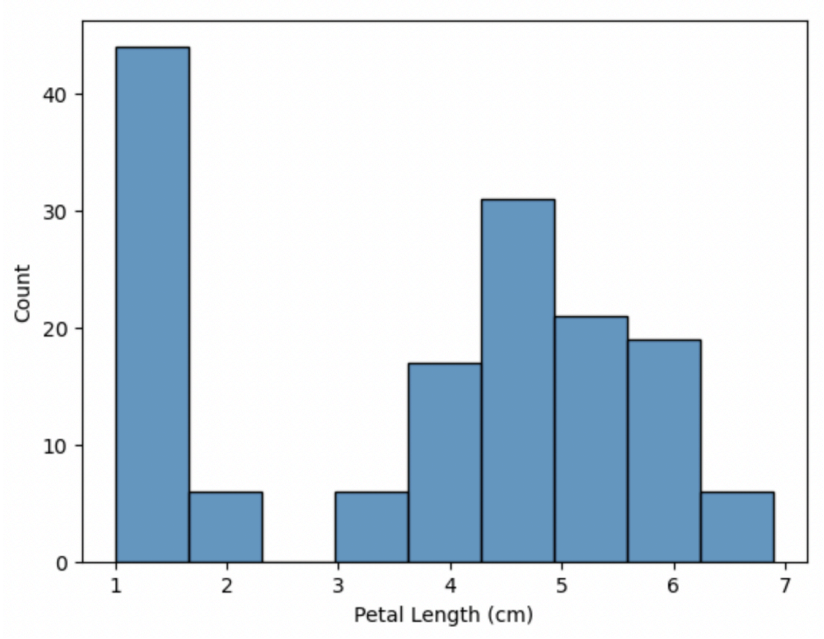
- Categorical scatterplot = swarmplot
  - sns.swarmplot(x=insurance_data['smoker'], y=insurance_data['charges'])



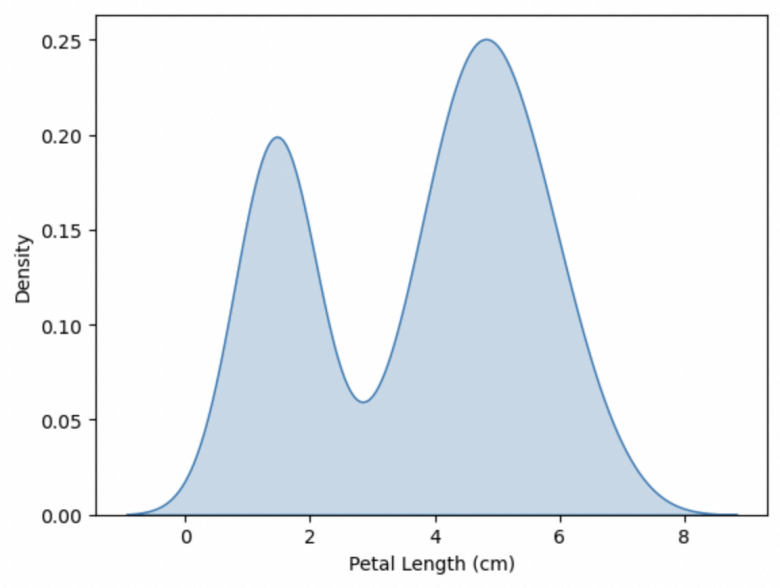
- Exercise Complete!

Distributions

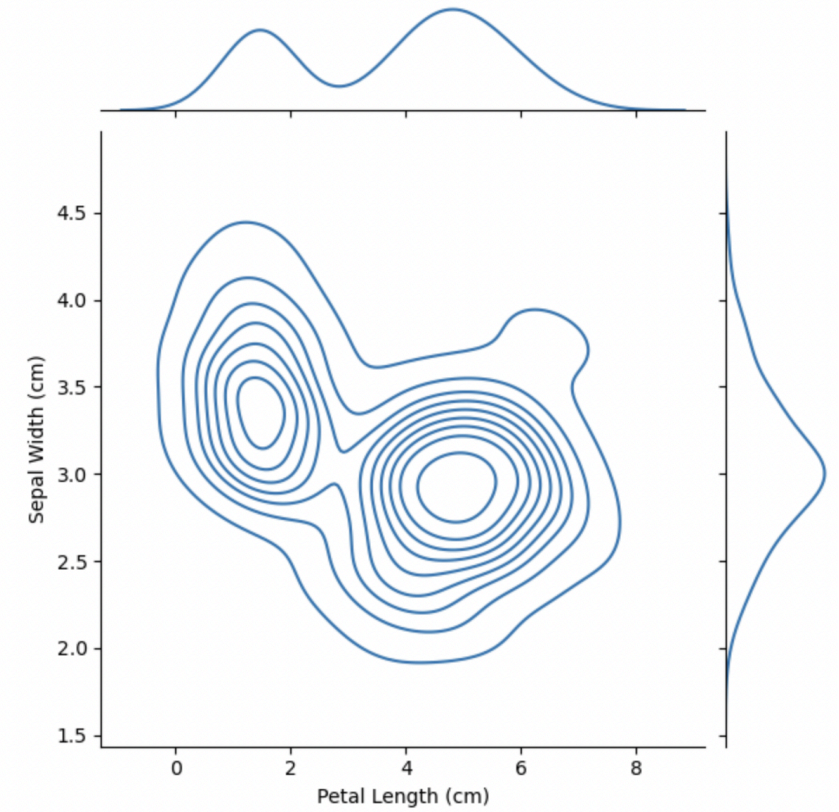
- Histograms
  - sns.histplot(iris_data[‘Petal Length (cm)’])
  - Only needs one data column, the y-axis is the frequency (count)



- Density Plots
  - Aka “smoothed histogram”
  - sns.kdeplot(data=iris_data['Petal Length (cm)'], shade=True)



