Contents

Project: Predict Taxi Fares with Random Forests

Task 1: Instructions

Read in the data consisting of 49999 New York taxi journeys.

- Load in the `tidyverse` package (which includes `dplyr` and `ggplot2`).
- Read in the `datasets/taxi.csv` file using the `read_csv()` function (not `read.csv()`) and store the resulting data frame into `taxi`.
- Take a look at the first few rows in `taxi` by using the `head()` function.

Good to know

This project assumes you have used the `dplyr` and `ggplot2` packages. Before taking on this project, we recommend that you have completed the courses Supervised Learning in R: Regression, Data Manipulation with dplyr, and Introduction to Data Visualization with ggplot2.

RStudio has created some very helpful cheat sheets, including two that will be helpful for this project: Data Wrangling and Data Visualization with ggplot2. We recommend that you keep them open in a separate tab to make it easy to refer to them.

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

Solution

```
# Loading the tidyverse
library(tidyverse)

# Reading in the taxi data
taxi <- read_csv('datasets/taxi.csv')

# Taking a look at the first few rows in taxi
head(taxi)
```

Task 2: Instructions

Rename, cleanup and add a new variable to the `taxi` dataset. All this as part of a single `%>%`-pipeline.

- `rename` the columns `pickup_latitude` and `pickup_longitude` into the shorter `lat` and `long`.
- `filter` so to keep only those rows where `fare_amount` or `tip_amount`is larger than zero (`> 0`).

- Finally, `mutate taxi` to include a new column called `total` calculated as the `log()` of `fare_amount + tip_amount`.

The reason to define `total` as the the `log()` of `fare_amount + tip_amount` is that by taking the log we remedy the effect of outliers by making really large numbers smaller.

Solution

```
# Renaming the location variables,
# dropping any journeys with zero fares and zero tips,
# and creating the total variable as the log sum of fare and tip
taxi <- taxi %>%
   rename(long = pickup_longitude, lat = pickup_latitude)  %>%
   filter(fare_amount > 0 | tip_amount > 0) %>%
   mutate(total = log(fare_amount + tip_amount) )
```

Task 3: Instructions

Filter the data (`taxi`) down to a rectangle containing Manhattan.

- `filter taxi` so that you keep only those rows in where `lat` is `between` 40.70 and 40.83, and `long` is `between` -74.025 and -73.93.
- Assign the result back to `taxi`.

`dplyr` contains a function called `between` which can simplify the code for this task. For how to use `between` with `filter` check out [this stack overflow answer](#).

Solution

```
# Reducing the data to taxi trips starting in Manhattan
# Manhattan is bounded by the rectangle with
# latitude from 40.70 to 40.83 and
# longitude from -74.025 to -73.93
taxi <- taxi  %>%
    filter(between(lat,  40.70, 40.83) &
           between(long, -74.025, -73.93))
```

Task 4: Instructions

Draw a map of Manhattan with a summary of the number of journeys on top.

- Load in the `ggmap` package and the `viridis` package (which includes a nice color scale).
- Complete the `ggmap` call by adding (`+`) a `geom_bin2d` layer with `data = taxi`, `long` on the x-axis and `lat` on the y-axis, `bins = 60`, and `alpha = 0.6`.
- Add (`+`) labels using the `labs` command for the `x`, `y` and `fill` dimensions.

For the syntax of `geom_bin2d`, and for examples of usage, check out [the official ggplot2 documentation](#).

Solution

```
# Loading in ggmap and viridis for nice colors
library(ggmap)
library(viridis)

# Retrieving a stored map object which originally was created by
# nycmap <- get_map("manhattan", zoom = 12, color = "bw")
manhattan <- readRDS("datasets/manhattan.rds")

# Drawing a density map with the number of journey start locations
ggmap(manhattan, darken=0.5)+
    scale_fill_viridis(option='plasma') +
    geom_bin2d(data = taxi, aes(x = long, y = lat), bins = 60, alpha = 0.6)
+
    labs(x = 'Longitude', y = 'Latitude', fill = 'Journeys')
```

Task 5: Instructions

Fit and visualize a regression tree to predict `total` using `lat` and `long`.

- Load in the `tree` package.
- Use the `tree()` function to fit a regression tree with `total` as the outcome and `lat` and `long` as the predictors. Assign the result to `fitted_tree`.
- Visualize `fitted_tree` by using the `plot()` and `text()` functions.

`tree()` takes a formula (`outcome ~ predictor1 + predictor2 + ...`) as it's first argument and the data frame as the second. For more information see [the documentation of the `tree` function](#). If `fitted_tree` is the output of `tree` then

```
plot(fitted_tree)
text(fitted_tree)
```

draws the tree.

Solution

```
# Loading in the tree package
library(tree)

# Fitting a tree to lat and long
fitted_tree <- tree(total ~ lat + long, data = taxi)

# draw a diagram of the tree structure
plot(fitted_tree)
text(fitted_tree)
```

Task 6: Instructions

Use the `pickup_datetime` column to add the hour, weekday, and month each trip was made.

- Load in the `lubridate` package.
- `mutate` the `taxi` dataset to include the new columns `hour`, `wday`, and `month`. Each new column should be created from the `pickup_datetime` using either of the functions `hour()`, `wday()`, and `month()`.
- Make sure to set the argument `label = TRUE` when using `wday()` and `month()`.

`taxi$pickup_datetime` describes when each taxi trip was started and `lubridate` includes many functions, like `hour()`, which extract information from a `datetime` object. See [the documentation for `lubridate`]().

Solution

```
# Loading in the lubridate package
library(lubridate)

# Generate the three new time variables
taxi <- taxi %>%
    mutate(hour = hour(pickup_datetime),
           wday = wday(pickup_datetime, label = TRUE),
           month = month(pickup_datetime, label = TRUE))
```

Task 7: Instructions

Fit a new regression tree that also includes `hour`, `wday`, and `month`.

- Like before, fit a regression tree using `tree()` and assign the result to fitted_tree. But this time include both `lat`, `long`, `hour`, `wday`, and `month` as predictors.
- Visualize the resulting `fitted_tree` using `plot()` and `text()`.
- Print a text summary of `fitted_tree` by using the `summary()` function.

Solution

```
# Fitting a tree to lat and long
fitted_tree <- tree(total ~ lat + long + hour + wday + month, data = taxi)

# draw a diagram of the tree structure
plot(fitted_tree)
text(fitted_tree)

# Summarizing the performance of the tree
summary(fitted_tree)
```

Task 8: Instructions

Instead of fitting a regression tree to the data, let's fit a random forest instead.

- Load in the `randomForest` package.
- Use the `randomForest()` function to fit at random forest with the same predictors as the for `tree` in the last task. Assign the result to `fitted_forest`.
- Set the following arguments to `randomForest()` to speed up the computation: `ntree = 80` and `sampsize = 10000`
- Print `fitted_forest` to show a summary of the result.

Solution

```
# Loading in the randomForest package
library(randomForest)

# Fitting a random forest
fitted_forest <- randomForest(total ~ lat + long + hour + wday + month,
    data=taxi, ntree=80, sampsize=10000)

# Printing the fitted_forest object
fitted_forest
```

`randomForest()` uses the same basic syntax as the `tree()` function. That is, the first argument is a formula and the second argument is the data.

Task 9: Instructions

Extract and plot the prediction of the fitted random forest.

- `fitted_forest$predicted` contains the predictions for the datapoints in `taxi`. Assign `fitted_forest$predicted` to `taxi$pred_total`.

- Copy in the plotting code from task 4.
- Replace `geom_bin2d` by `stat_summary_2d` but keep the arguments. Add `z = pred_total` to the `aes` call and `fun = mean` as an argument.
- Change `fill` label to something suitable.

`stat_summary_2d` is similar to `geom_bin2d` but takes the data connected to `z` and applies a function to it defined by the `fun =` argument. See [the documentation for `stat_summary_2d`]().

Solution

```
# Extracting the prediction from forest_fit
taxi$pred_total <- fitted_forest$predicted

# Plotting the predicted mean trip prices from according to the random
forest
ggmap(manhattan, darken=0.5) +
    scale_fill_viridis(option = 'plasma') +
    stat_summary_2d(data=taxi, aes(x = long, y = lat, z = pred_total),
                    fun = mean, alpha = 0.6, bins = 60) +
    labs(x = 'Longitude', y = 'Latitude', fill = 'Log fare+tip')
```

Task 10: Instructions

Plot the mean fare of the fitted random forest.

- Copy in the plotting code from the previous task.
- Change `z = pred_total` to `z = total`.
- Change `fun = mean` to use the defined function, that is, into `fun = mean_if_enough_data`.

You could use `fun = mean` for this plot, but that will leave a lot of squares with just a few data points which will be visually distracting. Therefore you'll use `mean_if_enough_data` instead which only returns the mean if there are 15 or more datapoints.

Solution

```
# Function that returns the mean *if* there are 15 or more datapoints
mean_if_enough_data <- function(x) {
    ifelse( length(x) >= 15, mean(x), NA)
}

# Plotting the mean trip prices from the data
ggmap(manhattan, darken=0.5) +
    stat_summary_2d(data=taxi, aes(x = long, y = lat, z = total),
                    fun = mean_if_enough_data,
                    alpha = 0.6, bins = 60) +
  scale_fill_viridis(option = 'plasma') +
```

```
  labs(x = 'Longitude', y = 'Latitude', fill = 'Log fare+tip')
```

Task 11: Instructions

Given the data and our model, where do people spend the most on taxi trips?

- Change the value of `spends_most_on_trips` to reflect where people spend the most on taxi trips. `"uptown"` or `"downtown"`?

Hint: Downtown is the lower tip of Manhattan while uptown refers to the upper half of Manhattan.

Solution

```
# Where are people spending the most on their taxi trips?
spends_most_on_trips <- "downtown" # "uptown" or "downtown"
```

If you want to know more

- Regression trees and random forests are covered in the DataCamp course Supervised Learning In R: Regression.

- Read more about the `ggmap` package here.

Project: Bad passwords and the NIST guidelines Guided

Task 1: Instructions

Load in and inspect the usernames and passwords of the fictional users.

- Import the `tidyverse` package.
- Use `read_csv` to load in the user data from `datasets/users.csv` and put it into the variable `users`.
- Count the number of users.
- Show the first 12 rows in `users` using the `head` function.

Loading the `tidyverse` package will also load the `stringr` and `readr` packages that you'll use in this project.

Make sure to use the `read_csv` (with an *underscore*) function from `tidyverse` to read in the data. The `read.csv` function which is built into R has a number of problems which the new `read_csv` function avoids.

Good to know

To complete this project, you need to know how to manipulate strings in R using the `stringr` package. You also need to know how to work with regular expressions. If you don't have experience with this, we recommend you completed the course [String Manipulation in R with stringr]() first.

An excellent companion while doing this project is the `stringr` cheat sheet from Rstudio which you can download [here]().

Solution

```
# Importing the tidyverse library
library(tidyverse)

# Loading in datasets/users.csv
users <- read_csv("datasets/users.csv")

# Counting how many users we've got
nrow(users)

# Taking a look at the 12 first users
head(users, 12)
```

Task 2: Instructions

Flag the passwords that are too short.

- Add the column `length` to `users` which should list the number of characters in each `password`.
- Flag the users with too short passwords by adding the column `users$too_short` which should be `TRUE` when `users$length` is less than 8.
- Count the number of users with passwords that are too short.
- Show the first 12 rows in `users`.

To solve this task, you need to be able to figure out the number of characters in each password. Check the `stringr` cheat sheet under **Manage Lengths** for a function that does just that.

A trick for how to count how many `TRUE` values there are in a column is to use the `sum` function. The `TRUE` values will be counted as `1` and the `FALSE` as `0`, and the `sum` will equal the number of `TRUE` values.

Solution

```
# Calculating the lengths of users' passwords
users$length <- str_length(users$password)

# Flagging the users with too short passwords
users$too_short <- users$length < 8

# Counting the number of users with too short passwords
sum(users$too_short)

# Taking a look at the 12 first rows
head(users, 12)
```

Task 3: Instructions

Load in the list with the 10,000 most common passwords.

- Read in `datasets/10_million_password_list_top_10000.txt`as a vector and put it in the variable `common_passwords`.
- Take a look at the top 100 common passwords.

The passwords are stored in a plain text file with one password per row. To read this in as a vector you need to use the `read_lines` function which takes the path to the dataset as the first argument.

Solution

```
# Reading in the top 10000 passwords
common_passwords <-
read_lines("datasets/10_million_password_list_top_10000.txt")

# Taking a look at the top 100
head(common_passwords, 100)
```

Task 4: Instructions

Flag the `user` passwords that are among the top 10,000 used passwords.

- Flag common `user` passwords by adding the column `users$common_password` which should be `TRUE` when a `password` is one of the `common_passwords`.
- Count the number of users using common passwords.
- Show the first 12 rows in `users`.

Solution

```
# Flagging the users with passwords that are common passwords
users$common_password <- users$password %in% common_passwords

# Counting the number of users using common passwords
sum(users$common_password)

# Taking a look at the 12 first rows
head(users, 12)
```

-

Task 5: Instructions

Flag the passwords that are among the 10,000 most common English words.

- Read in `datasets/google-10000-english.txt` as a vector and put it in the variable `words`.
- Flag `user` passwords that are common words by adding the column `users$common_word` which should be `TRUE` when a `password` is one of the `words`. The comparison should be *case-insensitive*.
- Count the number of users using common words as passwords.
- Show the first 12 rows in `users`.

A trick to make a comparison case insensitive is to simply use `str_to_lower` to transform the passwords to lowercase when checking whether they are in `words` or not.

Solution

```
# Reading in a list of the 10000 most common words
```

```
words <- read_lines("datasets/google-10000-english.txt")

# Flagging the users with passwords that are common words
users$common_word <- str_to_lower(users$password) %in% words

# Counting the number of users using common words as passwords
sum(users$common_word)

# Taking a look at the 12 first rows
head(users, 12)
```

Task 6: Instructions

Flag passwords that are the same as the users first or last name.

- Extract users first names from `users$user_name` into the new column `users$first_name`.
- Similarly, extract last names into the new column `users$last_name`.
- Add the column `users$uses_name` which should be `TRUE` when a `password` is the same as each users' first or last name.
- Count the number of users using names as passwords.
- Show the first 12 rows in `users`.

To extract the first and last names you can use the `str_extract` function. Check out [the `stringr` cheat sheet]() under **Subset String** for more info about that function.

You will need to supply `str_extract` with a regular expression matching what you want to extract. Again, check the second page of the cheat sheet for a reminder of what can go into a regular expression. For this task, you'll find use for the match-any-word-character `\\w`, and the word anchors `^` and `$`.

Solution

```
# Extracting first and last names into their own columns
users$first_name <- str_extract(users$user_name, "^\\w+")
users$last_name <- str_extract(users$user_name, "\\w+$")

# Flagging the users with passwords that matches their names
users$uses_name <- str_to_lower(users$password) == users$first_name |
                   str_to_lower(users$password) == users$last_name

# Counting the number of users using names as passwords
sum(users$uses_name)

# Taking a look at the 12 first rows
head(users, 12)
```

Task 7: Instructions

Flag the passwords that contain 4 or more repeated characters.

- Transform `users$password` into a list of vectors of single characters (`"abc"` → `c("a", "b", "c")`) and assign it to `split_passwords`.
- Use `sapply` to go through each `split_password` and calculate the max number of repeats. Put the result back into the column `users$max_repeats`.
- Add the column `users$too_many_repeats` which should be TRUE when a `password` has 4 or more repeated characters.
- Take a look at only the `users` with too many repeats.

There is a function in R called `rle` (standing for *run length encoding*) that is helpful here. Given a vector, it calculates the number of consecutive elements.

```
x <- c("a", "a", "b", "c", "c", "c")
n <- rle(x)$lengths
# n is now c(2, 1, 3)
```

A problem is that the passwords are strings (`"abc"`) and not vectors of single characters (`c("a", "b", "c")`). To fix this you first have to split the password strings into a list of vectors. Here is an example of how to do that:

```
x <- c("abc", "123")
l <- str_split(x, "")
# l is now list(c("a", "b", "c"),
#               c("1", "2", "3"))
```

Solution

```
# Splitting the passwords into vectors of single characters
split_passwords <- str_split(users$password, "")

# Picking out the max number of repeat characters for each password
users$max_repeats <- sapply(split_passwords, function(split_password) {
    rle_password <- rle(split_password)
    max(rle_password$lengths)
})

# Flagging the users with passwords with >= 4 repeats
users$too_many_repeats <- users$max_repeats >= 4

# Taking a look at the users with too many repeats
users[users$too_many_repeats,]
```

Task 8: Instructions

Flag *all* the bad passwords.

- Add the column `users$bad_password` which should be `TRUE` when a password is bad according to `too_short`, `common_password`, `common_word`, `uses_name`, or `too_many_repeats`.
- Count the number of users with bad passwords.
- Show the first 100 bad passwords in `users`.
-

Solution

```
# Flagging all passwords that are bad
users$bad_password <- users$too_short |
                      users$common_password |
                      users$common_word |
                      users$uses_name |
                      users$too_many_repeats

# Counting the number of bad passwords
sum(users$bad_password)

# Looking at the first 100 bad passwords
head(users$password[users$bad_password], 100)
```

Remember that you can use the *or* operator `|` to check if this *or* that is `TRUE`.

Task 9: Instructions

- Assign a new password to `new_password` that passes the NIST rules you've implemented in this project.

Solution

```
# Enter a password that passes the NIST requirements
# PLEASE DO NOT USE AN EXISTING PASSWORD HERE
new_password <- "greedyhorsedrafts42plays"
```

If you want to know more

- You can read the full NIST Special Publication 800-63B online.
- This blog post also gives you a summary of 800-63B.
- Here is an article explaining where the 10,000 common passwords you used in this project come from.
- Finally, some advice on how to come up with a good password.

Project: A Network Analysis of Game of Thrones Guided

Task 1: Instructions

Load in and inspect the edge list of the first book.

- Import the `pandas` module.
- Load the csv file for book 1 from `"datasets/book1.csv"` and assign it to `book1`.
- Print out the `head` (first 5 rows by default) of the DataFrame `book1`.

Good to know

To complete this Project you should be familiar with network analysis using the `networkx` package and be able to manipulate and inspect pandas DataFrames. We recommend that you have complete the following DataCamp courses:

- Introduction to Network Analysis in Python
- Data Manipulation with pandas

Solution

```
# Importing modules
import pandas as pd

# Reading in datasets/book1.csv
book1 = pd.read_csv('datasets/book1.csv')

# Printing out the head of the dataset
book1.head()
```

Task 2: Instructions

Create a graph object for the first book.

- Import the `networkx` module and give it the alias `nx`.
- Create an empty `Graph` object and assign it to the variable `G_book1`.

`networkx` provides different kind of graph objects, `graph`, `digraph`, `multigraph`, `multidigraph`. In this case, you will create a `graph` because the network is undirected, that is, an edge from character A to character B implies that there exists an edge the other way too, from character B to character A.

To read more about graph types you can consult the `networkx` documentation.

Solution

```
# Importing modules
import networkx as nx

# Creating an empty graph object
G_book1 = nx.Graph()
```

Task 3: Instructions

Add nodes and edges information to the network for book 1.

- Iterate through the DataFrame `book1` row-wise using `iterrows()`.
- Add edges to the graph object `G_book1` using `add_edge()`.

To populate the graph you need to iterate through `book1` to add weighted edges to the `G_book1` network using `add_edge(source, target, weight=)`. Remember that the `weight=` argument needs to be named explicitly when using `add_edge`. When iterating through the DataFrame, keep in mind that `iterrows()` returns a 2-tuple of an index and a pandas Series object. You need to use only the second part, that is, the pandas Series object to get all the information needed to populate the network. Here is a StackOverflow question that explains how to use iterrows().

You only need the source, target, and weight column to create the character co-occurrence network so only add this information to `G_book1`. To read more about the `add_edge()` method you can consult the networkx documentation.

Since we want to analyze all five books we have supplied code for creating the graphs for the other four books.

Solution

```
# Iterating through the DataFrame to add edges
for _, edge in book1.iterrows():
    G_book1.add_edge(edge['Source'], edge['Target'], weight=edge['weight'])

# Creating a list of networks for all the books
books = [G_book1]
book_fnames = ['datasets/book2.csv', 'datasets/book3.csv',
'datasets/book4.csv', 'datasets/book5.csv']
for book_fname in book_fnames:
    book = pd.read_csv(book_fname)
    G_book = nx.Graph()
    for _, edge in book.iterrows():
        G_book.add_edge(edge['Source'], edge['Target'],
weight=edge['weight'])
    books.append(G_book)
```

Task 4: Instructions

Find the most important characters according to degree centrality.

- Use `nx.degree_centrality(graph)` to calculate the centrality of all nodes from the first book (`book[0]`) and the fifth book (`book[4]`).
- Sort the resulting dictionaries `deg_cen_book1` and `deg_cen_book5` according to decreasing values and store the top 10 in `sorted_deg_cen_book1` and `sorted_deg_cen_book5`.
- Print out `sorted_deg_cen_book1` and `sorted_deg_cen_book5`.

To calculate degree centrality every node is assigned a number between 0 and 1 by computing the degree (the number of neighbors) and normalizing it by the total possible number of neighbors it can have, i.e n - 1 where n is the number of nodes in the network.

See this StackOverflow answer for how to sort a dictionary in python: https://stackoverflow.com/a/2258273

Solution

```
# Calculating the degree centrality of book 1
deg_cen_book1 = nx.degree_centrality(books[0])

# Calculating the degree centrality of book 5
deg_cen_book5 = nx.degree_centrality(books[4])

# Sorting the dictionaries according to their  degree centrality and
extracting the top 10
sorted_deg_cen_book1 = sorted(deg_cen_book1.items(), key=lambda x:x[1],
reverse=True)[0:10]

# Sorting the dictionaries according to their degree centrality and
extracting the top 10
sorted_deg_cen_book5 = sorted(deg_cen_book5.items(), key=lambda x:x[1],
reverse=True)[0:10]

# Printing out the top 10 of book1 and book5
print("Book 1", sorted_deg_cen_book1)
print("Book 5", sorted_deg_cen_book5)
```

Task 5: Instructions

Plot the evolution of degree centrality over the books for some of the characters.

- You are given a list `evol` that contains the computed degree centrality from all the books.

- Create a DataFrame with character names as columns, and index as books, where every entry is the degree centrality of the character in that particular book using `pd.DataFrame.from_records`.
- Plot the columns `Eddard-Stark`, `Tyrion-Lannister`, `Jon-Snow` from the DataFrame `degree_evol_df` using `.plot()`.

See [the pandas documentation for `pd.DataFrame.from_records`]() for more info.

Solution

```
%matplotlib inline

# Creating a list of degree centrality of all the books
evol = [nx.degree_centrality(book) for book in books]

# Creating a DataFrame from the list of degree centralities in all the
books
degree_evol_df = pd.DataFrame.from_records(evol)

# Plotting the degree centrality evolution of Eddard-Stark, Tyrion-
Lannister and Jon-Snow
degree_evol_df[['Eddard-Stark', 'Tyrion-Lannister', 'Jon-Snow']].plot()
```

Task 6: Instructions

Find the importance and evolution of characters according to betweenness centrality.

- Use `nx.betweenness_centrality(graph, weight='weight')` to calculate the weighted betweenness centrality of all the books.
- Create a DataFrame `betweenness_evol_df` just like in the previous task (`degree_evol_df`) but this time do this for betweenness centrality.
- You have been given code that finds the top 4 characters in each book and puts them into `list_of_char`.
- Plot the columns in `list_of_char` in the DataFrame `betweenness_evol_df` using `.plot()` with the argument `figsize=(13, 7)`.

Remember to use `.fillna(0)` when creating `betweenness_evol_df` as it's possible that a character is not in every book. This would result in `NaN` entries and we want to avoid that so we replace `NaN` with zero.

The intuition behind betweenness centrality is to find nodes which hold the network together, that is, if you remove such a node you are breaking apart the network. In a more mathematical way, betweenness centrality is calculated by finding shortest paths between all pairs of nodes and finding the node through which most of the paths pass.

```
# Creating a list of betweenness centrality of all the books just like we
did for degree centrality
evol = [nx.betweenness_centrality(book, weight='weight') for book in books]

# Making a DataFrame from the list
betweenness_evol_df = pd.DataFrame.from_records(evol).fillna(0)

# Finding the top 4 characters in every book
set_of_char = set()
for i in range(5):
    set_of_char |=
set(list(betweenness_evol_df.T[i].sort_values(ascending=False)[0:4].index))
list_of_char = list(set_of_char)

# Plotting the evolution of the top characters
betweenness_evol_df[list_of_char].plot(figsize=(13, 7))
```

Task 7: Instructions

Find the importance and evolution of characters according to PageRank.

- Repeat the previous task for PageRank using `nx.pagerank()` and create a list of PageRank measures for all the books.
- Create a DataFrame using `pd.DataFrame.from_records`.
- Plot the columns in `list_of_char` in the DataFrame `pagerank_evol_df` using `.plot()` with the argument `figsize=(13, 7)`.

Solution

```
# Creating a list of pagerank of all the characters in all the books
evol = [nx.pagerank(book) for book in books]

# Making a DataFrame from the list
pagerank_evol_df = pd.DataFrame.from_records(evol).fillna(0)

# Finding the top 4 characters in every book
set_of_char = set()
for i in range(5):
    set_of_char |=
set(list(pagerank_evol_df.T[i].sort_values(ascending=False)[0:4].index))
list_of_char = list(set_of_char)

# Plotting the top characters
pagerank_evol_df[list_of_char].plot(figsize=(13, 7))
```

Task 8: Instructions

Find the correlation between the three methods of measuring importance.

- Create a DataFrame using `pd.DataFrame.from_records` using the list of all `measures` for `books[4]`, and assign it to `cor`.
- Calculate the correlation using `.corr()`.

Make sure to take the transpose of the DataFrame `cor`, that is, `cor.T` first before calculating the correlation otherwise you get the correlation between characters and not the correlation between measures.

Solution

```
# Creating a list of pagerank, betweenness centrality, degree centrality
# of all the characters in the fifth book.
measures = [nx.pagerank(books[4]),
            nx.betweenness_centrality(books[4], weight='weight'),
            nx.degree_centrality(books[4])]

# Creating the correlation DataFrame
cor = pd.DataFrame.from_records(measures)

# Calculating the correlation
cor.T.corr()
```

Task 9: Instructions

- Use the `cor` DataFrame to find the *most* important character in the fifth book according to degree centrality, betweenness centrality, and PageRank.
- Print out the top character(s) according to these three measures.

This task can easily be completed using the idxmax method.

Solution

```
# Finding the most important character in the fifth book,
# according to degree centrality, betweenness centrality and pagerank.
p_rank, b_cent, d_cent = cor.idxmax(axis=1)

# Printing out the top character accoding to the three measures
print(p_rank)
print(b_cent)
print(d_cent)
```

Further Reading

If you more interested in network analysis you can have a look at the following resources:

- networkofthrones is a blog dedicated to data collection and all things networks about Game of Thrones.
- bookworm: Extracting social networks from novels.
- Part 2 of the DataCamp course on Network Analysis using Python.

The GitHub History of the Scala Language

Task 1: Instructions

Import the dataset into the notebook. All the relevant files can be found in the `datasets` subfolder.

- Import the `pandas` module.
- Load in `'datasets/pulls_2011-2013.csv'` and `'datasets/pulls_2014-2018.csv'`as pandas DataFrames and assign them to `pulls_one` and `pulls_two` respectively.
- Similarly, load in `'datasets/pull_files.csv'` and assign it to `pull_files`.

Good to know

For this Project, you need to be comfortable with pandas. The skills required to complete this Project are covered in Data Manipulation with pandas, and Merging DataFrames with pandas.

Task 2: Instructions

Combine the two pulls DataFrames and then convert `date` to a `DateTime` object.

- Append `pulls_one` to `pulls_two` and assign the result to `pulls`.
- Convert the `date` column for the `pulls` object from a string into a `DateTime` object.

For the conversion, we recommend using pandas' `to_datetime()` function. Set the `utc` parameter to `True`, as this will simplify future operations.

Coordinated Universal Time (UTC) is the basis for civil time today. This 24-hour time standard is kept using highly precise atomic clocks combined with the Earth's rotation.

Task 3: Instructions

Merge the two DataFrames.

- Merge `pulls` and `pull_files` on the `pid` column. Assign the result to the `data` variable.

The `pandas` DataFrame has a `merge` method that will perform the joining of two DataFrames on a common field.

Task 4: Instructions

Calculate and plot project activity in terms of pull requests.

- Group `data` by month and year (i.e. '2011-01', '2011-02', etc), and count the number pull requests (`pid`). Store the counts in a variable called `counts`.
    - There are a number of ways to accomplish this.
    - One way would be to create two new columns containing the year and month attributes of the `date` column, and then group by these two variables.
- Plot `counts` using a bar chart (this has been done for you).

Note, the scaffolding exists to help you create the two columns as suggested above. However, this exercise will only check whether you create `counts` correctly. Thus, alternate solutions are more than welcome!

Task 5: Instructions

Plot pull requests by user.

- Group the pull requests by each user and count the number of pull requests they submitted. Store the counts in a variable called `by_user`.
- Plot the histogram for `by_user`.

Task 6: Instructions

Identify the files changed in the last ten pull requests.

- Select the last ten pull requests and name the resulting DataFrame `last_10`.
- Merge `last_10` with the `pull_files` DataFrame on `pid`, assigning the result to `joined_pr`.
- Identify the unique files in `joined_pr` (via the `file` column) using `set()`.

Python's `DateTime` objects are comparable and sortable. A more recent date is larger than an older date. In task 2, we converted the `date` column into `DateTime` objects. Therefore, the largest ten values in the `date` column are the most recent ones.

pandas' `nlargest` method (documentation) is helpful for the first bullet.

Here is an example of using `set()` on Stack Overflow.

Task 7: Instructions

Identify the top 3 developers that submitted pull requests to `src/compiler/scala/reflect/reify/phases/Calculate.scala`.

- Select the pull requests that changed that file and name the resulting DataFrame `file_pr`.
- Count the number of changes made by each developer and name the resulting DataFrame `author_counts`.
- Print the top 3 developers.

---

pandas' `nlargest` method (documentation) is helpful for the third bullet.

Task 8: Instructions

Identify the most recent ten pull requests that touched `src/compiler/scala/reflect/reify/phases/Calculate.scala`.

- Select the pull requests that touched the file and name the resulting DataFrame `file_pr`.
- Merge `file_pr` with the `pulls` DataFrame on the `pid` column and name the resulting DataFrame `joined_pr`.
- Using `set()`, create a set of users for the ten most recent pull requests.

To find the ten most recent pull requests, use the `nlargest` function of a `DataFrame`. Again, pandas' `nlargest` method (documentation) may be helpful for this third bullet.

Task 9: Instructions

Plot the number of pull requests for two developers, over time.

- Using the `pulls` DataFrame, select all of the pull requests by these two developers and name the resulting DataFrame `by_author`.
- Fill in the `groupby` parameters to count the number of pull requests submitted by each author each year. That is, group by `user` and the year property of `date`.
- Plot `counts_wide` using a bar chart.

---

pandas' `isin` method (documentation will be helpful for bullet one.

`DateTime` objects expose the components of a date through their `dt` accessors.

`counts` is transformed to a wide format to make plotting the bar chart of pull request count (y-axis) by year (x-axis) by user (legend) easier.

Task 10: Instructions

Calculate the number of pull requests submitted by a developer to a file each year.

- Select the pull requests submitted by the authors from the `data` DataFrame and name the results `by_author`.
- Select the pull requests from `by_author` that affect the file and name the results `by_file`.
- Transform `grouped` into a wide format using `pivot_table`. Name the results `by_file_wide`.

The code required to complete bullet one in this task is the same as the code for bullet one in task 9, except on the `data` DataFrame instead of the `pulls` DataFrame.

`by_file` is transformed to a wide format to make plotting the bar chart of pull request count (y-axis) by year (x-axis) by user (legend) easier. The columns for `by_file_wide` are as follows:

- Index column: `date`
- Columns to expand: `user`
- Value columns: `pid`
- Fill value: 0

```
# Importing pandas

import pandas as pd

# Loading in the data

pulls_one = pd.read_csv('datasets/pulls_2011-2013.csv')

pulls_two = pd.read_csv('datasets/pulls_2014-2018.csv')

pull_files = pd.read_csv('datasets/pull_files.csv')


# Append pulls_one to pulls_two

pulls = pulls_two.append(pulls_one, ignore_index=True)

# Convert the date for the pulls object

pulls['date'] = pd.to_datetime(pulls['date'], utc=True)


# Merge the two DataFrames

data = pulls.merge(pull_files, on='pid')

%matplotlib inline


# Create a column that will store the month

data['month'] = data['date'].dt.month
```

```
# Create a column that will store the year
data['year'] = data['date'].dt.year

# Group by month_year and count the pull requests
counts = data.groupby(['year', 'month'])['pid'].count()

# Plot the results
counts.plot(kind='bar', figsize = (12,4))
# Required for matplotlib
%matplotlib inline

# Group by the submitter
by_user = data.groupby('user').agg({'pid': 'count'})

# Plot the histogram
by_user.hist()
# Identify the last 10 pull requests
last_10 = pulls.sort_values(by = 'date').tail(10)
last_10

# Join the two data sets
joined_pr = pull_files.merge(last_10, on='pid')

# Identify the unique files
files = set(joined_pr['file'])

# Print the results
files

# This is the file we are interested in:
file = 'src/compiler/scala/reflect/reify/phases/Calculate.scala'
```

```
# Identify the pull requests that changed the file
file_pr = data[data['file'] == file]

# Count the number of changes made by each developer
author_counts = file_pr.groupby('user').count()

# Print the top 3 developers
author_counts.nlargest(3, 'file')

file = 'src/compiler/scala/reflect/reify/phases/Calculate.scala'

# Select the pull requests that changed the target file
file_pr = pull_files[pull_files['file'] == file]

# Merge the obtained results with the pulls DataFrame
joined_pr = pulls.merge(file_pr, on='pid')

# Find the users of the last 10 most recent pull requests
users_last_10 = set(joined_pr.nlargest(10, 'date')['user'])

# Printing the results
users_last_10

%matplotlib inline

# The developers we are interested in
authors = ['xeno-by', 'soc']

# Get all the developers' pull requests
by_author = pulls[pulls['user'].isin(authors)]
```

```
# Count the number of pull requests submitted each year

counts = by_author.groupby([by_author['user'],
by_author['date'].dt.year]).agg({'pid': 'count'}).reset_index()


# Convert the table to a wide format

counts_wide = counts.pivot_table(index='date', columns='user', values='pid',
fill_value=0)


# Plot the results

counts_wide.plot(kind='bar')


authors = ['xeno-by', 'soc']

file = 'src/compiler/scala/reflect/reify/phases/Calculate.scala'


# Merge DataFrames and select the pull requests by the author

by_author = data[data['user'].isin(authors)]


# Select the pull requests that affect the file

by_file = by_author[by_author['file'] == file]


# Group and count the number of PRs done by each user each year

grouped = by_file.groupby(['user', by_file['date'].dt.year]).count()
['pid'].reset_index()


# Transform the data into a wide format

by_file_wide = grouped.pivot_table(index='date', columns='user', values='pid',
fill_value=0)


# Plot the results

by_file_wide.plot(kind='bar')
```

Project: The Android App Market on Google Play Guided

Task 1: Instructions

Import the data, drop duplicate rows, and inspect the data.

- Load `datasets/apps.csv` into a DataFrame and assign it to the variable `apps_with_duplicates`.
- Drop all duplicate rows from `apps_with_duplicates` using `drop_duplicates()` function and assign the result to `apps`.
- Print the total number of apps in `apps`.
- Print a concise summary of `apps` using the `info()` function. Which columns have missing (null) values?
  Make a note of the 4 columns that have missing values: `Rating`, `Size`, `Current Ver`, `Android Ver`.
  We will deal with this later in Task #5.
- Finally, use the `sample()` method to display a random sample of 5 rows from `apps`.

Good to know

This project lets you apply the skills from Data Manipulation with pandas. We recommend that you take this course before starting this project.

The data for this project was scraped from the Google Play website. While there are many popular datasets for Apple App Store, there aren't many for Google Play apps, which is partially due to the increased difficulty in scraping the latter as compared to the former.

A good practice is to always use the `info()`documentation function on your dataframe before beginning any analysis. This method prints information about the dataframe including the column data types, non-null values and memory usage.

Helpful links:

- pandas `read_csv()`documentation
- pandas `drop_duplicates()`documentation

Note: This project also uses the plotting libraries Seaborn and Plotly to help visualize the results of some steps. However, the tasks have been written in such a way that you should be able to complete them without any prior experience.

Task 2: Instructions

Clean the dataset.

- Create a list named `chars_to_remove` that contains the chars `+` , and `$`.
- Create a list named `cols_to_clean` that contains the following strings of column names: `Installs` and `Price`.
- For each column in `cols_to_clean` in the `apps` DataFrame, replace each character in `chars_to_remove` with the empty string `''`.
  Note: Make sure to use an empty string `''` and not a space character `' '`
- Convert each column series `apps[col]` to a numeric data type using the pandas `to_numeric()` function.

Important Note: If you run this same cell twice in a row or in succession, you will get an error. To avoid this, please always use the `Check Project` button after each task to run your notebook.

Task 3: Instructions

Create data for a bar chart that shows the distribution of apps across different categories.

- Apply the `unique()` function on `apps['Category']` to find the number of unique app categories.
- To count the number of apps in each category, apply the `value_counts()` function on `apps['Category']`. Then sort these counts in descending order using pandas `sort_values()` method and set the `ascending` flag to `False`

Helpful links:

- `unique()` documentation
- `value_counts()` documentation
- `sort_values()` documentation

Task 4: Instructions

Create a plot annotation for average app rating.

- Find the average app rating and assign it to `avg_app_rating`.

Helpful links:

- `mean()` documentation
- Plotly histogram documentation

Task 5: Instructions

Examine the relationship between size, price, and rating of apps using `jointplot()`.

Recall from Task #1 that we had observed some missing values in the `Rating` and `Size` columns. To make rational decisions, it is important that we do not consider these missing values in our analysis. We will work with a subset `apps_with_size_and_rating_present` DataFrame for this task.

- Select rows from `apps` where `Rating` is not null and `Size` is not null. Notice the use of ~ for NOT operation and the bitwise operator `&` to AND the two conditions. Store the result in the `apps_with_size_and_rating_present` dataframe.
- Apply the `groupby` function on `apps_with_size_and_rating_present['Category']`. Using `filter()` function, select only those groups or categories having greater than or equal to `250` apps. Assign the result to `large_categories` dataframe.
- Fill out `x` and `y` to create a joint plot of `Rating` as a function of `Size`.
- Subset `apps_with_size_and_rating_present` dataframe to select apps of type `Paid`. Assign the result to `paid_apps` dataframe.
- Fill out `x` and `y` to create a joint plot of `Rating` as a function of `Price`.

Helpful links:

- There are many ways to subset a dataframe and select rows based on a column value. This exercise from Data Manipulation with pandas may be a good starting place.
- `jointplot()` documentation

Task 6: Instructions

Use a strip plot to visualize the distribution of paid apps across different categories.

- Plot a strip plot with x-axis extending along the `Price` range and y-axis depicting the `Category`.
- Print `Category`, `App` and `Price` for apps that are priced above `200`.

Here are some interesting websites that can estimate app price:

- Estimate my app
- How much to make an app

Helpful links:

- `stripplot()` documentation

Task 7: Instructions

Filter out "junk" apps.
Note: For simplicity, we will continue to use the `popular_app_cats` dataframe (from previous task) and not our original dataframe `apps`

- Select rows from `popular_app_cats` that contain apps priced below $100 and assign it to `apps_under_100`.
- Re-plot your strip plot using `apps_under_100`.

Task 8: Instructions

Prep the data for a box plot that compares the number of installs of paid apps vs. number of installs of free apps.