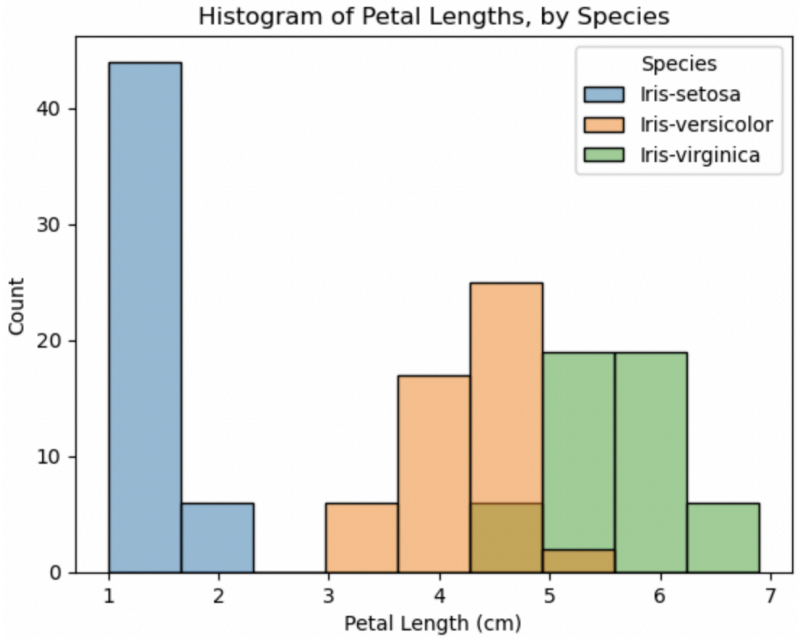
- Multidimensional KDE plot
- sns.jointplot(x=iris_data['Petal Length (cm)'], y=iris_data['Sepal Width (cm)'], kind="kde")



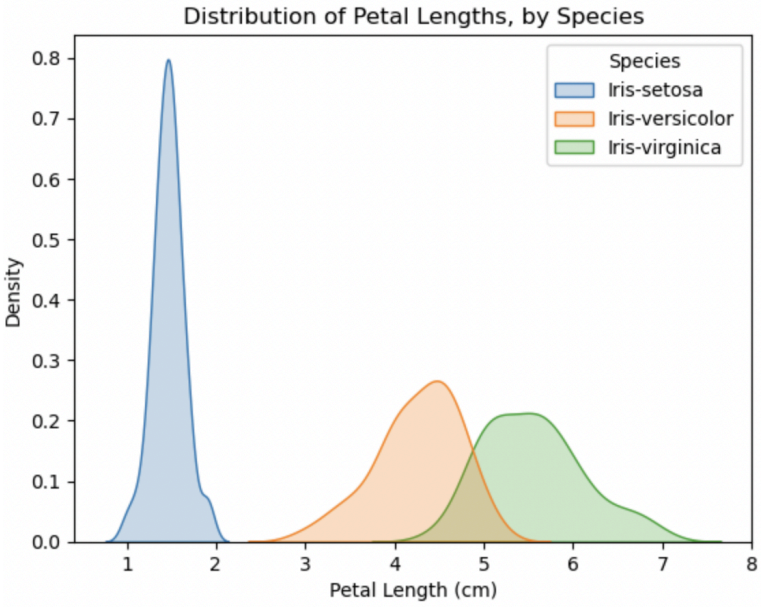
- Note: In addition to the combined density (KDE) plots in the center the graph above the x-axis is the single KDE plot for petal length while the one “above” the y-axis is the single KDE plot of the sepal width.

- Color-coded Plots

- Add parameters to histogram to communicate more! (with color)
- Define 3 things: data, x-axis (which column's data points to plot), and hue (which column to use to differentiate the x-axis column's data with color)
- sns.histplot(data=iris_data, x='Petal Length (cm)', hue='Species')



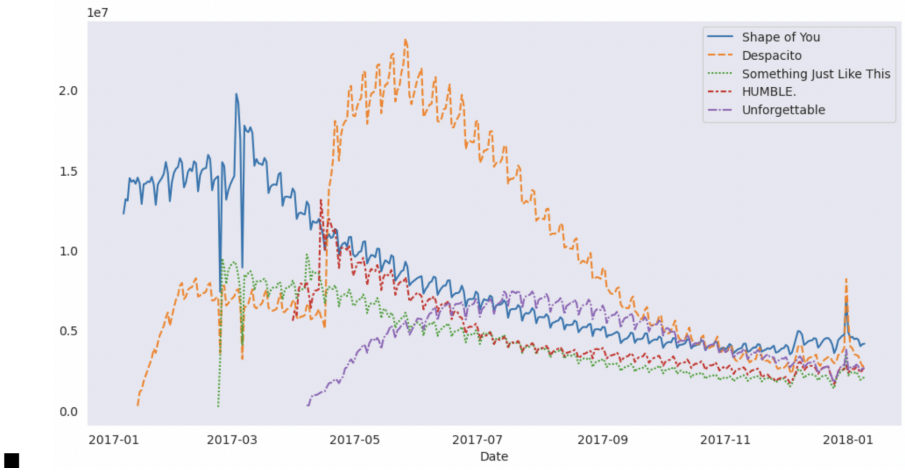
- You can also do the same thing with Density (KDE) plots!
- sns.kdeplot(data=iris_data, x='Petal Length (cm)', hue='Species', shade=True)



Choosing plot types and custom styles

- Trend: A pattern of change
  - Line Charts
- Relationships

    - Bar Charts: Quantities of different groups
    - Heat maps: Color coded patterns
    - Scatter plots: Patterns between 2 continuous variables (3 if color coded)
    - Regression plot: Adds regression line to the scatter plot for uncertain trends
    - Lmplot: multiple regression lines for multiple color-coded scatter groups
    - Swarmplots: categorical scatter plots that show relationship between continuous and categorical variable
- Distribution: To show possible values we can expect to see in a variable and how common they are
    - Histogram: a single numerical variable
    - Kdeplot (Density plot): a smooth, estimated distribution of 1-2 numerical variables
    - Jointplot: simultaneous display of 2D Kdeplot and individual Kdeplots for each variable
- Changing Style
    - Sns.set_style
    - Ex. Dark:



    - Other Options: darkgrid, whitegrid, dark, white, and ticks
- Exercise Complete!