- From `apps`, filter rows where for `Type == Paid`, and select the `Installs` column and assign it to `y` of `trace0`.
- From `apps`, filter rows where for `Type == Free`, and select the `Installs` column and assign it to `y` of `trace1`.
- Create a Python list containing variables `trace0` and `trace1`.

Helpful links:

- Plotly box plot documentation
- Interpreting box plots article

Task 9: Instructions

Load the user review data and plot it to visualize sentiment of paid vs. free apps.

- Read `datasets/user_reviews.csv` into the `reviews_df` DataFrame.
- Merge `apps` and `reviews_df` DataFrames on the common column `App` and assign the result to `merged_df`.
- Create a box plot with `Type` on the x-axis and `Sentiment_Polarity` on the y-axis.

If you'd like to learn more about sentiment analysis, check out DataCamp's Natural Language Processing Fundamentals in Python course.

Helpful links:

- pandas `merge()` function documentation
- `boxplot()` documentation

```
# Read in dataset

import pandas as pd
```

```
apps_with_duplicates = pd.read_csv("datasets/apps.csv")

# Drop duplicates
apps = apps_with_duplicates.drop_duplicates()

# Print the total number of apps
print('Total number of apps in the dataset = ', len(apps))

# Print a concise summary of apps dataframe
print(apps.info())

# Have a look at a random sample of n rows
n = 5
apps.sample(n)

# List of characters to remove
chars_to_remove = ['+', ',', '$']
# List of column names to clean
cols_to_clean = ['Installs', 'Price']

# Loop for each column
for col in cols_to_clean:
    # Replace each character with an empty string
    for char in chars_to_remove:
        apps[col] = apps[col].astype(str).str.replace(char, '')
    # Convert col to numeric
    apps[col] = pd.to_numeric(apps[col])

import plotly
plotly.offline.init_notebook_mode(connected=True)
import plotly.graph_objs as go
```

```
# Print the total number of unique categories
num_categories = len(apps['Category'].unique())
print('Number of categories = ', num_categories)

# Count the number of apps in each 'Category' and sort them in descending order
num_apps_in_category = apps['Category'].value_counts().sort_values(ascending = False)

data = [go.Bar(
        x = num_apps_in_category.index, # index = category name
        y = num_apps_in_category.values, # value = count
)]

plotly.offline.iplot(data)
# Average rating of apps
avg_app_rating = apps['Rating'].mean()
print('Average app rating = ', avg_app_rating)

# Distribution of apps according to their ratings
data = [go.Histogram(
        x = apps['Rating']
)]

# Vertical dashed line to indicate the average app rating
layout = {'shapes': [{
            'type' :'line',
            'x0': avg_app_rating,
            'y0': 0,
            'x1': avg_app_rating,
            'y1': 1000,
            'line': { 'dash': 'dashdot'}
         }]
```

```
        }

plotly.offline.iplot({'data': data, 'layout': layout})

%matplotlib inline
import seaborn as sns
sns.set_style("darkgrid")
import warnings
warnings.filterwarnings("ignore")

# Filter rows where both Rating and Size values are not null
apps_with_size_and_rating_present = apps[(~apps['Rating'].isnull()) &
(~apps['Size'].isnull())]

# Subset for categories with at least 250 apps
large_categories =
apps_with_size_and_rating_present.groupby(['Category']).filter(lambda x: len(x)
>= 250).reset_index()

# Plot size vs. rating
plt1 = sns.jointplot(x = large_categories['Size'], y = large_categories['Rating'],
kind = 'hex')

# Subset apps whose 'Type' is 'Paid'
paid_apps =
apps_with_size_and_rating_present[apps_with_size_and_rating_present['Type']
== 'Paid']

# Plot price vs. rating
plt2 = sns.jointplot(x = paid_apps['Price'], y = paid_apps['Rating'])

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_size_inches(15, 8)
```

```
# Select a few popular app categories
popular_app_cats = apps[apps.Category.isin(['GAME', 'FAMILY', 'PHOTOGRAPHY',
                                            'MEDICAL', 'TOOLS', 'FINANCE',
                                            'LIFESTYLE','BUSINESS'])]

# Examine the price trend by plotting Price vs Category
ax = sns.stripplot(x = popular_app_cats['Price'], y =
popular_app_cats['Category'], jitter=True, linewidth=1)
ax.set_title('App pricing trend across categories')

# Apps whose Price is greater than 200
apps_above_200 = popular_app_cats[['Category', 'App', 'Price']]
[popular_app_cats['Price'] > 200]
apps_above_200

# Select apps priced below $100
apps_under_100 = popular_app_cats[popular_app_cats['Price'] < 100]

fig, ax = plt.subplots()
fig.set_size_inches(15, 8)

# Examine price vs category with the authentic apps (apps_under_100)
ax = sns.stripplot(x='Price', y='Category', data=apps_under_100,
                   jitter=True, linewidth=1)
ax.set_title('App pricing trend across categories after filtering for junk apps')

trace0 = go.Box(
    # Data for paid apps
    y=apps[apps['Type'] == 'Paid']['Installs'],
    name = 'Paid'
)
```

```
trace1 = go.Box(
    # Data for free apps
    y=apps[apps['Type'] == 'Free']['Installs'],
    name = 'Free'
)

layout = go.Layout(
    title = "Number of downloads of paid apps vs. free apps",
    yaxis = dict(
        type = 'log',
        autorange = True
    )
)

# Add trace0 and trace1 to a list
data = [trace0, trace1]
plotly.offline.iplot({'data': data, 'layout': layout})

# Load user_reviews.csv
reviews_df = pd.read_csv('datasets/user_reviews.csv')

# Inner join and merge
merged_df = pd.merge(apps, reviews_df, on = "App", how = "inner")

# Drop NA values from Sentiment and Translated_Review columns
merged_df = merged_df.dropna(subset=['Sentiment', 'Translated_Review'])

sns.set_style('ticks')
fig, ax = plt.subplots()
fig.set_size_inches(11, 8)

# User review sentiment polarity for paid vs. free apps
```

```
ax = sns.boxplot(x = 'Type', y = 'Sentiment_Polarity', data=merged_df)
ax.set_title('Sentiment Polarity Distribution')
```

Project: TV, Halftime Shows, and the Big Game Guided

Task 1: Instructions

Import `pandas` then load the data.

- Read the notebook on the right before the instructions here on the left.
- Import `pandas` under the alias `pd`.
- Load the dataset's CSV files (`'datasets/super_bowls.csv'`, `'datasets/tv.csv'`, and `'datasets/halftime_musicians.csv'`) into DataFrames.

Good to know

This project gives you an opportunity to apply the skills from Intermediate Python for Data Science. DataCamp projects are completed in Jupyter Notebooks. If you'd like more info on Jupyter Notebooks, check out this introduction.

DataCamp projects are more open-ended than DataCamp courses. The **"Check Project" button** checks to see if you have completed tasks in the project, though it doesn't check for absolute code "correctness" as there can be multiple "correct" solutions sometimes. The Jupyter Notebook will still provide error messages if your code causes an error. Consult the hint and the expected output image to see what one correct solution looks like.

The **hints** for this project consist of the solution code with minimal fill-in-the-blanks represented by underscores.

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

Helpful links for this task:

- CSV to DataFrame exercise

The output for *one* correct version of a solution looks like this:

	date	super_bowl	venue	city	state	atte
0	2018-02-04	52	U.S. Bank Stadium	Minneapolis	Minnesota	
1	2017-02-05	51	NRG Stadium	Houston	Texas	
2	2016-02-07	50	Levi's Stadium	Santa Clara	California	
3	2015-02-01	49	University of Phoenix Stadium	Glendale	Arizona	
4	2014-02-02	48	MetLife Stadium	East Rutherford	New Jersey	

	super_bowl	network	avg_us_viewers	total_us_viewers	r
0	52	NBC	103390000	NaN	
1	51	Fox	111319000	172000000.0	
2	50	CBS	111864000	167000000.0	
3	49	NBC	114442000	168000000.0	
4	48	Fox	112191000	167000000.0	

	super_bowl	musician	num_son
0	52	Justin Timberlake	1

Task 2: Instructions

Display and inspect the summaries of the TV and halftime musician DataFrames for issues.

- Use the `.info()` method to inspect the DataFrame `tv`.
- Use the `.info()` method to inspect the DataFrame `halftime_musicians`.

The `.info()` method wasn't covered in Intermediate Python for Data Science so if you're stuck, check out the hint for the full solution.

You don't need to use `display()` or `print()` with `.info()` in Jupyter Notebooks because it prints to the output by default. The `'\n'` prints a blank line in between the `.info()` summaries to make them more readable.

Helpful links:

- `.info()` method documentation
- Inspecting a DataFrame exercise (in another course)