Course 7: Feature Engineering

What is Feature Engineering

- Feature Engineering is the process of making the data better suited to the problem
- Why feature engineering
    - Improve model's predictive performance
    - Reduce computational/data needs
    - Interpret results better
- Feature engineering is basically making a clear pattern for your model to learn
    - Ex. Transforming the data points in a linear format and then fitting the model with a linear model → clear trend for model to learn
    - Allows your model to learn relationships it previously couldn't learn/identify
- How:
    - Establish a baseline score (MAE) of the data
    - Add your features and score again
    - Improve? Great, No? Discard features

Mutual Information

- Which features to choose? Where to begin? → Feature Utility Metric, a function that measures associations between feature and target
- Advantage: Detects ANY kind of relationship not just linear like correlation
- Useful to determine which features are useful. How?
    - Easy to use and interpret
    - Computationally efficient
    - Theoretically well-founded
    - Resistant to overfitting
    - Able to detect any kind of relationship
- Describes relationships in terms of uncertainty, the measure at which one quantity reduces uncertainty of another
    - Score ranges from 0.0 - inf (2.0 or above is uncommon)
    - Describes relative POTENTIAL to predict the target
    - Common for a feature to be informative but NOT have a good score with target
- Calculating MI score
    - Unlike Other models: Categorical (Continuous) and Numerical (Discrete) data need to be differentiated. Encode Categorical. Float type = Numerical
    - Two metrics in feature_selection module: mutual_info_regression for real values (numerical) and mutual_info_classif for categorical variables

```python
from sklearn.feature_selection import mutual_info_regression

def make_mi_scores(X, y, discrete_features):
    mi_scores = mutual_info_regression(X, y, discrete_features=discrete_features)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores

mi_scores = make_mi_scores(X, y, discrete_features)
mi_scores[::3]  # show a few features with their MI scores
```

```
curb_weight           1.540126
highway_mpg           0.951700
length                0.621566
fuel_system           0.485085
stroke                0.389321
num_of_cylinders      0.330988
compression_ratio     0.133927
fuel_type             0.048139
Name: MI Scores, dtype: float64
```

Creating Features

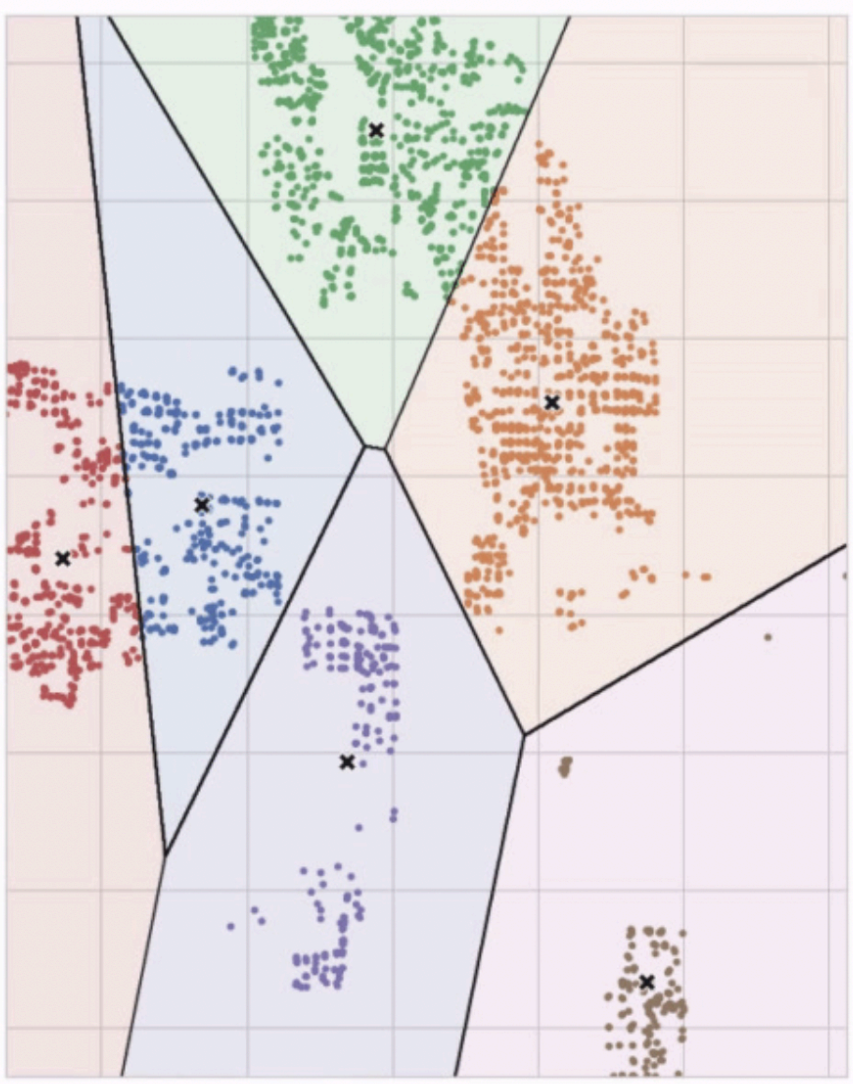
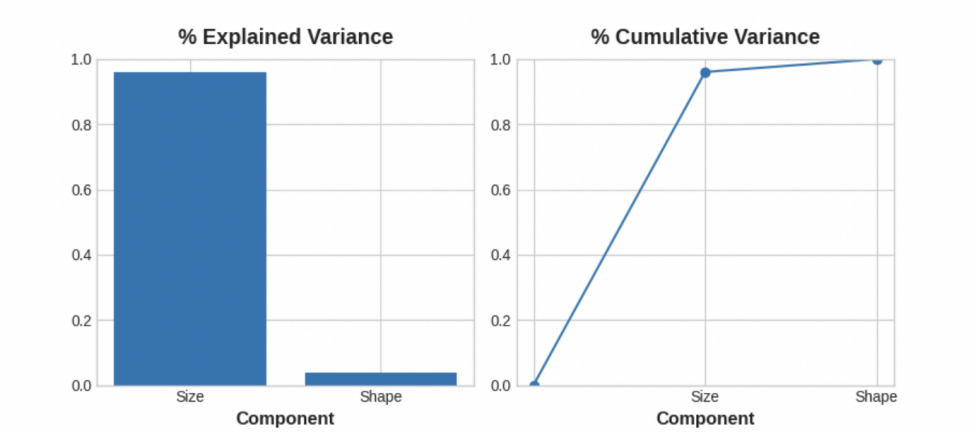
- Mathematical Transformations
  - Math transforms can combine features to make potentially new useful features or make features have a more clearly defined pattern
  - Ex. Simple Ratio
    - ```python
      autos["stroke_ratio"] = autos.stroke / autos.bore
      ```
  - Ex. Formula
    - ```python
      autos["displacement"] = (
          np.pi * ((0.5 * autos.bore) ** 2) * autos.stroke * autos.num_of_cylinders
      )
      ```
  - Ex. Log Normalization (For specifics see Data Cleaning Section)
    - ```python
      # If the feature has 0.0 values, use np.log1p (log(1+x)) instead of np.log
      accidents["LogWindSpeed"] = accidents.WindSpeed.apply(np.log1p)
      ```
- Counts
  - For binary features (i.e. the presence or absence of something) you can create a count feature to sum the occurrences of boolean features
      ```

```
roadway_features = ["Amenity", "Bump", "Crossing", "GiveWay",
    "Junction", "NoExit", "Railway", "Roundabout", "Station", "Stop",
    "TrafficCalming", "TrafficSignal"]
accidents["RoadwayFeatures"] = accidents[roadway_features].sum(axis=1)
```