The output for *one* correct version of a solution looks like this:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53 entries, 0 to 52
Data columns (total 9 columns):
super_bowl          53 non-null int64
network             53 non-null object
avg_us_viewers      53 non-null int64
total_us_viewers    15 non-null float64
rating_household    53 non-null float64
share_household     53 non-null int64
rating_18_49        15 non-null float64
share_18_49         6 non-null float64
ad_cost             53 non-null int64
dtypes: float64(4), int64(4), object(1)
memory usage: 3.8+ KB
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 134 entries, 0 to 133
Data columns (total 3 columns):
super_bowl    134 non-null int64
musician      134 non-null object
num_songs     88 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 3.2+ KB
```

Task 3: Instructions

Plot a histogram of combined points then display the rows with the most extreme combined point outcomes.

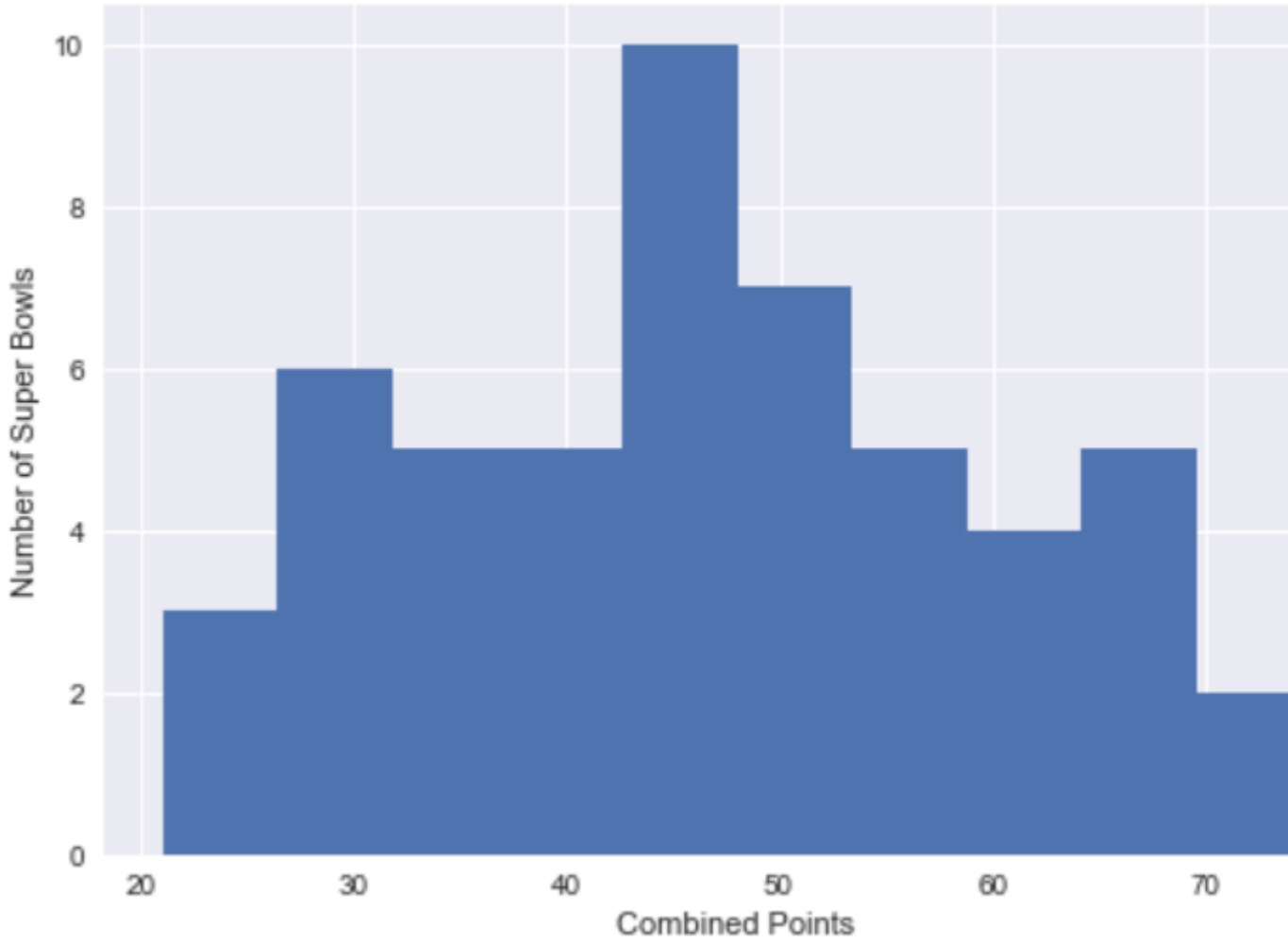
- From `matplotlib`, import the `pyplot` module under the alias `plt`.
- Create a histogram of the `combined_pts` column from the `super_bowls` DataFrame.
- Select the Super Bowl(s) where the combined score was less than 25.

`%matplotlib inline` is a magic Jupyter Notebook command that allows you to display your graphs without `plt.show()`. You only need to use `plt.show()` in this notebook if you want to display the plot before other outputs (which you do in this task).

Helpful links:

- Basic plots with `matplotlib` lesson
- Histograms lesson
- Filtering Pandas DataFrame lesson

The output for *one* correct version of a solution looks like this:



	date	super_bowl	venue	city	state	attendance	tea
0	2018-02-04	52	U.S. Bank Stadium	Minneapolis	Minnesota	67612	Ph
23	1995-01-29	29	Joe Robbie Stadium	Miami Gardens	Florida	74107	

	date	super_bowl	venue	city	state	attendance	team_w
43	1975-01-12	9	Tulane Stadium	New Orleans	Louisiana	80997	Pitts St
45	1973-	7	Memorial	Los	California	90182	

Task 4: Instructions

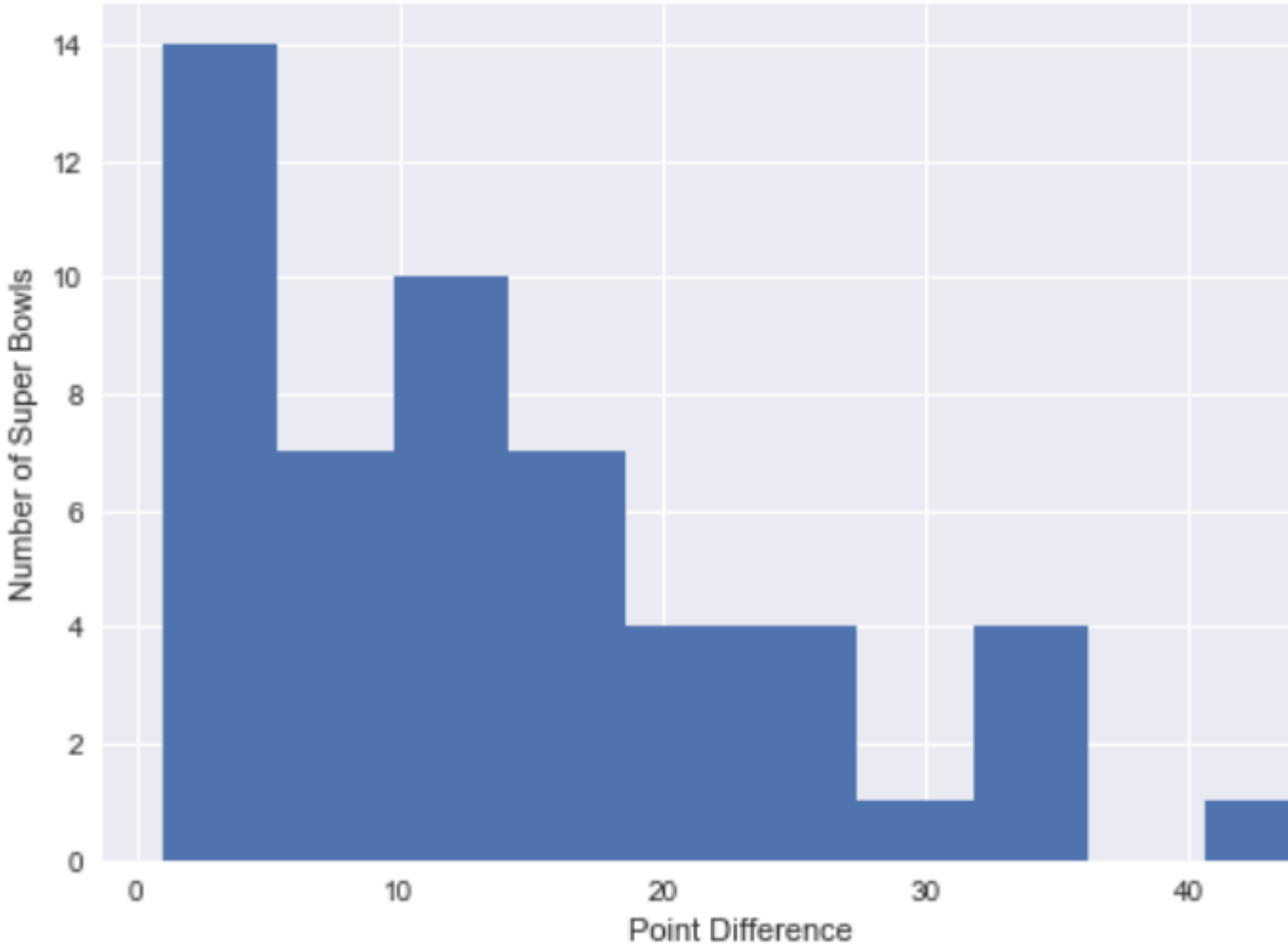
Modify and display the histogram of point differences, then display the rows with the most extreme point difference outcomes.

- Add a y-label with `'Number of Super Bowls'`.
- Display the plot with `plt.show()`.
- Select the Super Bowl(s) where the point difference was equal to 1.
- Select the Super Bowl(s) where the point difference was greater than or equal to 35.

Helpful links:

- Labels exercise

The output for *one* correct version of a solution looks like this:



	date	super_bowl	venue	city	state	attendance	team_winne
27	1991-01-27	25	Tampa Stadium	Tampa	Florida	73813	New Yor Giant

	date	super_bowl	venue	city	state	attendance	tea
4	2014-02-02	48	MetLife Stadium	East Rutherford	New Jersey	82529	
25	1993-01-31	27	Rose Bowl	Pasadena	California	98374	
28	1990-01-28	24	Louisiana Superdome	New Orleans	Louisiana	72919	

Task 5: Instructions

Import `seaborn` and plot household share vs. point difference.

- Import the `seaborn` module under the alias `sns`.
- Fill in the `x` argument of `sns.regplot()` with the point difference column
- Fill in the `y` argument of `sns.regplot()` with the household share column.

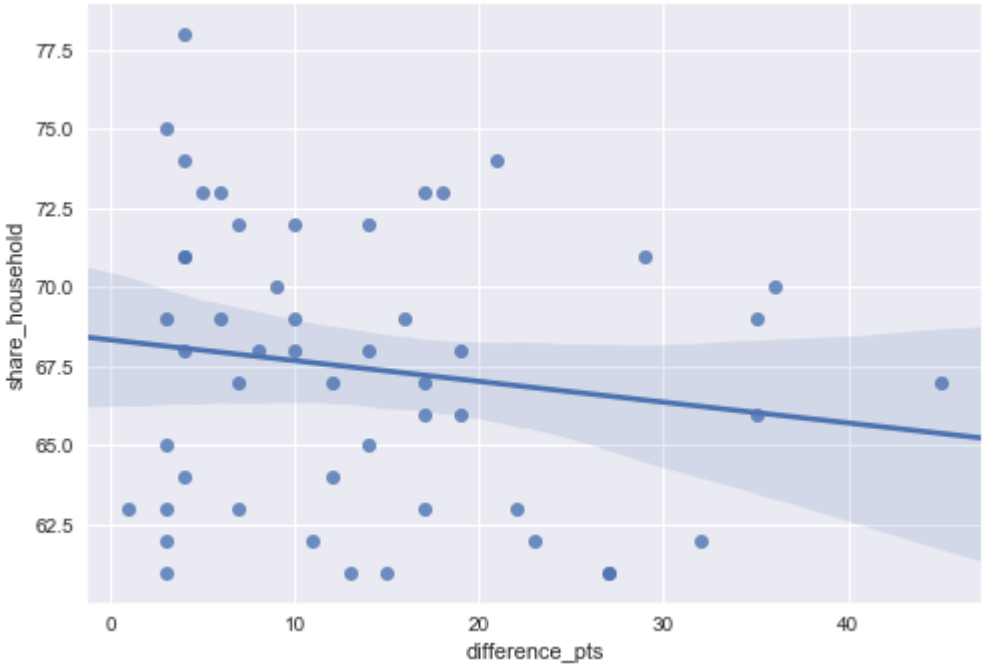
Remember column names are represented as strings!

`seaborn`'s `regplot()` is like scatter plot except more specialized for [visualizing linear relationships](). It draws a scatterplot, then fits a regression model and plots the resulting regression line and a 95% confidence interval for that regression.

Helpful links:

- Packages [lesson]()

The output for *one* correct version of a solution looks like this:



Task 6: Instructions

Create three line plots using the `tv` DataFrame to compare viewers, rating, and ad cost.

- For the first plot, plot `super_bowl` on the x-axis, `avg_us_viewers` on the y-axis, and set the line color to `'#648FFF'`.
- For the second plot, plot `super_bowl` on the x-axis, `rating_household` on the y-axis, and set the line color to `'#DC267F'`.

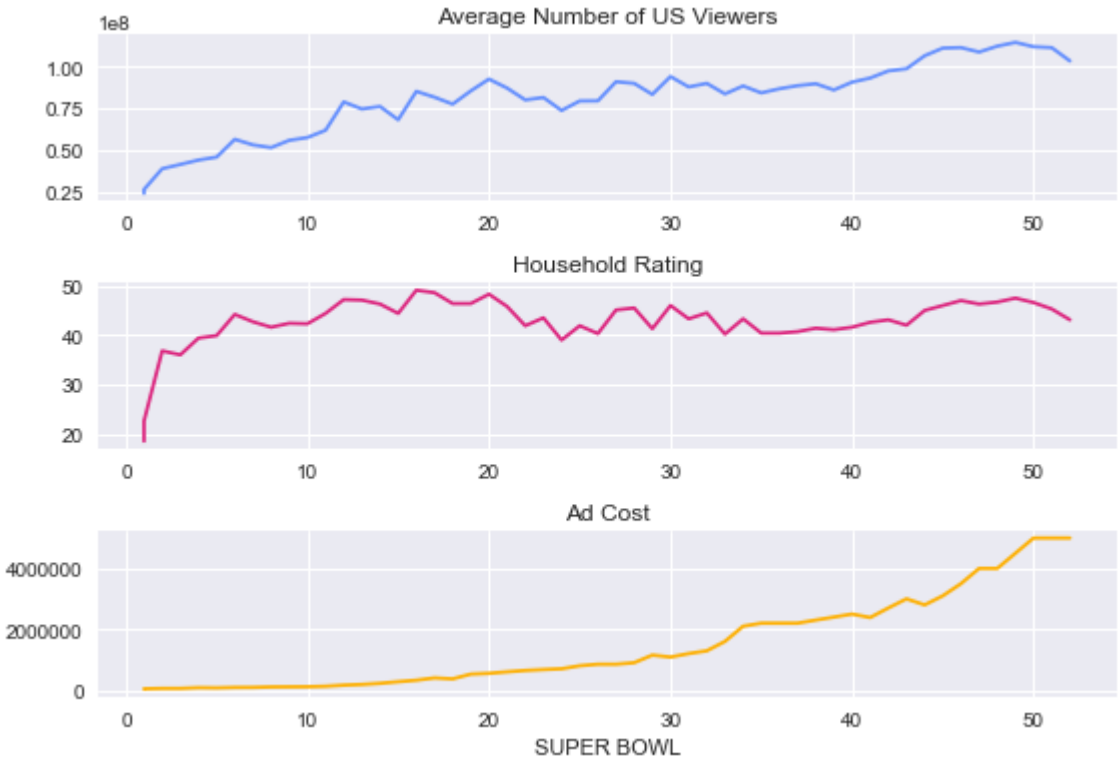
- For the third plot, plot `super_bowl` on the x-axis, `ad_cost` on the y-axis, and set the line color to `'#FFB000'`.

The colors for the lines were based on a palette suggestion from [Coloring for Colorblindness](#).

Helpful links:

- Line plot [exercise](#)

The output for *one* correct version of a solution looks like this:



Task 7: Instructions

Filter and display the musicians for halftime shows up to and including Super Bowl XXVII.

- Using `halftime_musicians`, select the musicians that performed in halftime shows up to and including Super Bowl XXVII (27) (i.e. Michael Jackson's performance).

The last line of code in a Jupyter Notebook cell automatically gets it output displayed so you don't need to use `display()` here.

The output for *one* correct version of a solution looks like this:

	super_bowl	mu
80	27	Michael J
81	26	Gloria
82	26	University of Minnesota Marchin
83	25	New Kids on the
84	24	Pete F
85	24	Doug K
86	24	Irma T
87	24	Pride of Nicholls Marchin
88	24	The Human J
89	24	Pride of A
90	23	Elvis
91	22	Chubby C

Task 8: Instructions

Select and display the musicians with more than one halftime show appearance.

- The new `halftime_appearances` DataFrame has two columns, `musician` and `super_bowl`, where `super_bowl` now contains the halftime show counts for each musician. Select the musicians that have appeared in more than one halftime show.

The `halftime_appearances` code is preloaded because it wasn't covered in the prerequisite for this project, [Intermediate Python for Data Science]. Grouping and rearranging data are covered in [Manipulating DataFrames with pandas].

The output for *one* correct version of a solution looks like this:

	musician	super
28	Grambling State University Tiger Marching Band	
104	Up with People	
1	Al Hirt	
83	The Human Jukebox	
76	Spirit of Troy	
25	Florida A&M University Marching 100 Band	
26	Gloria Estefan	
102	University of Minnesota Marching Band	
10	Bruno Mars	
64	Pete Fountain	
5	Beyoncé	
36	Justin Timberlake	
57	Nelly	
44	Los Angeles Unified School District All City H...	

Task 9: Instructions

Modify the histogram of number of songs performed for non-band musicians.

- In the `plt.hist()` function, set the number of bins argument equal to `most_songs` (the most number of songs performed in a halftime show by a single musician).
- Add an x-label with `'Number of Songs Per Halftime Show Performance'`.

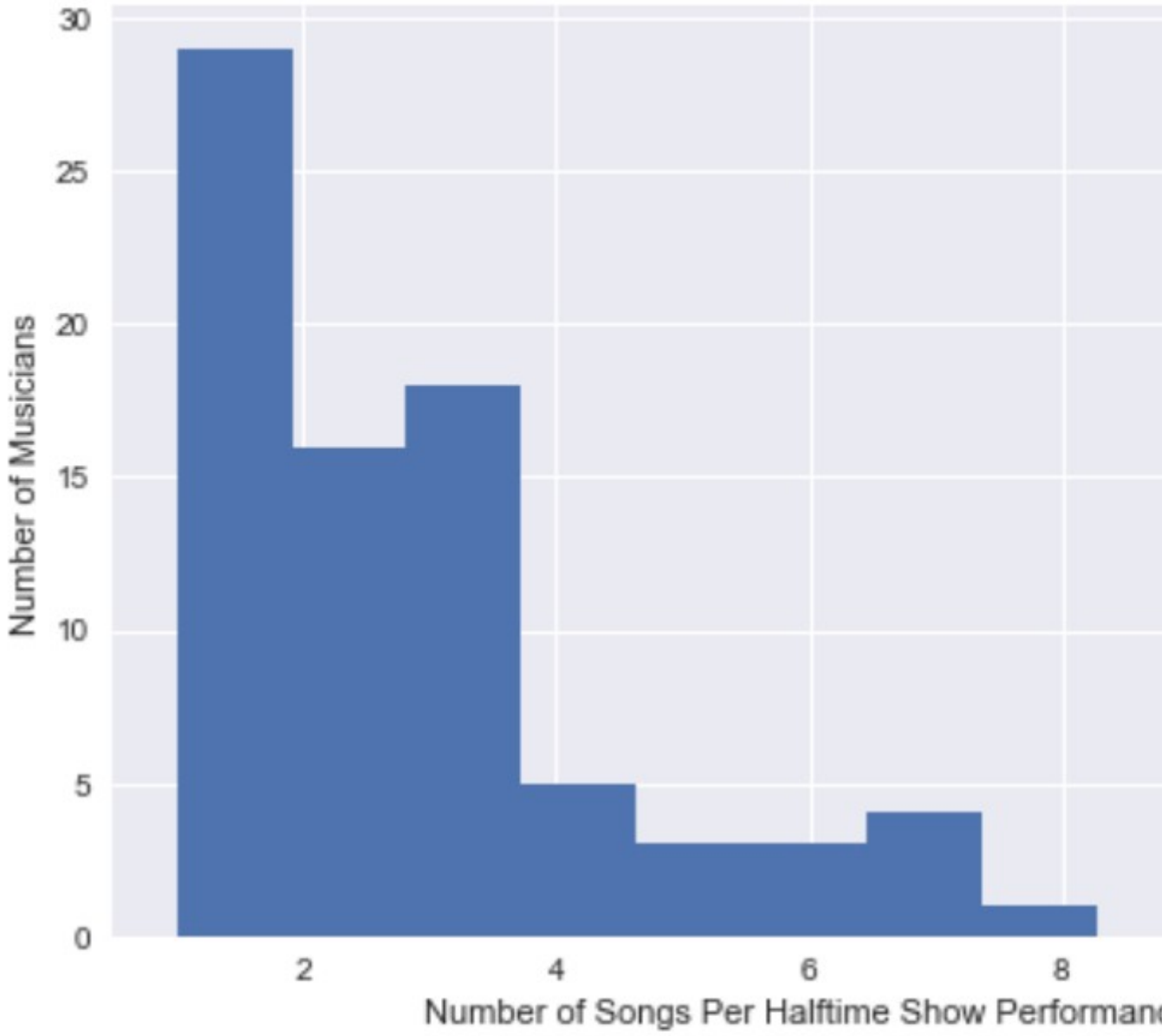
You can't filter out "Band" because Bruce Springsteen and the E Street Band performed at Super Bowl XLIII.

The `no_bands` code is preloaded because it wasn't covered in Intermediate Python for Data Science. The `.str.contains()` method is covered in Cleaning Data in Python.

Helpful links:

- Build a histogram: bins exercise

The output for *one* correct version of a solution looks like this:



	super_bowl	musician	num_songs
0	52	Justin Timberlake	11.0
70	30	Diana Ross	10.0
10	49	Katy Perry	8.0
2	51	Lady Gaga	7.0
90	23	Elvis Presto	7.0

Task 10: Instructions

Who will win Super Bowl LIII?

- The `patriots` and `rams` are playing in Super Bowl LIII. Assign the variable of the team you think will win to the `super_bowl_LIII_winner` variable.

Congratulations on reaching the end of the project! You just applied your Python skills in a real-world data analysis. The structure of this project (where code intersperses narrative) is an excellent structure for blog posts to add to your data science portfolio.

To continue building your Python skills, continue to the next course in your track. If you're not enrolled in a track, pick a new course from DataCamp's Python library.

Project: Importing and Cleaning Data Guided

Task 1: Instructions

Load the packages and read in the data.

- Load the `readr` and `dplyr` packages.
- Read in the data from the **datasets folder** using `read_csv()` and assign it to the variable, `wwc_raw`. The name of the data file is `2019_WWCFIFA_summary.csv`.
- Determine the class of `wwc_raw` and its dimensions. Use `glimpse()`, `summary()`, and `str()` to look its structure.

Good to know

This project lets you apply the skills from Introduction to the Tidyverse, Introduction to Importing Data in R, and Data Cleaning in R. We recommend that you take these courses before starting this project.

`wwc` is a shortened version of "Women's World Cup".

Helpful links:

- RStudio data import cheat sheet
- Packages vs. Libraries in R blog post

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

Task 2: Instructions

Import data and assign data types.

- Use `read_csv()` to read in the data again. This time parse `Round` to factor, `Date` to date, and `Venue` to factor by setting the `col_types` argument.
- Inspect the data by calling `glimpse()` and `summary()`.
- The dataset is not large - print it and explore.

Helpful links:

- `read_csv()` column parsers
- `read_csv()` parsing dates and times

Task 3: Instructions

Change all column names to lowercase and remove rows of `NA`.

- Load the `tidyr` package.
- Create `wwc_1` by first changing all column names to lowercase, then removing rows of `NA`.
- Get the dimensions of `wwc_1`, then inspect the first ten and last ten rows.

One way to remove rows of `NA` is to filter for `!is.na()`. Filter on a column of data that has non-NA values in all cells except for the cells that are part of the rows of NAs.

Helpful links:

- Missing and special values in R
- `rename_all()` documentation (scroll down)
- `tolower()` documentation
- `filter()` documentation
- `is.na()` documentation

Task 4: Instructions

Find the `NA` and replace them with correct data.

- Using `which()` and `is.na()`, find the index value for the `NA` in the `date` column of **`wwc_2`** and assign it to `index_date`. Use `[]` to subset and view the row with the missing data, then replace the `NA` with the correct data value (given).
- Repeat the process for `venue`.

A second approach would be to use `replace_na()`.

```
wwc_2 <-  wwc_1 %>%
  mutate(date = replace_na(date, "2019-06-09"),
         venue = replace_na(venue, "Groupama Stadium"))
```

Helpful links:

- `which()` documentation
- `is.na()` documentation
- `replace_na()` documentation

Task 5: Instructions

Separte data in columns and replace `NA`.

- Create `wwc_3` from `wwc_2` by calling `separate()` twice, and then calling `replace_na()` **twice** within `mutate()`. Separate `score` into `home_score` and `away_score`. Separate `pks` into `home_pks` and `away_pks`. Replace the `NA` in `home_pks` and `away_pks` with `0`.
- Print the data and marvel at your awesome data cleaning skills!

Don't forget to corretly set the `convert =` and `sep =` arguments in `separate()`. If you are stuck, check the hint.

Helpful links:

- `separate()`documentation
- `mutate()`documentation
- `replace_na()` documentation

Task 6: Instructions

Create a boxplot to look for outliers.

- Load the `ggplot2` package.
- Using `wwc_3`, create a boxplot of `attendance` by `venue` using `geom_boxplot()`. The first line of code is the call to `ggplot()` with the correct `x` and `y` aesthetics. The second line of code is the call to the boxplot geometry. Do not forget the `+`.

`geom_jitter()` is a convenient way to add random variation to the location of each point. This random variation makes overlapping points easier to visualize.

Helpful links:

- `ggplot2` package
- `geom_boxplot()` documentation
- `geom_jitter()` documentation
- `theme()` documentation

Task 7: Instructions

Summarize the attendance at each venue and fix the outlier.

- Summarize the number of games, minimum attendance, and maximum attendance **at each venue** in `wwc_3`.
- Use `mutate()`, and `which()` within `replace()`, to update the outlier with the correct value, `57900`. Assign the updated dataset to `wwc_4`.
- After updating the outlier, summarize the number of games, minimum attendance, and maximum attendance **at each venue** and print the summary table.

We are using a different method to replace the outlier here. Check the Helpful links or the hint if you are having difficulty.

Helpful links:

- `group_by()` documentation
- `summarize()` documentation
- `replace()` and `mutate()` SO Discussion

Task 8: Instructions

Redo the boxplot from Task 6. This time make it prettier.

- Add the correct geometries to create a boxplot of the attendance by venue. Add red, jittered points over the boxplot. Reduce the size of the points to `0.5`.
- Within `labs()`, add a title, `"Distribution of attendance by stadium"`, and a subtitle, `"2019 FIFA Women's World Cup"`.

Instead of loading the package `forcats`, we can call the function with `forcats::fct_reorder()`.

If you're stuck, look back at the code from Task 6 or at the hint.

Helpful links:

- `ggplot2` labs documentation
- `coord_flip()` documentation
- `fct_reorder()` documentation

Task 9: Instructions

Make a line plot of venue attendance over the duration of the tournament.

- Add the correct x and y aesthetics and color each line by `venue`.
- Add the correct geometry to make a line plot.

Helpful links:

- `geom_line()` documentation

Task 10: Instructions

Answer the following multiple-choice questions.

- Which match had the highest attendance during the tournament?
- In what stadium was the match with the highest attendance played?

You might have to do a little more coding to figure out the first question. Use the space below to run more code to help you answer the questions. You can run a cell without checking the entire project by clicking the "Run" button or using `Ctrl-Enter`.

Helpful links:

- `arrange()` documentation

```
# load the packages
library(readr)
library(dplyr)

# Read in the data from the datasets folder
wwc_raw <- read_csv("datasets/2019_WWCFIFA_summary.csv")

# Check the dimensions and structure of the data
dim(wwc_raw)
class(wwc_raw)
glimpse(wwc_raw)
str(wwc_raw)
summary(wwc_raw)

# Read in the data
wwc_raw <- read_csv("datasets/2019_WWCFIFA_summary.csv",
            col_types = cols(
                    Round = col_factor(),
                    Date = col_date(format = "%m/%d/%y"),
                    Venue = col_factor()
```

```
                                )
                )

# Call summary() and glimpse()
glimpse(wwc_raw)
summary(wwc_raw)

# Print the dataset
wwc_raw

# load the package
library(tidyr)

# Remove rows of NA
wwc_1  <- wwc_raw  %>%
 rename_all(tolower)  %>%
 filter(!is.na(round))

# Get the dimensions and inspect the first 10 and last 10 rows
dim(wwc_1)
head(wwc_1, 10)
tail(wwc_1, 10)

# Housekeeping
wwc_2 <- wwc_1

# Find and replace NA in column date
index_date <- which(is.na(wwc_2$date))
wwc_2[index_date, ]
wwc_2$date[index_date] <- "2019-06-09"

# Find and replace NA in column venue
```

```
index_venue <- which(is.na(wwc_2$venue))
wwc_2[index_venue, ]
wwc_2$venue[index_venue] <- "Groupama Stadium"

# Separate columns and replace NA
wwc_3  <- wwc_2  %>%
  separate(score, c("home_score", "away_score"), sep =  "-", convert = TRUE)
%>%
  separate(pks, c("home_pks", "away_pks"), sep = "-", convert = TRUE)  %>%
  mutate(home_pks = replace_na(home_pks, 0),
         away_pks = replace_na(away_pks, 0))

# Print the data
wwc_3

# Housekeeping for plot size
options(repr.plot.width=6, repr.plot.height=4)

# Load package
library(ggplot2)

# Make a boxplot of attendance by venue and add the point data
ggplot(wwc_3, aes(venue, attendance)) +
  geom_boxplot() +
  geom_jitter(color = "red", size = 0.5) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))

# Print the number of games played at each venue, and the min and max
attendance at each venue
wwc_3  %>%
  group_by(venue)  %>%
  summarize(nb_of_games = n(),
            min_attendance = min(attendance),
```

```
          max_attendance = max(attendance))

# Correct the outlier
wwc_4  <- wwc_3  %>%
  mutate(attendance = replace(attendance, which(attendance == 579000),
57900))

# Print the updated summary table
wwc_venue_summary  <- wwc_4 %>%
  group_by(venue)  %>%
  summarize(nb_of_games = n(),
          min_attendance = min(attendance),
          max_attendance = max(attendance))

# Print an updated summary table
wwc_venue_summary

# Housekeeping for plot size
options(repr.plot.width=6, repr.plot.height=4)

# Prettier boxplot of attendance data by venue
wwc_4  %>%
  ggplot(aes(x= forcats::fct_reorder(venue, attendance), y = attendance)) +
  geom_boxplot() +
  geom_jitter(color = "red", size = 0.5) +
  coord_flip() +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, , hjust = 1)) +
  labs(title = "Distribution of attendance by stadium",
      subtitle = "2019 FIFA Women's World Cup", x = "Stadium", y =
"Attendance")

# Housekeeping for plot size
```

```
options(repr.plot.width=6, repr.plot.height=4)

# Line plot of attendance over time
wwc_4  %>%
  ggplot(aes(date, attendance, color = venue)) +
  geom_line() +
  theme_minimal() +
  theme(legend.position = "bottom",
       legend.text = element_text(size = 8)) +
  guides(col = guide_legend(nrow = 3)) +
  labs(title = "Stadium attendance during the tournament",
       subtitle = "2019 FIFA Women's World Cup",
       x = "Date",
       y = "Attendance",
      color = "")

wwc_4 %>% arrange(desc(attendance))

# What game had the higest attendance?
ans_1  <- "B"

# Which stadium was the game with highest attendance play in?
ans_2  <- "A"
```

Project: Dr. Semmelweis and the Discovery of Handwashing Guided

ask 1: Instructions

Load in the dataset with the yearly number of deaths.

- Import the `pandas` module.
- Read in `datasets/yearly_deaths_by_clinic.csv` and assign it to the variable `yearly`.
- Print out `yearly`.

Good to know

To complete this project you need to know some python and be familiar with `pandas` DataFrames and bootstrap analysis. Here are relevant DataCamp exercises if you need to brush up your skills:

- From Intermediate Python for Data Science
  - Reading in a csv-file
  - Selecting columns using `[]`
- From Data Manipulation with pandas
  - Inspecting a DataFrame
- From Statistical Thinking in Python (Part 2)
  - Bootstrap analysis

Even if you've taken these courses you will still find this project challenging unless you use some external *documentation*. Here is a pandas cheat sheet summarizing the basics of pandas DataFrames. (You could also look at the official pandas documentation but be aware that it is *very technical*.)

Finally, know that *Google is your friend* and a good search pattern is **example of ??? in pandas** where **???** is whatever you need to do. For example, if you need to read in a csv file you could search for *example of reading a csv file in pandas*.

Task 2: Instructions

Calculate the yearly proportion of deaths.

- Calculate the proportion of `deaths` per number of `births` and store the result in the new column `yearly["proportion_deaths"]`.
- Extract the rows from clinic 1 into `yearly1` and the rows from clinic 2 into `yearly2`.
- Print out `yearly1`.

Here you need to be able to "pick out" or *subset* rows and columns in the `yearly` DataFrame. How to do that can be glanced from the [pandas cheat sheet] under the headings **Subset observations** and **Subset Variables**.

Task 3: Instructions

Plot the yearly proportion of deaths for both clinics.

- Plot `proportion_deaths` by `year` for the two clinics in a single plot. Use the DataFrame `plot` method.
- Label the plotted lines using the `label` argument to `plot`.
- Save the Axes object returned by the `plot` method into the variable `ax`.
- Change the y-axis label to `"Proportion deaths"`.

For plotting it is easiest to use the `plot` method that is built into DataFrames. To get two lines into the same plot we need to use a trick you might not have seen before. If `df1` and `df2` are two DataFrames you can plot their data together like this:

```
ax = df1.plot(x="col_a", y="col_b",
              label="df1")
df2.plot(x="col_a", y="col_b",
         label="df2", ax=ax)
```

By capturing the `ax` object and giving it as an argument in the plot statement we get both lines in the same plot. The `ax` object can be used to further modify the plot. This is how you would add a label to the y-axis:

```
ax.set_ylabel("A label")
```

Task 4: Instructions

Load in the dataset with the monthly number of deaths for Clinic 1.

- Read in `datasets/monthly_deaths.csv` and assign it to the variable `monthly`. Make sure to tell `read_csv` to parse the `date` column as a date.
- Calculate the proportion of `deaths` per number of `births` and store the result in the new column `monthly["proportion_deaths"]`.
- Print out the first rows in `monthly` using the `head()` method.

The `read_csv` method doesn't automatically detect which columns contain dates. You can tell `read_csv` this by giving a list of the date columns as the optional argument `parse_dates`. For example, if `datasets/my_data.csv` is a csv-file with a date column `date` then you can read it in like this:

```
my_df = pd.read_csv(
    "datasets/my_data.csv",
    parse_dates=["date"])
```

Task 5: Instructions

Plot the monthly proportion of deaths for Clinic 1.

- Plot `proportion_deaths` by `date` for the `monthly` date using the DataFrame `plot` method.
- Save the Axes object returned by the `plot` method into the variable `ax`.
- Change the y-axis label to `"Proportion deaths"`

Task 6: Instructions

Make a plot that highlights the effect of handwashing.

- Split `monthly` into `before_washing` (the rows in `monthly` before `handwashing_start`) and `after_washing` (the rows in `monthly` at and after `handwashing_start`).
- Plot `proportion_deaths` in `before_washing` and `after_washing` into the same plot. Use the DataFrame `plot` method.
- Label the plotted lines using the `label` argument to `plot`.
- Save the Axes object returned by the `plot` method into the variable `ax`.
- Change the y-axis label to `"Proportion deaths"`.

Since the column `monthly["date"]` was read in as a date column we can now compare it to other dates using the comparison operators (<, >=, ==, etc.). For example, to pick out the row exactly at `handwashing_start` we could write:

```
at_washing = monthly[
  monthly["date"] == handwashing_start]
```

Task 7: Instructions

Calculate the average reduction in proportion of deaths due to handwashing.

- Select the column `proportion_deaths` in `before_washing` and put it into `before_proportion`.
- Do the same for `proportion_deaths` in `after_washing` and put it into `after_proportion`.
- Calculate the difference in mean monthly proportion of deaths as mean `after_proportion` minus mean `before_proportion`.

For info on how to calculate the mean of `before_proportion` and `after_proportion` take a look under the heading **Summarize data** in the pandas cheat sheet.

Task 8: Instructions

Make a bootstrap analysis of the difference in mean monthly proportion of deaths.

- `boot_before` and `boot_after` should be sampled with replacement from `before_proportion` and `after_proportion`.
- Append 3000 bootstrapped differences in means to `boot_mean_diff`.
- Calculate a 95% `confidence_interval` as the 2.5% and 97.5% quantiles of `boot_mean_diff`.

A bootstrap analysis is a quick way of getting at the uncertainty of an estimate, in your case the estimate is the `mean_diff` you calculated in task 7. A bootstrap analysis works by *simulating* redoing the data collection by drawing randomly from the data and allowing a value to be drawn many times. Using a pandas column `my_col` (also called a *Series*) this can be done like this:

```
boot_col = my_col.sample(frac=1, replace=True)
```

The estimate is then calculated using `boot_col` instead of `my_col`. This process is repeated a large number of times and the distribution of the bootstrapped estimates represents the uncertainty around the original estimate. If `boot_mean` is a list of bootstrap estimates you can calculate a 95% confidence interval using pandas:

```
pd.Series(boot_mean).quantile([0.025, 0.975])
```

If you want to learn more about how the bootstrap works you should check out the course [Statistical Thinking in Python (Part 2)](#)!

Task 9: Instructions

- Given the data Semmelweis collected, is it `True` or `False` that doctors should wash their hands?

Congratulations, you've made it this far! If you haven't tried it already, you should **check** your project now by clicking the "Check project" button.

Good luck! :)

Project: Dr. Semmelweis and the Discovery of Handwashing Guided

Task 1: Instructions

- Load in the `tidyverse` package.
- Read in `datasets/yearly_deaths_by_clinic.csv` using `read_csv` and assign it to the variable `yearly`.
- Print out `yearly`.

Good to know

The `tidyverse` package automatically loads in the packages `ggplot2`, `dplyr`, and `readr`. Make sure you use `read_csv` from the `readr` package, and not `read.csv`, to read in the data. This project assumes you can manipulate data frames using `dplyr` and make simple plots using `ggplot2`. You can learn these skills in the course Introduction to the Tidyverse. The most relevant exercises are:

- Using mutate to change or create a column
- Adding color to a scatter plot
- Summarizing by continent
- Visualizing median GDP per capita by continent over time

Even if you've taken this course you will still find this project challenging unless you use some external *documentation*. In this project Rstudio's ggplot2 cheat sheet and dplyr cheat sheet can come in handy.

Task 2: Instructions

- Use `mutate` to add the column `proportion_deaths` to `yearly` calculated as the proportion of `deaths` per number of `births`.
- Print out `yearly`.

For an example of how `mutate` works look under **Make New Variables** in the dplyr cheat sheet.

Task 3: Instructions

- Use `ggplot` to make a line plot of `proportion_deaths` by `year` with one line per clinic.
- The lines should have different `colors`.

If you don't remember how to plot line plots with `ggplot` check out the ggplot2 cheat sheet under **Geoms**, **continuous function**.

Task 4: Instructions

- Read in `datasets/monthly_deaths.csv` and assign it to the variable `monthly`.
- Add the column `proportion_deaths` to `monthly` calculated as the proportion of `deaths` per number of `births`.
- Print out the first rows in `monthly` using the `head()` function.

Task 5: Instructions

- Make a line plot of `proportion_deaths` by `date` for the `monthly` data frame using `ggplot`.
- Use the `labs` function to give the x-axis and y-axis *any* prettier labels.

For how to use the `labs` function to add labels check out the ggplot2 cheat sheet under **Labels**.

Task 6: Instructions

- Add a `TRUE/FALSE` column to `monthly` called `handwashing_started` which is `TRUE` for `dates` where obligatory handwashing was enforced.
- Make a line plot of `proportion_deaths` by `date` for the `monthly` data frame using `ggplot`. Make the `color` of the line depend on `handwashing_started`.
- Use the `labs` function to give the x-axis and y-axis *any* prettier labels.

Since the column `monthly$date` is a `Date` column you can now compare it to other `Dates` using the comparison operators (<, >=, ==, etc.). For example, the following would create a new column in `monthly` which is `FALSE` for all `dates` except for the month when handwashing started:

```
monthly <- monthly %>%
  mutate(is_start_month =
    date == handwashing_start)
```

Task 7: Instructions

- Use `group_by` and `summarise` to calculate the `mean` proportion of deaths before and after handwashing was enforced.
- Put the resulting table into `monthly_summary`.

The resulting data frame should look like below, but with 0.????? replaced by the actual numbers.

handwashing_started	mean_proportion_deaths
FALSE	0.?????
TRUE	0.?????

Look under **Group Cases** in the [dplyr cheat sheet] for an example of how `group_by` and `summarise` work together.

Task 8: Instructions

- Use the `t.test` function to calculate a 95% confidence interval around how much dirty hands increases `proportion_deaths`.

A t-test is a simple statistical model for the means of two groups where you have continuous measurements. The two groups we have are monthly `proportion_deaths` *before* handwashing had started and then *after* it was enforced. A t-test produces a lot of numbers, but what we are interested in is the *confidence interval*, here a measure of uncertainty around what the increase in mortality could be due to doctors not washing their hands.

If `df` is a data frame, `outcome` is a numeric column in `df`, and `group` is a `TRUE`/`FALSE` column splitting `df` into two groups, then the following would run a t-test for the two groups:

```
t.test(outcome ~ group, data = df)
```

The tilde (~) should be read as "depends on", and so the above means "assume the `outcome` depends on `group`".

Task 9: Instructions

- Given the data Semmelweis collected, is it `TRUE` or `FALSE` that doctors should wash their hands?

Congratulations, you've made it this far! If you haven't tried it already, you should **check** your project now by clicking the "Check project" button.

Good luck! :)

Project: Phyllotaxis: Draw Flowers Using Mathematics

Task 1: Instructions

Load the package.

- Read the project introduction.
- Load the `ggplot2` package.

Good to know

To complete this project, you will need to know your way around the `ggplot2` package, one of the most useful and downloaded packages of R which should be in the toolbox of every R user. If you're not familiar with `ggplot2`, we recommend that you complete the Introduction to Data Visualization with ggplot2 course first.

The `options()` function used in this notebook is to set global options related to the way that R works. Here, we use it to define the size of the images in the notebook.

You don't need any mathematical background to complete this project, but if you want to know more about phyllotaxis, you can check out this article on Wikipedia.

Task 2: Instructions

Create a scatter plot of 50 points arranged in a circle.

- Read through the given code to create the data frame, `df`.
- Complete the statement to make a scatter plot using `geom_point()`.

Storing a `ggplot()` object in a variable (`p`), will not display an image on the screen, but it allows you to modify the object in a subsequent step. Writing `aes(x, y)` will map the first parameter (`x`) to the x-axis and the second (`y`) to the y-axis. This is the same as writing `aes(x=x, y=y)`.

Helpful links:

- ggplot2 and its geometries
- `seq()` documentation

Task 3: Instructions

Create a scatter plot of spiralzed points.

- Create the variable `points` which defines the number of points to draw and set its value to 500.
- Create another variable called `angle` and set its value to $\pi(3 - \sqrt{5})$.
- Make a scatter plot.

To correctly set `angle` you need to translate the mathematical formula above into R code. If you can't figure it out, check out the hint below.

Task 4: Instructions

Remove plot components and change the background color.

- Remove the grid, ticks, axes titles, and axes text from the spiral plot, and make the background white.

To make these changes to the plot appearance, you need to use the `theme()` function.

There are some predetermined themes in `ggplot2` that would help with the plot's appearance, **but** we will use the `theme()` function instead to have more control over every detail of the plot.

Many colors in R can be specified by name: search for *colors in R* in Google and you will find the complete list.

Helpful links:

- `theme()` documentation

Task 5: Instructions

Make the plot more aesthetically appealing.

- Copy and paste the last lines of code that created the plot from Task 4.
- Change the call to `geom_point()` so that the `size` of points equals 8, the `alpha` (transparency) equals 0.5, and the `color` is `darkgreen`.

After doing this, you should see a plot where all points will have the same `size`, `alpha`, and `color`. The `alpha` parameter will produce a darker color where points overlap.

Colors can be directly specified by name in R (i.e. `"darkgreen"`) or by its *hexadecimal code* (i.e. "#006400"). `alpha` values go from 0 (totally transparent) to 1 (totally opaque), and `size` can take any value, but if it is negative no points will be displayed.

Helpful links:

- Changing the appearance of points in `geom_point()`

Task 6: Instructions

Create a plot that looks like a dandelion.

- Copy and paste the solution from Task 5.
- Within `geom_point()`, map the size aesthetic to the variable `t`, remove the color, and set the shape (outside the aesthetics) to an asterisk (`*`).
- Remove the legend from the plot.

As `size` now depends on the variable `t`, a legend will appear. You can add an argument to `theme()` to remove it. The Cookbook for R has more information about how to do this.

Every `shape` is defined by a number: search for *R plot symbols* on Google, and you'll find many lists of shapes and codes available for `geom_point()`.

Helpful links:

- Cookbook for R

Task 7: Instructions

Create a sunflower plot.

- Copy and paste the solution from Task 6.
- Change the `shape` of all points to *filled triangles*, and change the `color` of the points to yellow.
- Change the color of the background to `"darkmagenta"`.

The code for filled triangles is the 7th prime number. If you're not that into prime numbers, remember that you can always search for R plot symbols on Google.

Task 8: Instructions

Change the value of the angle.

- Change the value of `angle` from the Golden Angle (which is about `2.4`) to `2.0`.
- Copy and paste the code from Task 7 and create the plot.

Task 9: Instructions

Create a magenta flower by changing a few plot parameters.

- Set `angle` to `13*pi/180`, and double the amount of `points`.
- Within `geom_point()`, set `alpha` to 0.1, `shape` to open circles, and `color` to "magenta4". Also, remove `size` from the aesthetics and set it to 80.
- Set the background fill to "white".

Congratulations! You have finished the project!

We removed `size` from the aesthetics because we no longer want the size of the point to depend on the data. There are so many possible images you can create here - play around with different values of `angle`, `points`, and the `ggplot()` parameters to see what you can come up with.

Helpful links:

- Table of different shape values
- Color names (Scroll down)
- Data Visualization with ggplot2 (Part 1)
- Data Visualization with ggplot2 (Part 2)

Project: Rise and Fall of Programming Languages

Task 1: Instructions

Load the dataset and the packages required to analyze the dataset.

- Load the `readr` and `dplyr` packages.
- Load the dataset `datasets/by_tag_year.csv` into a variable named `by_tag_year` using the `read_csv()` function (**not** `read.csv()`).
- Print `by_tag_year`.

Good to know

This project lets you practice the skills from Introduction to the Tidyverse, including filtering, grouping and summarizing data, and visualizing with ggplot2. We recommend that you take that course before starting this project.

This tidyverse cheat sheet will be handy throughout the project (the dplyr and ggplot2 sections, specifically).

The dataset on questions per year was downloaded from the Stack Exchange Data Explorer. The file is also downloadable here.

You will be loading and working with packages throughout the project. You can load packages in R using the `library()` function.

Task 2: Instructions

Create a new column that for each tag-year combination contains the fraction of questions in that year that have that tag.

- Use `mutate()` to add a column called `fraction` to `by_tag_year`, representing `number` divided by `year_total`. Name the new table `by_tag_year_fraction`.
- Print `by_tag_year_fraction`.

Helpful links:

- dplyr's `mutate()` function documentation
- Mutate exercises in the Introduction to the Tidyverse course
- tidyverse cheat sheet

Task 3: Instructions

Filter for R tags.

- Use `filter()` to get only the observations from `by_tag_year_fraction` that represent R, saving them as `r_over_time`.
- Print `r_over_time`.

Helpful links:

- dplyr's `filter()` function documentation
- Filter exercises from the Introduction to the Tidyverse course

Task 4: Instructions

Load the visualization packages and plot fraction of R tags of overall questions over time with a line plot.

- Load the `ggplot2` package.
- Plot `r_over_time` with `year` on the x-axis and `fraction` on the y-axis. Add a `geom_line()` layer to the plot to create a line plot.

Helpful links:

- Line plot exercises from the Introduction to the Tidyverse course

Task 5: Instructions

Filter for the observations where `tag` is R, dplyr, or ggplot2, plot their fraction of overall questions over time with a line plot.

- Combine the tags "r", "dplyr" and "ggplot2" into a vector named `selected_tags` using `c()`.
- Use `filter()` on `by_tag_year_fraction`, along with the `%in%` operator, to get only the subset of tags in `selected_tags`. Name the new table `selected_tags_over_time`.
- Visualize the popularity of these three tags with a line plot in ggplot2 (with `year` on the x-axis and `fraction` on the y-axis) using color to represent tag.

The `%in%` operator is useful for seeing which items in a vector are present in another. Just as you used `tag ==` to filter for one tag before, you could use `tag %in%` to filter for any within a vector.

Helpful links:

- R's `c()` function documentation
- dplyr's `filter()` function documentation
- Filter exercises in the Introduction to the Tidyverse course

- Line plot exercises in the Introduction to the Tidyverse course

Task 6: Instructions

Find and sort the total number of questions for each tag.

- Use the `group_by()` and `summarize()` verbs on `by_tag_year` to find the **total** number of questions for each tag, saving the column as `tag_total`. Then use the `arrange()` verb to sort the table in descending order of the `tag_total` column. Save the result to `sorted_tags`.
- Print `sorted_tags`.

Helpful links:

- Grouping and summarizing exercises in the Introduction to the Tidyverse course
- dplyr's `arrange()` function documentation
- The arrange verb exercises in the Introduction to the Tidyverse course

Task 7: Instructions

Filter for the largest tags and plot them on a line plot.

- Use the `filter()` verb to filter `by_tag_year_fraction` only for the tags in `highest_tags`, which are the six largest tags.
- Create a line plot of the fraction of questions each of these tags made up over time, using color to represent the tag.

You can extract just one column from a table using `$`. As seen in the sample code, `sorted_tags$tag` extracts just the `tag` column.

You can get just the first six items from a vector using `head()`.

Helpful links:

- dplyr's `filter()` function documentation
- Filter exercises in the Introduction to the Tidyverse course
- Line plot exercises in the Introduction to the Tidyverse course

Task 8: Instructions

Filter for specific tags then plot their fraction of overall questions over time with a line plot.

- Combine the tags "android", "ios" and "windows-phone" into a vector named `my_tags` using `c()`.

- Use `filter()` on `by_tag_year_fraction` to get only the subset of tags in `my_tags`. Name the new table `by_tag_subset`.
- Visualize the popularity of these tags with a line plot in ggplot2 (with `year` on the x-axis and `fraction` on the y-axis) using color to represent tag.

If you want to experiment with other tags, note that only tags with at least 1000 questions are included in this dataset so you may need to try a few. If you're having trouble thinking of tags, [try looking at the list on this page for ideas!]

And congratulations on reaching the end of the Project!

Helpful links:

- R's `c()` function [documentation]
- dplyr's `filter()` function [documentation]
- Filter [exercises] in the Introduction to the Tidyverse course
- Line plot [exercises] in the Introduction to the Tidyverse course