    - Can also create your own boolean values w/ Pandas df methods
    - Ex. Determine the presence of an element in a concrete mix using the greater than (gt) method

```
components = [ "Cement", "BlastFurnaceSlag", "FlyAsh", "Water",
               "Superplasticizer", "CoarseAggregate", "FineAggregate"]
concrete["Components"] = concrete[components].gt(0).sum(axis=1)
```

- Building up / Breaking Down features
    - You can create new features from features with long strings in specific formats
        - Ex. Addresses, Phone numbers, Dates
        - Each can be split into new features with more info, i.e. the 1st 3 digits of phone numbers identifies location

```
customer[["Type", "Level"]] = (  # Create two new features
    customer["Policy"]           # from the Policy feature
    .str                         # through the string accessor
    .split(" ", expand=True)     # by splitting on " "
                                 # and expanding the result into separate columns
)
```

    - You can also join features if there's reason to believe they're related

```
autos["make_and_style"] = autos["make"] + "_" + autos["body_style"]
autos[["make", "body_style", "make_and_style"]].head()
```

- Group Transforms
    - Combining rows to make a new feature. Ex. "average income in person's state of residence" = state of residence + income + math transform (mean)
    - Use when a category interaction is discovered

```
customer["AverageIncome"] = (
    customer.groupby("State")  # for each state
    ["Income"]                 # select the income
    .transform("mean")         # and compute its mean
)
```

    - *Note: It's best to create a grouped feature using only the TRAINING set and then join it with the validation set

```
# Create splits
df_train = customer.sample(frac=0.5)
df_valid = customer.drop(df_train.index)

# Create the average claim amount by coverage type, on the training set
df_train["AverageClaim"] = df_train.groupby("Coverage")["ClaimAmount"].transform("mean")

# Merge the values into the validation set
df_valid = df_valid.merge(
    df_train[["Coverage", "AverageClaim"]].drop_duplicates(),
    on="Coverage",
    how="left",
)

df_valid[["Coverage", "AverageClaim"]].head(10)
```

- Tips on Feature Creation
  - Linear models learn sums and differences naturally but can't learn anything more complex
  - Ratios are hard for models to learn so creating them = easy gains
  - Neural nets and linear models do better with normalized features
    - Neural Nets especially need values closer to 0
    - Tree based models (i.e. Random forest and XGBoost) can benefit from normalization but rarely
  - Tree models CAN approximate a wide variety of features but combinations are especially important b/c they can still benefit from having them explicitly created, especially with small datasets
  - Counts are super helpful for tree models because they can't learn them naturally (no way of aggregating multiple features at once)
  - If you've discovered an interaction effect between a numeric feature and a categorical feature, you might want to model it explicitly using a one-hot encoding, like so:

```
# One-hot encode Categorical feature, adding a column prefix "Cat"
X_new = pd.get_dummies(df.Categorical, prefix="Cat")

# Multiply row-by-row
X_new = X_new.mul(df.Continuous, axis=0)

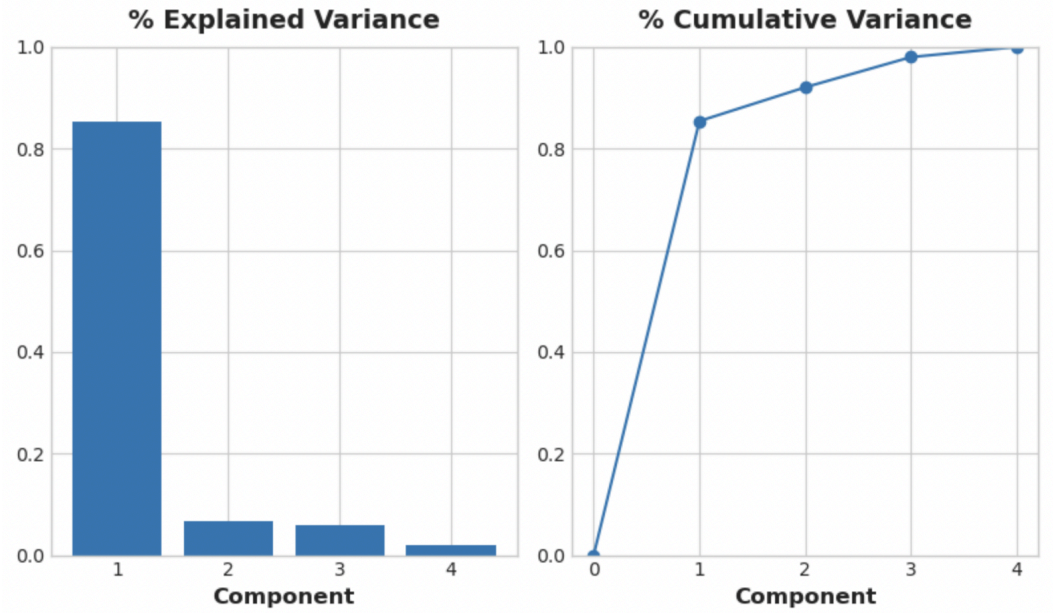
# Join the new features to the feature set
X = X.join(X_new)
```

- Exercise Complete!

<u>Clustering w/ K-Means</u>

- Unsupervised algorithm
  - No predefined target
  - Learn property of data → represent the features
  - Remember to scale the data for incomparable targets!
- Clustering = Grouping like items into buckets
- Cluster feature is categorical
- Why:
  - Breaks model into smaller "easier to learn" chunks
  - Divide and Conquer

- Center Point of Clusters = “Centroids” = k
- Intersections of the circumference of centroids forms a line → these lines form the Voronoi tesselation → indicates which cluster future points will be placed



K-means clustering creates a Voronoi tessallation of the feature space.

  -
- How:
  - Randomly initialize some predefined # of centroids
  - Assign points to nearest cluster centroid
  - Move (adjust) each centroid to minimize distance to its points
  - Repeat until centroids are no longer being adjusted (moving) or specified max # of iterations is reached
- KMeans is sensitive to Scale → Normalize Data!
- Verify
  - Plot Kmeans on a scatterplot with cluster colors to show geographic distributions
  - Create boxplot of target distribution over clusters
  - Cluster distribution should match (mostly) with scatterplot
- Exercise Complete!

Principal Component Analysis

- Another model-based method of feature engineering
- Clustering : Proximity :: PCS : variation
- Always applied to STANDARDIZED data
    - Variation = correlation
    - Unstandardized: variation = covariance
- Idea: Instead of describing data w/ original features describe via axes of variation
    - # original features = # axis of variation = new PCA "features" aka components
    - PCA features = linear combinations (weighted sums) of the original features
        - Weights = "loadings"
    - #PC components = # original features
- PCA tells us the AMOUNT of variation in each component AND DIRECTION of each variation



Size accounts for about 96% and the Shape for about 4% of the variance between Height and Diameter.

- Two Ways to use PCA for feature engineering
    - 1. Descriptive techniques
        - Compute MI scores to determine which variation is (most) predictive for target
        - Which original features merit combining?
            - Ex. Size Component good → Product of Height and Diameter feature
    - 2. Use components themselves as features
        - B/c they expose variational structure of data directly → might be more informative than original features
        - Use Cases:
            - Dimension Reductionality:
                - Features highly redundant → PCA partitions redundancy into >1 near zero variance components which can then be dropped
            - Anomaly Detection:
                - Unusual variation (not apparent in original features) show up in low variance components
            - Noise Reduction:

        - Sensor Readings Collections share common background noise → PCA collects signals into smaller number of features while leaving noise alone → Better signal:noise ratio
    - Decorrelation:
        - PCA transforms correlated features into uncorrelated components which makes some ML algorithms run easier
- PCA Best Practices
    - Quantitative data only
    - Sensitive to Scale → Standardize data before
    - Remove/constrain outliers
- Example
    -
```
features = ["highway_mpg", "engine_size", "horsepower", "curb_weight"]

X = df.copy()
y = X.pop('price')
X = X.loc[:, features]

# Standardize
X_scaled = (X - X.mean(axis=0)) / X.std(axis=0)
```
    -
```
from sklearn.decomposition import PCA

# Create principal components
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Convert to dataframe
component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
X_pca = pd.DataFrame(X_pca, columns=component_names)

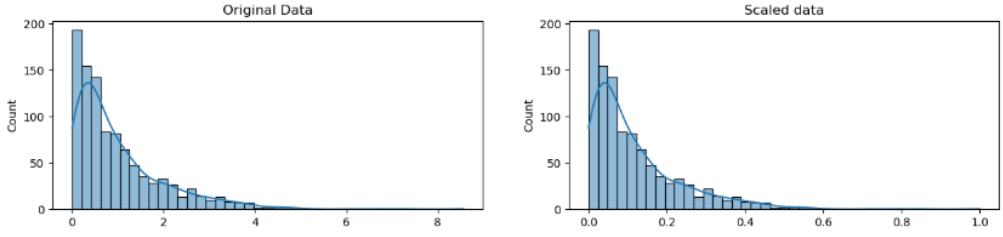
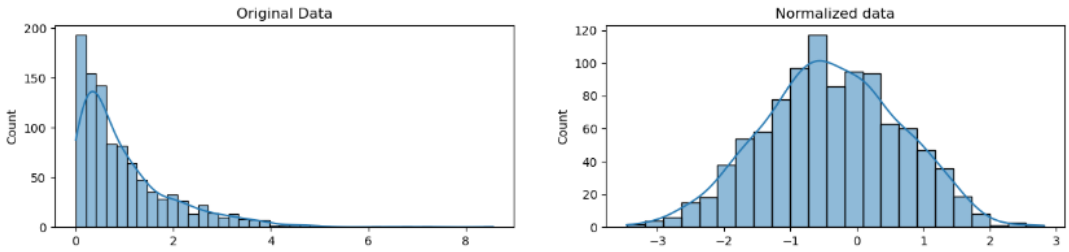
X_pca.head()
```
    -
```
loadings = pd.DataFrame(
    pca.components_.T,  # transpose the matrix of loadings
    columns=component_names,  # so the columns are the principal components
    index=X.columns,  # and the rows are the original features
)
loadings
```