```
# Load package

library(readr)

library(dplyr)


# Load dataset

by_tag_year <- read_csv("datasets/by_tag_year.csv")


# Inspect the dataset

print(by_tag_year)


# Add fraction column

by_tag_year_fraction <- by_tag_year %>%

  mutate(fraction = number / year_total)


# Print the new table

print(by_tag_year_fraction)


# Filter for R tags
```

```
r_over_time <- by_tag_year_fraction %>%
  filter(tag == "r")

# Print the new table
print(r_over_time)

# Load ggplot2
library(ggplot2)

# Create a line plot of fraction over time
ggplot(r_over_time) +
  geom_line(aes(x = year, y = fraction))

# A vector of selected tags
selected_tags <- c("r", "dplyr", "ggplot2")

# Filter for those tags
selected_tags_over_time <- by_tag_year_fraction %>%
  filter(tag %in% selected_tags)

# Plot tags over time on a line plot using color to represent tag
ggplot(selected_tags_over_time, aes(x = year,
                                    y = fraction,
                                    color = tag)) +
  geom_line()

# Find total number of questions for each tag
sorted_tags <- by_tag_year %>%
  group_by(tag) %>%
  summarize(tag_total = sum(number)) %>%
  arrange(desc(tag_total))
```

```
# Print the new table
print(sorted_tags)

# Get the six largest tags
highest_tags <- head(sorted_tags$tag)

# Filter for the six largest tags
by_tag_subset <- by_tag_year_fraction %>%
  filter(tag %in% highest_tags)

# Plot tags over time on a line plot using color to represent tag
ggplot(by_tag_subset, aes(x = year,
                          y = fraction,
                          color = tag)) +
  geom_line()

# Get tags of interest
my_tags <- c("android", "ios", "windows-phone")

# Filter for those tags
by_tag_subset <- by_tag_year_fraction %>%
  filter(tag %in% my_tags)

# Plot tags over time on a line plot using color to represent tag
ggplot(by_tag_subset, aes(x = year,
                          y = fraction,
                          color = tag)) +
  geom_line()
```

Project: Visualizing COVID-19

Task 1: Instructions

- Load the `readr`, `ggplot2`, and `dplyr` packages.
- Read in `datasets/confirmed_cases_worldwide.csv` using `read_csv()` and assign it to the variable `confirmed_cases_worldwide`.

Good to know

This project uses concepts found in the following courses.

- Lots of plotting, including log-transforming scales and annotating plots, as covered in Introduction to Data Visualization with ggplot2 and Intermediate Data Visualization with ggplot2.
- Simple manipulation of data frames, as covered in Data Manipulation with dplyr.
- Reading datasets from CSV files, as covered in Introduction to Importing Data in R.

Helpful links:

- tidyverse cheat sheet.
- `library()` function documentation.
- `readr`'s `read_csv()` function documentation.
- Importing data using `read_csv()` is covered in Introduction to Importing Data in R ch2ex2.

Task 2: Instructions

- Using `confirmed_cases_worldwide`, draw a ggplot with aesthetics `cum_cases` (y-axis) versus `date` (x-axis).
- Make it a line plot by adding a line geometry.
- Set the y-axis label to `"Cumulative confirmed cases"`.

Helpful links:

- `ggplot2`'s `geom_line()` function documentation.
- `ggplot2`'s `ylab()` function documentation.
- Drawing single line plots is covered in Introduction to Data Visualization with ggplot2 ch3ex14.
- Setting axis labels is covered in Introduction to Data Visualization with ggplot2 ch2ex11.

Task 3: Instructions

- Read in the dataset for confirmed cases in China and the rest of the world from `datasets/confirmed_cases_china_vs_world.csv`, assigning to `confirmed_cases_china_vs_world`.
- Use `glimpse()` to explore the structure of `confirmed_cases_china_vs_world`.
- Draw a ggplot of `confirmed_cases_china_vs_world`, assigning to `plt_cum_confirmed_cases_china_vs_world`.
- Add a line layer. Add aesthetics within this layer: `date` on the x-axis, `cum_cases` on the y-axis, then group and color the lines by `is_china`.

Helpful links:

- `readr`'s `read_csv()` function documentation.
- `tibble`'s `glimpse()` function documentation.
- `ggplot2`'s `geom_line()` function documentation.
- Importing data using `read_csv()` is covered in Introduction to Importing Data in R ch2ex2.
- Drawing multiple lines is covered in Introduction to Data Visualization with ggplot2 ch3ex15.
- Adding aesthetics to a geometry is covered in Introduction to Data Visualization with ggplot2 ch1ex11.

Task 4: Instructions

A dataset of World Health Organization events , `who_events` is provided. Modify the plot `plt_cum_confirmed_cases_china_vs_world` as follows.

- Add a vertical line layer with the `xintercept` aesthetic mapped to `date`. Use the `who_events` data, and make it a dashed line.
- Add a text layer with `x` mapped to `date` and `label` mapped to `event`. Place the labels at `100000` on the y-axis. Use the `who_events` data again.

Helpful links:

- `geom_vline()` function documentation.
- `geom_text()` function documentation.
- Using `geom_vline()` for vertical lines is covered in Introduction to Data Visualization with ggplot2 ch4ex12.
- Setting the `data` argument to a geometry is covered in the video Introduction to Data Visualization with ggplot2 ch2ex1, around 1m30s.
- Setting the `linetype` is covered (in the context of changing themes) in Introduction to Data Visualization with ggplot2 ch4ex3.
- Using `geom_text()` for vertical lines is covered in Introduction to Data Visualization with ggplot2 ch4ex11.

Task 5: Instructions

- Filter rows of `confirmed_cases_china_vs_world` for observations of China where the date is greater than or equal to `"2020-02-15"`, assigning to `china_after_feb15`.
- Using `china_after_feb15`, draw a line plot of `cum_cases` versus `date`.
- Add a smooth trend line, calculated the using linear regression method, without the standard error ribbon.

Helpful links:

- `dplyr`'s `filter()` function documentation.
- `ggplot2`'s `geom_smooth()` function documentation.
- Filtering data frames is covered in Data Manipulation with dplyr ch1ex6.
- Adding smooth trend lines is covered in Intermediate Data Visualization with ggplot2 ch1ex2.

Task 6: Instructions

- Filter rows of `confirmed_cases_china_vs_world` for observations of Not China, assigning to `not_china`.
- Using `not_china`, draw a line plot of `cum_cases` versus `date`, assigning to `plt_not_china_trend_lin`.
- Add a smooth trend line, calculated the using linear regression method, without the standard error ribbon.

Helpful links:

- `dplyr`'s `filter()` function documentation.
- `ggplot2`'s `geom_smooth()` function documentation.
- Filtering data frames is covered in Data Manipulation with dplyr ch1ex6.
- Adding smooth trend lines is covered in Intermediate Data Visualization with ggplot2 ch1ex2.

Task 7: Instructions

- Modify the plot, `plt_not_china_trend_lin`, to use a logarithmic scale on the y-axis.

Helpful links:

- `scale_y_log10()` function documentation.
- Logarithmic axis scales are covered in Intermediate Data Visualization with ggplot2 ch2ex7.

Task 8: Instructions

Code to import data on confirmed cases by country is provided. Chinese data has been excluded to focus on the rest of the world.

- *Look at the output of `glimpse()` to see the structure of `confirmed_cases_by_country`.*
- Using `confirmed_cases_by_country`, group by `country`.
- Summarize to calculate `total_cases` as the maximum value of `cum_cases`.
- Get the top seven rows by `total_cases`.

Helpful links:

- `dplyr`'s `group_by()` function documentation.
- `dplyr`'s `summarize()` function documentation.
- `dplyr`'s `top_n()` function documentation.
- Calculating summary statistics with `summarize()` is covered in Data Manipulation with dplyr ch2ex6.
- Combining `group_by()` with `summarize()` is covered in Data Manipulation with dplyr ch2ex7.
- Getting the top N results is covered in Data Manipulation with dplyr ch2ex10.
- Using all three functions together is covered in Data Manipulation with dplyr ch2ex12.

Task 9: Instructions

- Read in the dataset for confirmed cases in China and the rest of the world from `datasets/confirmed_cases_top7_outside_china.csv`, assigning to `confirmed_cases_top7_outside_china`.
- Use `glimpse()` to explore the structure of `confirmed_cases_top7_outside_china`.
- Using `confirmed_cases_top7_outside_china`, draw a line plot of `cum_cases` versus `date`, grouped and colored by `country`.
- Set the y-axis label to `"Cumulative confirmed cases"`.

Helpful links:

- `readr`'s `read_csv()` function documentation.
- `tibble`'s `glimpse()` function documentation.
- `ggplot2`'s `geom_line()` function documentation.
- Importing data using `read_csv()` is covered in Introduction to Importing Data in R ch2ex2.
- Drawing multiple lines is covered in Introduction to Data Visualization with ggplot2 ch3ex15.
- Setting axis labels is covered in Introduction to Data Visualization with ggplot2 ch2ex11.

```
# Load the readr, ggplot2, and dplyr packages

library(readr)

library(ggplot2)

library(dplyr)


# Read datasets/confirmed_cases_worldwide.csv into
confirmed_cases_worldwide

confirmed_cases_worldwide <-
read_csv("datasets/confirmed_cases_worldwide.csv")


# Print out confirmed_cases_worldwide

confirmed_cases_worldwide


# Draw a line plot of cumulative cases vs. date

# Label the y-axis

ggplot(confirmed_cases_worldwide, aes(date, cum_cases)) +

  geom_line() +

  ylab("Cumulative confirmed cases")


# Read in datasets/confirmed_cases_china_vs_world.csv

confirmed_cases_china_vs_world <-
read_csv("datasets/confirmed_cases_china_vs_world.csv")


# See the result

glimpse(confirmed_cases_china_vs_world)


# Draw a line plot of cumulative cases vs. date, grouped and colored by is_china

# Define aesthetics within the line geom

plt_cum_confirmed_cases_china_vs_world <-
ggplot(confirmed_cases_china_vs_world) +

  geom_line(aes(date, cum_cases, group = is_china, color = is_china)) +

  ylab("Cumulative confirmed cases")
```

```
# See the plot
plt_cum_confirmed_cases_china_vs_world

who_events <- tribble(
  ~ date, ~ event,
  "2020-01-30", "Global health\nemergency declared",
  "2020-03-11", "Pandemic\ndeclared",
  "2020-02-13", "China reporting\nchange"
) %>%
  mutate(date = as.Date(date))

# Using who_events, add vertical dashed lines with an xintercept at date
# and text at date, labeled by event, and at 100000 on the y-axis
plt_cum_confirmed_cases_china_vs_world +
  geom_vline(aes(xintercept = date), data = who_events, linetype = "dashed") +
  geom_text(aes(date, label = event), data = who_events, y = 1e5)

# Filter for China, from Feb 15
china_after_feb15 <- confirmed_cases_china_vs_world %>%
  filter(is_china == "China", date >= "2020-02-15")

# Using china_after_feb15, draw a line plot cum_cases vs. date
# Add a smooth trend line using linear regression, no error bars
ggplot(china_after_feb15, aes(date, cum_cases)) +
  geom_line() +
  geom_smooth(method = "lm", se = FALSE) +
  ylab("Cumulative confirmed cases")

# Filter confirmed_cases_china_vs_world for not China
not_china <- confirmed_cases_china_vs_world %>%
  filter(is_china == "Not China")
```

```
# Using not_china, draw a line plot cum_cases vs. date
# Add a smooth trend line using linear regression, no error bars
plt_not_china_trend_lin <- ggplot(not_china, aes(date, cum_cases)) +
  geom_line() +
  geom_smooth(method = "lm", se = FALSE) +
  ylab("Cumulative confirmed cases")

# See the result
plt_not_china_trend_lin

# Modify the plot to use a logarithmic scale on the y-axis
plt_not_china_trend_lin +
  scale_y_log10()

# Run this to get the data for each country
confirmed_cases_by_country <-
read_csv("datasets/confirmed_cases_by_country.csv")
glimpse(confirmed_cases_by_country)

# Group by country, summarize to calculate total cases, find the top 7
top_countries_by_total_cases <- confirmed_cases_by_country %>%
  group_by(country) %>%
  summarize(total_cases = max(cum_cases)) %>%
  top_n(7, total_cases)

# See the result
top_countries_by_total_cases

# Read in the dataset from datasets/confirmed_cases_top7_outside_china.csv
confirmed_cases_top7_outside_china <-
read_csv("datasets/confirmed_cases_top7_outside_china.csv")
```

```
# Glimpse the contents of confirmed_cases_top7_outside_china
glimpse(confirmed_cases_top7_outside_china)


# Using confirmed_cases_top7_outside_china, draw a line plot of
# cum_cases vs. date, grouped and colored by country
ggplot(confirmed_cases_top7_outside_china, aes(date, cum_cases, color =
country, group = country)) +
  geom_line() +
  ylab("Cumulative confirmed cases")
```

Exploring 67 Years of LEGO

Task 1: Instructions

Welcome to the Python project **Exploring 67 years of LEGO**!

If you haven't done a DataCamp project before you should check out the Intro to Projects first to learn about the interface. In this project, you also need to know your way around `pandas` DataFrames and it's recommended that you take a look at the course Data Manipulation with pandas. For your first task:

- Read the first paragraph in the notebook to familiarize yourself with the topic!
- Feel free to poke around the interface.
- When you are finished, click on the **Next Task** button at the bottom.

At any point in the project, you can click on the **Check Project** button at the bottom to test whether your output matches the solution.

- If all the tests pass, the numbered task circles on the side will turn green.
- If some tests fail, the incorrect tasks will turn orange.

You can view the test results in the sidebar to understand what failed and update your code accordingly. If you are unable to get all the tests to pass despite repeated attempts, you can click on the **Hint** button to get a useful hint.

Task 2: Instructions

- Import `pandas` and alias it as `pd`.
- Read the csv file located in the path `'datasets/colors.csv'` into a DataFrame named `colors`.
- Inspect the first five rows of the resulting `colors` DataFrame.

Task 3: Instructions

- Create a variable named `num_colors` that counts the number of distinct colors.
- Print it out.

Task 4: Instructions

- Summarize colors based on their transparency.
- Save the result as a variable named `colors_summary`.
- Print out `colors_summary`.

Executing your code should result in the table shown below.

is_trans	id	name	rgb
f	107	107	107
t	28	28	28

Task 5: Instructions

- Read the data in `datasets/sets.csv` as a DataFrame named `sets`.
- Create a summary of the average number of parts per year and save it as `parts_by_year`.
- Plot the average number of parts per year.

The first few rows of `parts_by_year` should resemble the table shown below:

year	num_parts
1950	10.14
1953	16.50
1954	12.36
1955	36.86
1956	18.50

Task 6: Instructions

- Create a summary of the number of distinct themes shipped by year.
- Save it as a `DataFrame` named `themes_by_year`.
- Print the first couple of rows in `themes_by_year`.

The first few rows of your data should resemble the table shown below.

year	theme_id
1950	2
1953	1
1954	2
1955	4
1956	3

Note: In this step you may need to use an aggregation function that you have yet to encounter. The [documentation here]() may be of assistance. You will want to find a way to return the number of unique values in each group.

Task 7: Instructions

In 1999, Lego expanded into licensed sets with the introduction of Star Wars themed sets. In that year, how many unique themes were released?

- Assign your answer to the variable `num_themes`.
- Print `num_themes`.

Project: Word Frequency in Moby Dick

Task 1: Instructions

Import the four Python modules you'll use in this project:

- `requests` to fetch the html file that contains the book.
- `BeautifulSoup`, from the `bs4` module, to extract the words from the html file.
- `nltk` to process our text.
- `Counter`, from the `collections` module, to analyze the frequncy of our processed words.

Good to know

To complete this project, you need to know how to import web data into Python and how to work with natural language text. Before starting this project we recommend that you have completed the following courses:

- Intermediate Importing Data in Python
- Introduction to Natural Language Processing in Python

This Project is based on a live screencast by DataCamp's own Hugo Bowne-Anderson. When you've finished the Project, or if you get stuck, do check out the screencast with Hugo's solution (the screencast starts 12 minutes into the video). You can also find Hugo's solution notebook here.

Task 2: Instructions

Request the Moby Dick HTML file.

- Get the following URL and assign it to `r`:

```
https://s3.amazonaws.com/assets.datacamp.com/production/project_147/
datasets/2701-h.htm
```

- Extract the text from `r` and assign it to `html`.
- Print out the first 2000 characters in `html`.

For a guide to how you use `requests` to download a webpage check out the `request` kickstart guide.

Note that the HTML file you are asked to request in this task is a cashed version of this file from Project Gutenberg.

Task 3: Instructions

Extract the text from the html version of the book.

- Create a `BeautifulSoup` object from `html` as assign it to `soup` .
- Extract the text from the `soup` and assign it to `text`.
- Print out the `text` starting from character number 32000 until character number 34000.

For how to get started using `BeautifulSoup` to read and extract the text check out [this quick start](#).

Task 4: Instructions

Tokenize the Moby Dick text.

- Initialize a regex tokenizer object `tokenizer` using `nltk.tokenize.RegexpTokenizer`, passing in a regular expression that will split the text into individual words.
- Use the correct method of your new `tokenizer` object to tokenize `text` and assign the resulting list of words to `tokens`.
- Print out the first 8 words / tokens.

For how to use the `nltk.tokenize.RegexpTokenizer` function, please see [the example in the `nltk` documentation](#).

Task 5: Instructions

Convert the words / tokens to lowercase.

- Loop through the words in `tokens`, make them lowercase, and store them in a list called `words`.
- Print out the first 8 `words` to make sure they are all lowercase.

Task 6: Instructions

Load in the English stop words.

- Load in the English stop words from `nltk` and assign them to `sw`.
- Print out the first 8 stop words in `sw`.

See the `nltk` documentation for [how to load in the stop words](#).

Before being able to load in the stop words, you have to download them using the command:

```
nltk.download('stopwords')
```

But in this project, this step has already been done for you.

Task 7: Instructions

Create a new list with the words from Moby Dick, where stop words have been removed.

- Create a list `words_ns` that contains all words that are in `words` but *not* in `sw`.
- Print out the five first words in `words_ns`.

Task 8: Instructions

- Inintialize a `Counter` object called `count` using our `words_ns` list.
- Use the corresponding method to return the 10 most common words and their counts, assigning the result to `top_ten`.
- Print `top_ten`.

Note: *This step was recently updated. If you are receiving feedback related to the variable `freqdist`, you are likely using an outdated notebook. This can be avoided by refreshing the notebook (and losing all your progress), or by adding the following code to your solution:*

```
freqdist = nltk.FreqDist(words_ns)
```

Task 9: Instructions

- Take a look at the counts you printed in the last task and assign the most common word in Moby Dick to `most_common_word`.

If you want to know more

This Project is based on a live screencast by DataCamp's own Hugo Bowne-Anderson. When you've finished the Project, or if you get stuck, do check out the screencast with Hugo's solution (the screencast starts 12 minutes into the video). You can also find Hugo's solution notebook here.