	PC1	PC2	PC3	PC4
highway_mpg	-0.492347	0.770892	0.070142	-0.397996
engine_size	0.503859	0.626709	0.019960	0.594107
horsepower	0.500448	0.013788	0.731093	-0.463534
curb_weight	0.503262	0.113008	-0.678369	-0.523232

-

```
# Look at explained variance
plot_variance(pca);
```



-

```
def plot_variance(pca, width=8, dpi=100):
    # Create figure
    fig, axs = plt.subplots(1, 2)
    n = pca.n_components_
    grid = np.arange(1, n + 1)
    # Explained variance
    evr = pca.explained_variance_ratio_
    axs[0].bar(grid, evr)
    axs[0].set(
        xlabel="Component", title="% Explained Variance", ylim=(0.0, 1.0)
    )
    # Cumulative Variance
    cv = np.cumsum(evr)
    axs[1].plot(np.r_[0, grid], np.r_[0, cv], "o-")
    axs[1].set(
        xlabel="Component", title="% Cumulative Variance", ylim=(0.0, 1.0)
    )
    # Set up figure
    fig.set(figwidth=8, dpi=100)
    return axs
```

        - PC1 shows a high variance between large vehicles with poor gas mileage and small vehicles with good gas mileage (hence negative weight on highway_mpg and largest weight on engine_size)

Let's also look at the MI scores of the components. Not surprisingly, `PC1` is highly informative, though the remaining components, despite their small variance, still have a significant relationship with `price`. Examining those components could be worthwhile to find relationships not captured by the main Luxury/Economy axis.

```
mi_scores = make_mi_scores(X_pca, y, discrete_features=False)
mi_scores
```

```
PC1    1.013264
PC2    0.379156
PC3    0.306703
PC4    0.203329
Name: MI Scores, dtype: float64
```

    - PC3 for example shows relationship between horsepower and curbweight (see loadings table)
        - New feature to create from that → ratio?

Target Encoding

- PCA : Numerical :: Target Encoding : Categorical
- Idea: Replace a feature's categories with some number derived from target
    - Ex. Average price of each vehicle type (mean encoding); With binary numbers it's called binary encoding
- Problem: Overfitting
    - Target Encoding has high risk of overfitting because of 1) unknown categories and 2) rare categories that cause the encoding to be unrepresentative of future data points
- Solution: Smoothing
    - Blend in-category average with overall average
    - encoding = weight * in_category + (1 - weight) * overall
        - weight = n / (n + m)
        - M is the "smoothing factor"; larger m = more weight on overall estimate
- Use Cases:
    - High-Cardinality Features: Too many categories, one hot encoding might generate too many features. Target Encoding encodes based on the feature's most important property: relationship with the target.
    - Domain-motivated features: Reveals categorical variable's true relation with feature
- How:

The `category_encoders` package in `scikit-learn-contrib` implements an m-estimate encoder, which we'll use to encode our `Zipcode` feature.

```
from category_encoders import MEstimateEncoder

# Create the encoder instance. Choose m to control noise.
encoder = MEstimateEncoder(cols=["Zipcode"], m=5.0)

# Fit the encoder on the encoding split.
encoder.fit(X_encode, y_encode)

# Encode the Zipcode column to create the final training data
X_train = encoder.transform(X_pretrain)
```

- To generate plot to view its performance:

```
feature = encoder.cols

plt.figure(dpi=90)
ax = sns.distplot(y_train, kde=True, hist=False)
ax = sns.distplot(X_train[feature], color='r', ax=ax, hist=True, kde=False, norm_hist=True)
ax.set_xlabel("SalePrice");
```

- Exercise Complete!

Course 8: Data Cleaning

Handling Missing Values

- How many?
  - Use isnull() → df.isnull().sum() = # null entries per column = missing_values_count
  - ```
    # how many total missing values do we have?
    total_cells = np.product(nfl_data.shape)
    total_missing = missing_values_count.sum()

    # percent of data that is missing
    percent_missing = (total_missing/total_cells) * 100
    print(percent_missing)
    ```
- Why is it missing?
  - Not Applicable → Keep
  - Missing Recording → Impute
- .dropna() method to drop rows with missing values
  - Use parameter axis=1 to drop COLUMNS
- .fillna() method replaces all instances of NaN with a # of your choice
  - Slightly better fill:
  - ```
    # replace all NA's the value that comes directly after it in the same column,
    # then replace all the remaining na's with 0
    subset_nfl_data.fillna(method='bfill', axis=0).fillna(0)
    ```
- Exercise complete!

Scaling and Normalizing Data

- Scaling: Changing the RANGE of your data
  - For Distance dependent methods like "support vector machines" (SVM) or KNN
  - Ex. Dollar = 100 Yen, scale so that model doesn't think Dollar = Yen
  - 

    - The shape doesn't change but the range of data changes
- Normalization: Changing the DISTRIBUTION SHAPE of data; a more radical transformation
  - Idea: Change observations so they become normalized
  - Some models assume a normal distribution (ex. Linear Discriminant Analysis [LDA], Gaussian naive Bayes, anything with "Gaussian in the name)
  - Different ways to normalize (Ex. Box-Cox Transformation)

- 

- Exercise complete!