```
# Importing requests, BeautifulSoup, nltk, and Counter

import requests
```

```
from bs4 import BeautifulSoup

import nltk

from collections import Counter


# Getting the Moby Dick HTML

r = requests.get('https://s3.amazonaws.com/assets.datacamp.com/production/
project_147/datasets/2701-h.htm')


# Setting the correct text encoding of the HTML page

r.encoding = 'utf-8'


# Extracting the HTML from the request object

html = r.text


# Printing the first 2000 characters in html

print(html[0:2000])


# Creating a BeautifulSoup object from the HTML

soup = BeautifulSoup(html, "html.parser")


# Getting the text out of the soup

text = soup.get_text()


# Printing out text between characters 32000 and 34000

print(text[32000:34000])


# Creating a tokenizer

tokenizer = nltk.tokenize.RegexpTokenizer('\w+')


# Tokenizing the text

tokens = tokenizer.tokenize(text)
```

```
# Printing out the first 8 words / tokens
tokens[0:8]

# Create a list called words containing all tokens transformed to lower-case
words = [token.lower() for token in tokens]

# Printing out the first 8 words / tokens
words[:8]

# Getting the English stop words from nltk
sw = nltk.corpus.stopwords.words('english')

# Printing out the first eight stop words
sw[:8]

# Create a list words_ns containing all words that are in words but not in sw
words_ns = [word for word in words if word not in sw]

# Printing the first 5 words_ns to check that  stop words are gone
words_ns[:5]

# Initialize a Counter object from our processed list of words
count = Counter(words_ns)

# Store 10 most common words and their counts as top_ten
top_ten = count.most_common(10)

# Print the top ten words and their counts
print(top_ten)

# What's the most common word in Moby Dick?
most_common_word = 'whale'
```

Project: Risk and Returns: The Sharpe Ratio

Task 1: Instructions

Read in the stock data for Facebook, Amazon and the S&P 500.

- Load in the stock data from `datasets/stock_data.csv` and assign it to `stock_data`.
- Load in the benchmark data from `datasets/benchmark_data.csv` and assign it to `benchmark_data`.
- When reading in the data change the `parse_dates` parameter to set the `'Date'` column to `datetime64`, set the `index_col` parameter to set the `'Date'` column as the index, and use `.dropna()` to get rid of missing values.

A goal here is to read in the data as a *time series* by moving the `Date` column to the `index`.

You can load the data using `pd.read_csv()`. Make sure you use the `parse_dates` parameter to set the `'Date'` column to `datetime64`, and the `index_col` parameter to set the same column as index.

Good to know

This project assume you have completed the DataCamp course [Importing and Managing Financial Data using Python]. If you haven't completed this course, we recommend that you do so first.

Task 2: Instructions

Take a peek at the data you loaded in the last task.

- Display a summary of each `DataFrame`'s content using `.info()`
- Show the first few lines of each `DataFrame` using `.head()`

You can plot the `.info()` summary without using `print`, but `.head()` requires `print()` unless it's the last statement in the cell.

Task 3: Instructions

Plot and summarize the `stock_data`.

- Use the `pandas` `.plot()` method on `stock_data` to show a line plot.

- Set the parameter `subplots=True` to show two plots since the stock prices are at different levels.
- Set `'Stock Data'` as the `title` for the plot.
- Apply the `.describe()` method to the stock data to produce summary statistics.

You can use the `.plot()` method parameter `title` to display the title. If you use a semicolon after the `.plot()` command you avoid the output of the reference to the `matplotlib` object that is returned by this command.

Task 4: Instructions

Plot and summarize the `benchmark_data`.

- Use the `pandas .plot()` method on `benchmark_data` to show a line plot.
- Set `'S&P 500'` as the `title` for the plot.
- Apply the `.describe()` method to the benchmark data to produce summary statistics.

Task 5: Instructions

Calculate, plot and summarize the `stock_data` *returns*.

- Apply `pandas` method `.pct_change()` method to the `stock_data` to calculate the daily returns.
- Use the `.plot()` method on the result to show a line plot of the daily returns.
- Apply the `.describe()` method to your daily returns to take a look at summary statistics.

The `pandas` method `.pct_change()` calculates the relative change from one value to the next in a `DataFrame`.

Task 6: Instructions

Calculate, summarize, and plot daily returns for the `benchmark_data`.

- Select the S&P 500 prices as a `Series` from the `benchmark_data` using single brackets `[]` and apply `.pct_change()` as in he last task.
- Use`.plot()` to display a line plot of the result.
- Take a look at the summary statistics using `.describe()`

Compare the summary stats for the index to those for the individual stocks. What do you notice?

Task 7: Instructions

Calculate, `plot` and `describe` the difference between `stock_returns` and `sp_returns`.

- Use the `.sub()` method to subtract the `sp_returns` from the `stock_returns` and assign the resulting `DataFrame` to `excess_returns`. Make sure to set the parameter `axis=0` to align the dates for both time series.
- Calculate `excess_returns` summary statistics using `.describe()`

Task 8: Instructions

Calculate and plot the mean of `excess_returns`.

- Calculate the average of `excess_returns` using `.mean()` and assign the result to `avg_excess_return`
- Plot the result using the `pandas` method `.plot.bar()` and set `'Mean of the Return Difference'` as the `title`.

Task 9: Instructions

Calculate and visualize the standard deviation of `excess_returns`.

- Calculate the standard deviation of `excess_returns` using `.std()` and assign the result to `sd_excess_return`.
- Visualize the result as a bar chart and set `'Standard Deviation of the Return Difference'` as the `title` for the plot.

Task 10: Instructions

Use `avg_excess_return` and `sd_excess_return` to calculate the Sharpe ratio, then annualize.

- Apply `.div()` to divide `avg_excess_return` by `sd_excess_return` and assign the result to `daily_sharpe_ratio`.
- Calculate the square root of 252 using `np.sqrt()` and assign the result to the variable `annual_factor`.
- Use `.mul()` to multiply `daily_sharpe_ratio` by `annual_factor` and assign the result to `annual_sharpe_ratio`.
- Display the result as a bar plot, setting `'Annualized Sharpe Ratio: Stocks vs S&P 500'` as the `title`.

Task 11: Instructions

- Select the stock you would have picked in 2016 based on the Sharpe Ratio by setting either `buy_amazon` or `buy_facebook` to `True`.

```
# Importing required modules

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt


# Settings to produce nice plots in a Jupyter notebook

plt.style.use('fivethirtyeight')

%matplotlib inline


# Reading in the data

stock_data = pd.read_csv('datasets/stock_data.csv',

    parse_dates=['Date'],

    index_col='Date'

    ).dropna()

benchmark_data = pd.read_csv('datasets/benchmark_data.csv',

    parse_dates=['Date'],

    index_col='Date'

    ).dropna()


# Display summary for stock_data

print('Stocks\n')

stock_data.info()

print(stock_data.head())
```

```
# Display summary for benchmark_data
print('\nBenchmarks\n')
benchmark_data.info()
benchmark_data.head()

# visualize the stock_data
stock_data.plot(title='Stock Data', subplots=True);

# summarize the stock_data
stock_data.describe()

# plot the benchmark_data
benchmark_data.plot();

# summarize the benchmark_data
benchmark_data.describe()

# calculate daily stock_data returns
stock_returns = stock_data.pct_change()

# plot the daily returns
stock_returns.plot();

# summarize the daily returns
stock_returns.describe()
```

```
# calculate daily benchmark_data returns
sp_returns = benchmark_data['S&P 500'].pct_change()

# plot the daily returns
sp_returns.plot();

# summarize the daily returns
sp_returns.describe()

# calculate the difference in daily returns
excess_returns = stock_returns.sub(sp_returns, axis=0)

# plot the excess_returns
excess_returns.plot();

# summarize the excess_returns
excess_returns.describe()

# calculate the mean of excess_returns
avg_excess_return = excess_returns.mean()

# plot avg_excess_returns
avg_excess_return.plot.bar(title='Mean of the Return Difference');

# calculate the standard deviations
sd_excess_return = excess_returns.std()
```

```
# plot the standard deviations

sd_excess_return.plot.bar(title='Standard Deviation of the Return Difference');


# calculate the daily sharpe ratio

daily_sharpe_ratio = avg_excess_return.div(sd_excess_return)


# annualize the sharpe ratio

annual_factor = np.sqrt(252)

annual_sharpe_ratio = daily_sharpe_ratio.mul(annual_factor)


# plot the annualized sharpe ratio

annual_sharpe_ratio.plot.bar(title='Annualized Sharpe Ratio: Stocks vs S&P
500');


# Uncomment your choice.

buy_amazon = True

# buy_facebook = True
```

Project: Bad passwords and the NIST guidelines

Task 1: Instructions

Load in and inspect the usernames and passwords of the fictional users.

- Import the `tidyverse` package.
- Use `read_csv` to load in the user data from `datasets/users.csv` and put it into the variable `users`.
- Count the number of users.
- Show the first 12 rows in `users` using the `head` function.

Loading the `tidyverse` package will also load the `stringr` and `readr` packages that you'll use in this project.

Make sure to use the `read_csv` (with an *underscore*) function from `tidyverse` to read in the data. The `read.csv` function which is built into R has a number of problems which the new `read_csv` function avoids.

Good to know

To complete this project, you need to know how to manipulate strings in R using the `stringr` package. You also need to know how to work with regular expressions. If you don't have experience with this, we recommend you completed the course [String Manipulation in R with stringr]() first.

An excellent companion while doing this project is the `stringr` cheat sheet from Rstudio which you can download here.

Task 2: Instructions

Flag the passwords that are too short.

- Add the column `length` to `users` which should list the number of characters in each `password`.
- Flag the users with too short passwords by adding the column `users$too_short` which should be `TRUE` when `users$length` is less than 8.
- Count the number of users with passwords that are too short.
- Show the first 12 rows in `users`.

To solve this task, you need to be able to figure out the number of characters in each password. Check the `stringr` cheat sheet under **Manage Lengths** for a function that does just that.

A trick for how to count how many `TRUE` values there are in a column is to use the `sum` function. The `TRUE` values will be counted as `1` and the `FALSE` as `0`, and the `sum` will equal the number of `TRUE` values.

Task 3: Instructions

Load in the list with the 10,000 most common passwords.

- Read in `datasets/10_million_password_list_top_10000.txt`as a vector and put it in the variable `common_passwords`.
- Take a look at the top 100 common passwords.

The passwords are stored in a plain text file with one password per row. To read this in as a vector you need to use the `read_lines` function which takes the path to the dataset as the first argument.

Task 4: Instructions

Flag the `user` passwords that are among the top 10,000 used passwords.

- Flag common `user` passwords by adding the column `users$common_password` which should be `TRUE` when a `password` is one of the `common_passwords`.
- Count the number of users using common passwords.
- Show the first 12 rows in `users`.

Task 5: Instructions

Flag the passwords that are among the 10,000 most common English words.

- Read in `datasets/google-10000-english.txt` as a vector and put it in the variable `words`.
- Flag `user` passwords that are common words by adding the column `users$common_word` which should be `TRUE` when a `password` is one of the `words`. The comparison should be *case-insensitive*.
- Count the number of users using common words as passwords.
- Show the first 12 rows in `users`.

A trick to make a comparison case insensitive is to simply use `str_to_lower` to transform the passwords to lowercase when checking whether they are in `words` or not.

Task 6: Instructions

Flag passwords that are the same as the users first or last name.

- Extract users first names from `users$user_name` into the new column `users$first_name`.
- Similarly, extract last names into the new column `users$last_name`.
- Add the column `users$uses_name` which should be `TRUE` when a `password` is the same as each users' first or last name.
- Count the number of users using names as passwords.
- Show the first 12 rows in `users`.

To extract the first and last names you can use the `str_extract` function. Check out [the `stringr` cheat sheet](#) under **Subset String** for more info about that function.

You will need to supply `str_extract` with a regular expression matching what you want to extract. Again, check the second page of the cheat sheet for a reminder of what can go into a regular expression. For this task, you'll find use for the match-any-word-character `\\w`, and the word anchors `^` and `$`.

Task 7: Instructions

Flag the passwords that contain 4 or more repeated characters.

- Transform `users$password` into a list of vectors of single characters (`"abc"` → `c("a", "b", "c")`) and assign it to `split_passwords`.
- Use `sapply` to go through each `split_password` and calculate the max number of repeats. Put the result back into the column `users$max_repeats`.
- Add the column `users$too_many_repeats` which should be TRUE when a `password` has 4 or more repeated characters.
- Take a look at only the `users` with too many repeats.

There is a function in R called `rle` (standing for *run length encoding*) that is helpful here. Given a vector, it calculates the number of consecutive elements.

```
x <- c("a", "a", "b", "c", "c", "c")
n <- rle(x)$lengths
# n is now c(2, 1, 3)
```

A problem is that the passwords are strings (`"abc"`) and not vectors of single characters (`c("a", "b", "c")`). To fix this you first have to split the password strings into a list of vectors. Here is an example of how to do that:

```
x <- c("abc", "123")
l <- str_split(x, "")
# l is now list(c("a", "b", "c"),
#               c("1", "2", "3"))
```

Task 8: Instructions

Flag *all* the bad passwords.

- Add the column `users$bad_password` which should be `TRUE` when a password is bad according to `too_short`, `common_password`, `common_word`, `uses_name`, or `too_many_repeats`.
- Count the number of users with bad passwords.
- Show the first 100 bad passwords in `users`.

Remember that you can use the *or* operator `|` to check if this *or* that is `TRUE`.

Task 9: Instructions

- Assign a new password to `new_password` that passes the NIST rules you've implemented in this project.

If you want to know more

- You can read the full NIST Special Publication 800-63B online.
- This blog post also gives you a summary of 800-63B.
- Here is an article explaining where the 10,000 common passwords you used in this project come from.
- Finally, some advice on how to come up with a good password.

Project: Exploring the Kaggle Data Science Survey

Task 1: Instructions

Load the data and look at the first 10 responses.

- Load the `tidyverse` package.
- Using `read_csv`, load `datasets/kagglesurvey.csv` and assign it to the variable `responses`.
- Print the first 10 entries of `responses`.

This project was updated on December 26, 2019. If you started the project before that date, please click the circular arrow in the bottom-right corner of the screen to reset the project. If you would like to save your code, download your project before resetting it.

Good to Know

The `tidyverse` package automatically loads in `dplyr`, `ggplot2`, `readr`, `tidyr`, and a few other helpful packages. Learn more about the `tidyverse` here.

Before starting this project you should be comfortable manipulating data frames and have some experience working with the `tidyverse` packages `dplyr`, `tidyr`, and `ggplot2`. We recommend that you have completed at least one of the following courses:

- Introduction to the Tidyverse
- Introduction to Data Visualization with ggplot2

Code examples will frequently utilize the pipe operator (`%>%`). More information about using the pipe operator in R can be found here.

Task 2: Instructions

Split the tools each respondent uses into separate rows.

- Print the tools and languages used by the first respondent (found in column 2: `WorkToolsSelect`).
- Create a new data frame called, `tools`, by using `str_split()` to split the `WorkToolsSelect` column at the commas, then `unnest()` the new column to fill `work_tools`.
- View the first 6 rows of tools.

Helpful links:

- `str_split()`

- `unnest()`

Task 3: Instructions

Find the number of respondents that use each language or tool.

- Create a new data frame, `tool_count`, by grouping the `tools` data by `work_tools`, then use `summarise()` to calculate the number of responses within each group.
- Sort `tool_count` so that the most popular tools are at the top.
- Print the first 6 results of `tool_count`.

Your final data frame should have only two columns: `work_tools` and the count of users.

Helpful links:

- Data Transformation with `dplyr` cheet sheet
- `n()`

Task 4: Instructions

Create a bar chart that displays tool popularity.

- Use `ggplot2` to create a bar chart of `work_tools` in the `tool_count` data frame. Use `fct_reorder()` to arrange the bars so that the tallest are on the far right.
- Rotate the bar labels 90 degrees.

Helpful links:

- `fct_reorder()`
- ggplot2 cookbook
- `ggplot` axis: set and rotate text labels
- modifying a theme in ggplot2

Task 5: Instructions

Calculate the number of respondents that use R, Python, and both tools.

- Create a new column called `language_preference` which should be set to:
  - **"R"** if `WorkToolsSelect` contains "R" but not "Python".
  - **"Python"** if `WorkToolsSelect` contains "Python" but not "R".
  - **"both"** if `WorkToolsSelect` contains both "R" and "Python".
  - **"neither"** if `WorkToolsSelect` contains neither "R" nor "Python".
- Print the first 6 rows of `debate_tools`.

While you can use nested `ifelse()` statements to accomplish this task, we recommend using the `case_when()` function from `dplyr`. You can find more information about `case_when()` here.

To determine if the comma-separated string in the `WorkToolsSelect` column contains R or Python, we recommend using `str_detect()` from `stringr`. More information on `str_detect()` can be found here.

Task 6: Instructions

Calculate total number of users that use R, Python, or both, and plot the results.

- Group `debate_tools` by `language_preference` and then use `summarise()` to calculate the number of each response.
- Remove the rows of respondents that use "neither" R nor Python.
- Create a bar chart of language preference counts using ggplot.

To remove a row from a data frame, you can use `dplyr`'s `filter()` function.

Task 7: Instructions

Find language recommendations for users that use R, Python, or both languages.

- Group `debate_tools` by `language_preference` and `LanguageRecommendationSelect`. Summarise the number of recommendations for each language within each group and include only the top four most common recommendations for each language preference.

To only keep the top four most common recommendations for each language, we'll have to arrange `language_preference` from most popular to least popular, then add a column that counts the row number within each group (using `row_number()`) and filter for the row numbers that are equal to and less than four.

Learn more about `row_number()` and the other `dplyr` ranking functions here.

Task 8: Instructions

Create a faceted plot showing the top four language recommendations from users of R, Python, both, and neither.

- Use the `ggplot` function `facet_wrap()` to create a faceted plot of recommendation frequency. You should get four sub-plots, one for each type of value in the `language_preference` column.

The `facet_wrap()` function creates a series of plots; you just need to tell it how it should split them. For instance, if you had data about different countries and you wanted a separate plot for each country, you could do something like this:

```
ggplot(my_data, aes(x = x, y = y)) +
    geom_bar(stat = "identity") +
    facet_wrap(~country)
```

Task 9: Instructions

Would R-users find the following statement TRUE or FALSE?

- R is the language I recommend for new data scientists.

Congratulations! You've made it to the end of this project!

If you haven't already, try to *check* your project by clicking the **Check Project** button.

If you're looking to learn more from this dataset, you can find the questions we explored here and many others on Kaggle. You can also explore my analysis of the full dataset as well as the analysis by many other talented data lovers.

Good luck! :)

Load necessary packages

library(tidyverse)

Load the data

responses <- read_csv('datasets/kagglesurvey.csv')

Print the first 10 rows

head(responses, n = 10)

```
# Printing the first respondent's tools and languages
responses[1, 2]

# Add a new column, and unnest the new column
tools <- responses  %>%
    mutate(work_tools = str_split(WorkToolsSelect, ","))  %>%
    unnest(work_tools)

# View the first 6 rows of tools
head(tools)

# Group the data by work_tools, summarise the counts, and arrange in descending order
tool_count <- tools  %>%
    group_by(work_tools)  %>%
    summarise(count = n())  %>%
    arrange(desc(count))

# Print the first 6 results
head(tool_count)

# Create a bar chart of the work_tools column, most counts on the far right
ggplot(tool_count, aes(x = fct_reorder(work_tools, count), y = count)) +
    geom_bar(stat = "identity") +
    theme(axis.text.x  = element_text(angle=90, vjust=0.5, hjust= 1))
```

```
# Create a new column called language preference

debate_tools <- responses  %>%

    mutate(language_preference = case_when(

        str_detect(WorkToolsSelect, "R") & ! str_detect(WorkToolsSelect, "Python") ~ "R",

        str_detect(WorkToolsSelect, "Python") & ! str_detect(WorkToolsSelect, "R") ~
"Python",

        str_detect(WorkToolsSelect, "R") & str_detect(WorkToolsSelect, "Python")   ~ "both",

        TRUE ~ "neither"

    ))

# Print the first 6 rows

head(debate_tools)


# Group by language preference, calculate number of responses, and remove "neither"

debate_plot <- debate_tools %>%

    group_by(language_preference)  %>%

    summarise(count = n())  %>%

    filter(!language_preference == "neither")


# Creating a bar chart

ggplot(debate_plot, aes(x = language_preference, y = count)) +

  geom_bar(stat = "identity")


# Group by, summarise, arrange, mutate, and filter

recommendations <- debate_tools  %>%

    group_by(language_preference, LanguageRecommendationSelect)  %>%

    summarise(count = n())  %>%
```

```
    arrange(language_preference, desc(count))  %>%
    mutate(row = row_number()) %>%
    filter(row <= 4)