Parsing Dates

- Problem: “Date” entries are obvious to users as dates but not obvious to Python, Python thinks they are “objects”
- Solution: Parse the Date objects into a type called “DateTime”
  - ```
    # create a new column, date_parsed, with the parsed dates
    landslides['date_parsed'] = pd.to_datetime(landslides['date'], format="%m/%d/%y")
    ```
- Multiple Date Formats → Have Pandas infer what the correct format should be
  - `landslides['date_parsed'] = pd.to_datetime(landslides['Date'], infer_datetime_format=True)`
  - Use only when necessary:
    - Fails when data is entered poorly
    - Slower than specifying format
- After Object → DateTime Object
  - Extract day:
    ```
    # get the day of the month from the date_parsed column
    day_of_month_landslides = landslides['date_parsed'].dt.day
    day_of_month_landslides.head()
    ```
  - Drop NA columns
  - You can now work with dates intuitively!
- Identifying Problem Columns
  - Find rows with funny date lengths:
    ```
    date_lengths = earthquakes.Date.str.len()
    date_lengths.value_counts()
    ```
    ```
    Date
    10    23409
    24        3
    Name: count, dtype: int64
    ```
  - Identify problem rows:
    ```
    indices = np.where([date_lengths == 24])[1]
    print('Indices with corrupted data:', indices)
    earthquakes.loc[indices]
    ```
    ```

- Exercise Complete!

Character Encoding

- Data is stored in a type called “Bytes” aka binary to string
- Conversion from byte data type to string (human readable format) is called “Encoding”
    - Most common encoding is “UTF-8” and is default
- Other encodings do exist and converting the wrong encoding results in missing/unknown characters → unusable data set!
    - Throws “Unicode Decode Error”
- Solution
    - Blanket solution of replacing all characters that raise error with unknown character
        - myString.encode(“ascii”, encode= “replace”)
        - Not recommended b/c loss of data
    - Import “educated guesser” package called charset_normalizer to guess the encoding
        - Look at first 10k data points (don’t pass in the whole file, this is slow)
        - ```
          # look at the first ten thousand bytes to guess the character encoding
          with open("../input/kickstarter-projects/ks-projects-201801.csv", 'rb') as rawdata:
              result = charset_normalizer.detect(rawdata.read(10000))
          ```
        - Incorrect? Try on a different set of 10k data points
    - Check
        - ```
          # convert it back to utf-8
          print(after.decode("ascii"))
          ```
        - If decode has no suspicious characters then it works
    - Once correct encoding is discovered:
        - Read into csv with extra parameter (the encoding)
        - ```
          # read in the file with the encoding detected by charset_normalizer
          kickstarter_2016 = pd.read_csv("../input/kickstarter-projects/ks-projects-201612.csv", encoding='Windows-1252')
          ```
        - Save your file (saves as UTF-8 by default)
        - ```
          # save our file (will be saved as UTF-8 by default!)
          kickstarter_2016.to_csv("ks-projects-201801-utf8.csv")
          ```
- Exercise complete!

Inconsistent Data

- Not all data is entered the same way = inconsistencies = same information represented multiple ways
- Start by finding all unique entries in a column and examining those entries

```
# get all the unique values in the 'Country' column
countries = professors['Country'].unique()

# sort them alphabetically and then take a closer look
countries.sort()
countries
```

- Common ones
  - Capitalization
    - Ex. Germany vs. germany
  - Whitespace (not visible!)

```
# convert to lower case
professors['Country'] = professors['Country'].str.lower()
# remove trailing white spaces
professors['Country'] = professors['Country'].str.strip()
```

- Less Common ones + how to automate data correction
  - "FuzzyWuzzy" data package
  - Fuzzy Matching: The process of finding other strings very similar to a specified target string. Closeness is defined as the # of changes (add/remove/substitute) of a character in a string.
    - Ex. Snapple → apple is 2 changes, in → on is 1 change

```
# get the top 10 closest matches to "south korea"
matches = fuzzywuzzy.process.extract("south korea", countries, limit=10, scorer=fuzzywuzzy.fuzz.token_sort_ratio)

# take a look at them
matches
```

    - Returns a ratio score for the 10 most common matches
  - Function to replace all entries in a column with a ratio score above a certain threshold:

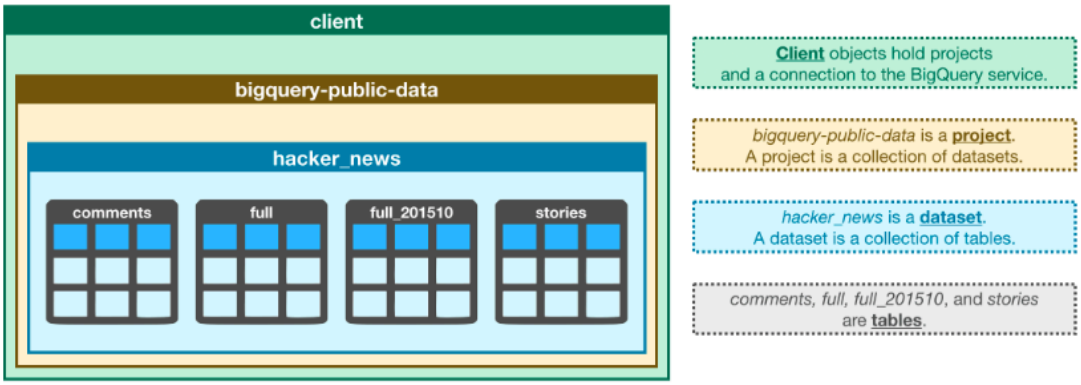
```
# function to replace rows in the provided column of the provided dataframe
# that match the provided string above the provided ratio with the provided string
def replace_matches_in_column(df, column, string_to_match, min_ratio = 47):
    # get a list of unique strings
    strings = df[column].unique()

    # get the top 10 closest matches to our input string
    matches = fuzzywuzzy.process.extract(string_to_match, strings,
                                         limit=10, scorer=fuzzywuzzy.fuzz.token_sort_ratio)

    # only get matches with a ratio > 90
    close_matches = [matches[0] for matches in matches if matches[1] >= min_ratio]

    # get the rows of all the close matches in our dataframe
    rows_with_matches = df[column].isin(close_matches)

    # replace all rows with close matches with the input matches
    df.loc[rows_with_matches, column] = string_to_match

    # let us know the function's done
    print("All done!")
```

- Exercise Complete!

Course 9: Intro to SQL

Getting Started with SQL and Big Query

- SQL = Structured Query Language
- BigQuery: A web service that allows application of SQL to (large) datasets
    - *Note: ACS Internship uses Oracle SQL Developer and their own internal database (MSPubs)
    - Setup:
        - From google.cloud import bigquery
        - Create a "client" object to talk to the dataset via SQL
            - Client = bigquery.client()
        - Construct reference to the dataset
            - Dataset_ref = client.dataset("hacker_news", project="bigquery-public-data")
        - Fetch data from API w/ reference
            - Dataset = client.get_dataset(dataset_ref)
- Analyzing a database
    - Dataset = collection of tables
        - List all tables in the dataset
            -
```
# List all the tables in the "hacker_news" dataset
tables = list(client.list_tables(dataset))

# Print names of all tables in the dataset (there are four!)
for table in tables:
    print(table.table_id)
```
        - Fetch a table
            -
```
# Construct a reference to the "full" table
table_ref = dataset_ref.table("full")

# API request - fetch the table
table = client.get_table(table_ref)
```
    -



    - Table Structure = Schema. Understand Schema → pull data we want

- Table.schema

```
[SchemaField('title', 'STRING', 'NULLABLE', 'Story title', (), None),
 SchemaField('url', 'STRING', 'NULLABLE', 'Story url', (), None),
 SchemaField('text', 'STRING', 'NULLABLE', 'Story or comment text', (), None),
 SchemaField('dead', 'BOOLEAN', 'NULLABLE', 'Is dead?', (), None),
 SchemaField('by', 'STRING', 'NULLABLE', "The username of the item's author.", (), None),
 SchemaField('score', 'INTEGER', 'NULLABLE', 'Story score', (), None),
 SchemaField('time', 'INTEGER', 'NULLABLE', 'Unix time', (), None),
 SchemaField('timestamp', 'TIMESTAMP', 'NULLABLE', 'Timestamp for the unix time', (), None),
 SchemaField('type', 'STRING', 'NULLABLE', 'Type of details (comment, comment_ranking, poll, st
ory, job, pollopt)', (), None),
 SchemaField('id', 'INTEGER', 'NULLABLE', "The item's unique id.", (), None),
 SchemaField('parent', 'INTEGER', 'NULLABLE', 'Parent comment ID', (), None),
 SchemaField('descendants', 'INTEGER', 'NULLABLE', 'Number of story or poll descendants', (), N
one),
 SchemaField('ranking', 'INTEGER', 'NULLABLE', 'Comment ranking', (), None),
 SchemaField('deleted', 'BOOLEAN', 'NULLABLE', 'Is deleted?', (), None)]
```

Each `SchemaField` tells us about a specific column (which we also refer to as a **field**). In order, the information is:

- The **name** of the column
- The **field type** (or datatype) in the column
- The **mode** of the column (`'NULLABLE'` means that a column allows NULL values, and is the default)
- A **description** of the data in that column

- Convert desired entries of table to pandas dataframe
  - `client.list_rows(table, max_results=5).to_dataframe()`

Select, From, & Where

- SELECT; to specify the column you want
- FROM; specify the table
  - Ex. query = “””
    - SELECT Name
    - FROM ‘bigquery-public-data.pet_records_pets’
      - *used ` instead of single apostrophe ‘
    - ”””””
- WHERE; specify specific conditions for data retrieval (to avoid large unnecessary retrievals)
  - Ex. query = “””
    - SELECT Name
    - FROM `bigquery-public-data.pet_records_pets`
    - WHERE Animal = ‘Cat’
    - “””””
- Find Size of query w/o running it:

```
# Create a QueryJobConfig object to estimate size of query without running it
dry_run_config = bigquery.QueryJobConfig(dry_run=True)

# API request - dry run query to estimate costs
dry_run_query_job = client.query(query, job_config=dry_run_config)

print("This query will process {} bytes.".format(dry_run_query_job.total_bytes_processed))
```

- Set query limit (run only if query < max size)

```
# Only run the query if it's less than 1 GB
ONE_GB = 1000*1000*1000
safe_config = bigquery.QueryJobConfig(maximum_bytes_billed=ONE_GB)

# Set up the query (will only run if it's less than 1 GB)
safe_query_job = client.query(query, job_config=safe_config)

# API request - try to run the query, and return a pandas DataFrame
job_post_scores = safe_query_job.to_dataframe()
```

Group By, Having & Count

- COUNT(); returns count of things
  - In: Name of Column → Out: # entries in column
  - Ex. query = “””
    - SELECT COUNT(ID)
    - FROM `bigquery-public-data.pet_records.pets`
    - “””
  - Example of an aggregate function (takes many values and returns one)
- GROUP BY
  - Takes the name of one or more columns and treats all rows of the same value in that column(s) as a single function
  - Ex. query = “””
    - SELECT Animal, COUNT(ID)
    - FROM `bigquery-public-data.pet_records.pets`
    - GROUP BY Animal
    - “””
    - Plain English = Find all columns with animals and their counts in the table group the animals by their count
  - Returns:

Animal	f0_
Rabbit	1
Dog	1
Cat	2

- GROUP BY … HAVING
  - HAVING is the conditional for grouping; HAVING : GROUP BY :: WHERE : SELECT
  - Ex. query = “””
    - SELECT Animal, COUNT(ID)
    - FROM `bigquery-public-data.pet_records.pets`
    - GROUP BY Animal
    - HAVING COUNT(ID) > 1
    - “””
    - Plain English = Find all animals and their counts, group the counts by distinct animals, and select only the groups that have a count > 1
- Aliasing (AS) and tips
  - Adding the “AS” keyword after a query can give the column a more descriptive name
  - Ex. query = “””
    - SELECT Animal, COUNT(ID)
    - FROM `bigquery-public-data.pet_records.pets`
    - GROUP BY Animal
    - “””
  - Returns
    -

	parent	NumPosts
0	24397272.0	87
1	28875764.0	43
2	30487933.0	40
3	27890790.0	96
4	19678914.0	48

    - Instead of:
    -

	parent	f0_
0	29934192.0	175
1	3373702.0	75
2	24066748.0	45
3	31123102.0	42
4	9996333.0	754

  - Inputting 1 into the Count query tells it to count the # of ROWS in the group
    - Ex. COUNT(1)
    - Good if unsure what to pass to count function