# Create a faceted bar plot
ggplot(recommendations, aes(x = LanguageRecommendationSelect, y = count)) +
    geom_bar(stat = "identity") +
    facet_wrap(~language_preference)

# Would R users find this statement TRUE or FALSE?
R_is_number_one = TRUE
```

Wrangling and Visualizing Musical Data

Task 1: Instructions

Read in the McGill Billboard chord dataset.

- Load in the `dplyr`, `readr`, and `ggplot2` packages.
- Read in `'datasets/bb_chords.csv'` using `read_csv` and assign it to `bb`.
- Display the first rows of `bb`.

Make sure to use `read_csv` (with an *underscore*) to read in the data. The `read.csv` function, which is built into R, has a number of problems which the `read_csv` function avoids.

Good to know

This project assumes familiarity with standard tidyverse tools for R like the `dplyr`, `ggplot2` and the pipe operator (`%>%`). Before taking on this project we recommend that you have completed the following courses:

- Introduction to the Tidyverse
- Intermediate Data Visualization with ggplot2

RStudio has created some very helpful cheat sheets for working in the tidyverse, including two that will be helpful for this project: Data Wrangling and Data Visualization with ggplot2. If you're a serious data wrangler, you might even print them out and laminate them!

Task 2: Instructions

Find the most common chords in the McGill Billboard Dataset.

- Count the number of occurrences of each raw `chord` type in the dataset (`bb`) using `count()`, and sort the results from most common (highest count) to least common (lowest count).
- Store the result in `bb_count`.
- Display the 20 most common chords.

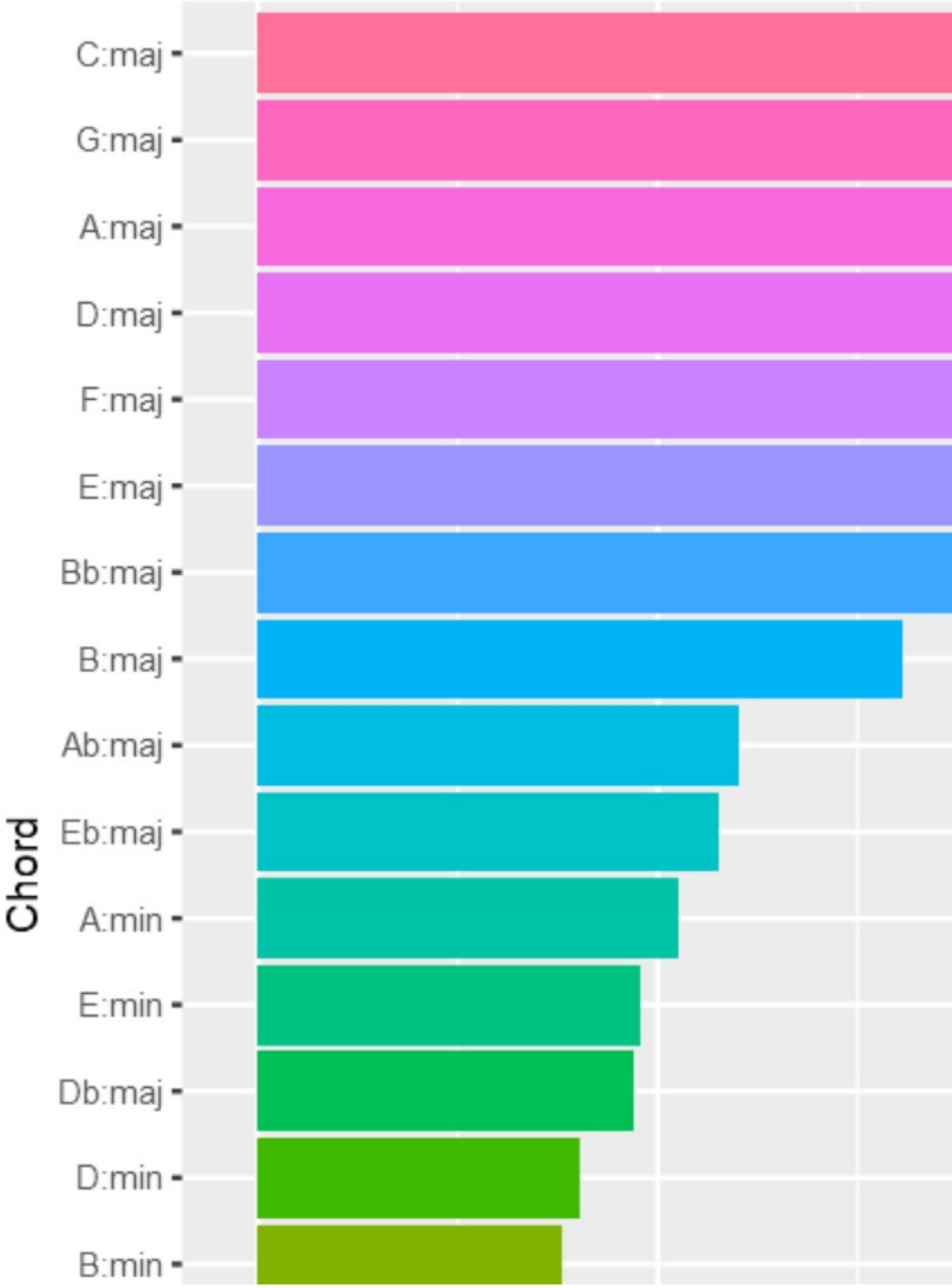
For readability (and to do things the tidyverse way!), try to write your code as a string of verb-based commands, one command per line, connected by `%>%`.

Task 3: Instructions

Plot the top 20 chords as a flipped bar plot.

- Starting with the first 20 records from `bb_count`, use `mutate` to create a new column `share` with the percentage of how often each chord type occurs.
- Also using `mutate`, `reorder` the `chord` column according to the value in `share`.
- Pipe the results into `ggplot()` and make a column plot where the X axis represents `chord` and the Y axis is represents `share`.
- Make your plot more readable by adding labels with `xlab()` and `ylab()`, and by flipping the plot using `coord_flip()`.

Do your best to make your visualization look like this:


Chord
C:maj
G:maj
A:maj
D:maj
F:maj
E:maj
Bb:maj
B:maj
Ab:maj
Eb:maj
A:min
E:min
Db:maj
D:min
B:min

A picture is worth a thousand words -- perhaps, even more, when visualizing data! That's why we're working so hard to make the visualizations as readable as possible -- using percentages, arranging values in descending order, etc.

You may also try adding a splash of color. (Remember that column plots require color to be added with `fill = chord` rather than `color`.) When color adds to the aesthetic, but not a new dimension of information, I recommend removing the color legend with `theme(legend.position='none')`.

As you're working through the above steps, think about what the plot would look like without some of these options. For example, what advantage does converting raw chords counts to percentages have for those reading the plot? How readable would the plot be without the axis labels? Without reordering columns? What value does `coord_flip()` add to this plot?

Task 4: Instructions

Create a count of chord *bigrams*.

- Use `mutate()` to add two new columns to `bb`: `next_chord` and `next_title`. These should contain the data from the `chord` and `title` columns, but shifted one row up. Use the `lead()` function inside your `mutate()` command to do this.
- Create a `bigram` column that concatenates `chord` with `next_chord`, with a space in between.
- Use `filter()` to remove any records in our new data frame where `title` and `next_title` are not identical.
- Count the number of occurrences of each bigram type and store the results in `bb_bigram_count`.
- Display the 20 most common chord bigrams.

There are natural language processing (NLP) tools that will *tokenize* texts by *n-grams* (phrases of *n* words). However, our chord data is already in a tidy table, rather than in something that looks like paragraph form. Thankfully, `dplyr` contains functions like `lag()` and `lead()` that make it easy to access data from other rows in the data frame efficiently, and we can use them to construct our bigrams using `paste()` (or `str_c` from `stringr`).

Why we `filter` in step 3 might not be obvious, but it's incredibly important. The last chord of one song combined with the first chord of the next song is *not* a bigram. Depending on the order of songs in the dataset, if we skip this step, we could end up with chord "progressions" connecting songs that occur perhaps 30 years apart in history!

Task 5: Instructions

Create a flipped bar plot that shows the 20 most common chord bigrams.

- Copy your code from Step 3, and modify it to work with `bb_bigram_count` instead of `bb_count`.
- Adjust the plot labels to fit chord *changes* instead of just chords.

Copy-and-paste isn't cheating! In fact, knowing how to successfully copy, paste, and tweak existing code (yours, or someone else's -- with permission, of course) is an integral part of data science. It not only saves time and brain power, it also limits mistakes in your code when you use code you already know works. The iterative process of tweaking that code can also help you write more efficient code in the future.

Of course, if you copy-and-paste the same code several times, you may just want to write a custom function instead!

Task 6: Instructions

Find and display the 30 artists with the most songs in the McGill Billboard Dataset.

- Using bb, isolate the artist and title columns using select().
- We still have one record per *chord*. Use `unique()` to remove duplicates and leave a single record per *song*.
- As in earlier tasks, use `count()` to find how many songs each artist has in the dataset, and sort the results in descending order.
- Display the first 30 records in the sorted table.

In order to tag as many songs as possible quickly in the next task, we can simply identify a small number of prolific artists whose songs we can tag all at once. By isolating the 30 most prolific artists in the dataset, we can look at the results and pick a few good candidates.

When used in a piped string of commands, `unique()` does not need to take any arguments, since each command treats the output of the previous command as its first argument.

Task 7: Instructions

Add a new column `instrument` to `bb`, including "piano" or "guitar" for piano- and guitar-driven songs.

- Use `inner_join()` with `tags` to attach an `instrument` column to `bb` and assign the result to `bb_tagged`.
- Display the new data frame `bb_tagged` to make sure the join was successful.

When adding a custom column to an entire data frame based on data in another column, it is usually much faster to use the appropriate `join` operation than to write a looping function. `inner_join()` will even remove all rows in `bb` that do not correspond to the artists in `tags`. And in this case, since both `bb` and `tags` have an `artist` column, you do not need to specify a column by which to join.

Try it out with `left_join()` and `full_join()`, too. What are the differences? Do any produce the same results in this case? What would happen if you started with `tags` and applied a join operation to `bb`? Which join(s) would produce the desired results?

Task 8: Instructions

Created a faceted plot that shows the frequency of the most common chords side-by-side for songs by piano- and guitar-driven artists.

- Starting with `bb_tagged`, use `filter()` to keep only the `top_20` chords.
- Use `count()` to find the number of times each `chord` occurs for each `instrument`, and sort the results.
- Pipe the results to `ggplot()` and make a bar plot, using `chord` as the X axis and `n` (the result of `count()`) as your Y axis.
- Use `facet_grid()` to place guitar and piano plots side by side for comparison. Then use `coord_flip()` for readability, and provide appropriate labels for the X and Y axes.

If you like, add a splash of color with `fill` (and if so, set `theme(legend.position='none')`).

`facet_wrap()` and `facet_grid()` are incredibly powerful visualization tools. They allow you to add dimensions to your data visualization story without making things hard on your readers.

Try playing around with faceting a bit. What happens when you count `chord` and `artist` and pass `artist` to `facet_grid()`? What other parameters could you visualize in this way that tell a compelling story?

Task 9: Instructions

Create the same faceted plot as in Task 8, but for chord bigrams.

- Copy and modify your code from Task 4 to add a `bigram` column, this time to `bb_tagged`.
- Copy and modify your code from Task 8 to produce a faceted plot of bigram frequency from the `top_20_bigram`s that compares guitar- and piano-driven songs.
- Remember to change all references to chords (including in the axis labels) to bigrams.

Task 10: Instructions

Complete the project by confirming the validity of the hypothesis, as well as the need for further data analysis to draw a conclusion.

- Is the hypothesis that guitar-driven and piano-driven songs have different chord tendencies valid and worth deeper exploration? `TRUE` or `FALSE`? Set `hypothesis_valid` to reflect your answer.
- To draw a conclusion about this hypothesis, do we still need to explore more data? `TRUE` or `FALSE`? Set `more_data_needed` to reflect your answer.

Great work! You've uncovered some interesting things about musical chord progressions *and* learned a little about how natural language processing (NLP) analysis techniques can be used to study musical symbolic data.

Do you want to know more?

If this project got you hungry for more musical data analysis, check out my blog post, What is computational musicology? There are links to academic studies, tools, and datasets for further exploration. The Million Song Dataset is especially cool.

And if you're also a Python fan, check out music21, an advanced toolkit for computational musicology in Python.

Exploring the Bitcoin Cryptocurrency Market

Task 1: Instructions

Load the saved CSV file and select relevant columns.

- Load `datasets/coinmarketcap_06122017.csv` into a DataFrame named `dec6` using `read_csv()` from `pandas`.
- Select the columns `id` and `market_cap_usd` and assign them to `market_cap_raw`.
- Use `count()` to count and print the number of values in `market_cap_raw`.

Good to know

To complete this project, you need to be fluent with `pandas` DataFrames. Before starting this project, we recommend that you have completed the following courses:

- Data Manipulation with pandas
- Data Cleaning in Python

This project uses `pandas.DataFrame.plot()` and the `Axes` API in matplotlib extensively, so these are good references to have open in a separate tab.

Task 2: Instructions

Filter out the coins with no known market capitalization.

- `query()` the DataFrame and filter out all the valueless coins and assign the new DataFrame to `cap`.
- Use `count()` again to count and print the number of values in `cap`.

Using the `query()` method of a DataFrame is a convenient alternative to using slicing selectors. For example, this:

```
df.query('value > 0')
```

Gives you the same result as this:

```
df[ df['value'] > 0 ]
```

but with less code.

Keep in mind that `query()` uses numexpr syntax by default instead of python syntax. It means that this:

```
(condition1 and condition2) or condition3
```

Should be written like this using numexpr:

```
(condition1 & condition2) | condition3
```

Task 3: Instructions

Visualize the market capitalization of the top 10 cryptocurrencies.

- Select the first 10 coins, set the index to `id`, and assign the resulting DataFrame to `cap10`.
- Calculate the percentage of market capitalization for each coin using `assign()` and assign it to `cap10` again.
- Plot the top 10 coin's `market_cap_perc` in a barplot with the title "Top 10 market capitalization" and assign it to `ax`.
- Using the `ax` object, annotate the y axis with "% of total cap".

Check the pandas docs for using assign with lambda for calculating the % market cap. Remember that `.assign` iterates over all rows and creates a new column, but you can plug in numbers external to the DataFrame, for example:

```
cap.market_cap_usd.sum()
```

;-) . Also, don't forget to multiply by 100 inside the lambda to turn the resulting proportion into a percentage.

Pandas has an interface for every major plot type, for example `DataFrame.plot.hist()` and `DataFrame.plot.bar()`. For annotating the y axis using the `ax` object you could take a look at the available methods in the matplotlib docs for the Axes object.

Task 4: Instructions

Make the plot from the last task more informative with colors and a nice log scale.

- Make a plot like in the last task, but of `market_cap_usd`. Add the given `COLORS` and make the y-axis $\log^{10}$ scaled.
- Again, use the `ax` object to annotate the y axis with "USD".
- Remove the useless label on the x axis.

Scale the y axis using an arguent to `.plot.bar()`, so it is only visual. Do not modify the actual value of the column!

Task 5: Instructions

Create a DataFrame that contains volatility information on cryptocurrencies.

- Select the columns `id`, `percent_change_24h`, and `percent_change_7d` from `dec6` and assign the resulting DataFrame to `volatility`.
- Set the index to `id` and drop all rows that contain NaNs.
- Sort `volatility` by `percent_change_24h` in ascending order.
- Print out the `.head()` of `volatility`.

Task 6: Instructions

Make a bar plot that shows the biggest gainers and the biggest losers. Finish writing the function that will show the top losers to the left and the top gainers to the right.

- Use `.plot.bar()` to plot the "top losers" from `volatility_series` in `'darkred'` color.
- Set the figure main title using the `fig.suptitle()` method.
- Set the ylabel for the plot on the left using its `Axes` object
- Use `.plot.bar()` again to plot the "top winners" bar chart in `'darkblue'`
- Call the function`top10_subplot` with `volatility.percent_change_24h` and the supplied title.

The function assumes that `volatility_series` is sorted and so `volatility_series[:10]` would pick out the top 10 losers and `volatility_series[-10:]` would pick out the top 10 winners.

In this task, the subplot is already defined for you. To assign a pandas plot to a matplotlib subplot, you need to do the following

```
fig, axes = plt.subplots(...)
#assigns the resulting pandas plot to the first subplot
df1.plot.bar(ax=axes[0])
#assigns the resulting pandas plot to the second subplot
df2.plot.bar(ax=axes[1])
```

Task 7: Instructions

Call the function you created in the last task above, but with the weekly data.

- Sort `volatility` by `percent_change_7d` in ascending order and assign it to `volatility7d`.
- Call `top10_subplot` with volatility7d and the supplied title.

Keep in mind that our data is not sorted now and that `top10_subplot` assumes the Series is in ascending order.

Task 8: Instructions

- Use the `.query()` method to select all large cap coins in `cap`. That is, coins where `market_cap_usd` is +10 billion USD.

- Assign the resulting DataFrame to `largecaps`.
- Print out `largecaps`.

Task 9: Instructions

Group *large*, *mid* and *small* cap coins into a group called *biggish* and make a barplot of counts of *biggish*, *micro* and *nano* coins.

- Count how many biggish, micro and nano coins there are using the given function `capcount`.
- Make a list with these 3 numbers and assign it to `values`.
- Make a barplot with `values` and the provided labels.

These are the market cap definitions from Investopedia:

- Large cap: +10 billion
- Mid cap: 2 billion - 10 billion
- Small cap: 300 million - 2 billion
- Micro cap: 50 million - 300 million
- Nano cap: Below 50 million

As `capcount` uses the `.query()` method the argument to `capcount` should be a string defining a condition for what values to select.

For this final task we will use the matplotlib `bar` interface, instead of `pandas`, as it is more convenient. Check the matplotlib.pyplot.bar docs for a reference.

```
# Importing pandas

import pandas as pd
```

```
# Importing matplotlib and setting aesthetics for plotting later.

import matplotlib.pyplot as plt

%matplotlib inline

%config InlineBackend.figure_format = 'svg'

plt.style.use('fivethirtyeight')
```

```
# Reading datasets/coinmarketcap_06122017.csv into pandas

dec6 = pd.read_csv('datasets/coinmarketcap_06122017.csv')


# Selecting the 'id' and the 'market_cap_usd' columns

market_cap_raw = dec6[['id', 'market_cap_usd']]


# Counting the number of values

market_cap_raw.count()


# Filtering out rows without a market capitalization

cap = market_cap_raw.query('market_cap_usd > 0')


# Counting the number of values again

cap.count()


#Declaring these now for later use in the plots

TOP_CAP_TITLE = 'Top 10 market capitalization'

TOP_CAP_YLABEL = '% of total cap'


# Selecting the first 10 rows and setting the index

cap10 = cap[:10].set_index('id')


# Calculating market_cap_perc

cap10 = cap10.assign(market_cap_perc = lambda x: (x.market_cap_usd /
cap.market_cap_usd.sum())*100)
```

```
# Plotting the barplot with the title defined above
ax = cap10.market_cap_perc.plot.bar(title=TOP_CAP_TITLE)

# Annotating the y axis with the label defined above
ax.set_ylabel(TOP_CAP_YLABEL);

# Colors for the bar plot
COLORS = ['orange', 'green', 'orange', 'cyan', 'cyan', 'blue', 'silver', 'orange', 'red', 'green']

# Plotting market_cap_usd as before but adding the colors and scaling the y-axis
ax = cap10.market_cap_usd.plot.bar(title=TOP_CAP_TITLE, logy=True, color = COLORS)

# Annotating the y axis with log(USD)
ax.set_ylabel('USD')

# Final touch! Removing the xlabel as it is not very informative
ax.set_xlabel('');

# Selecting the id, percent_change_24h and percent_change_7d columns
volatility = dec6[['id', 'percent_change_24h', 'percent_change_7d']]

# Setting the index to 'id' and dropping all NaN rows
volatility = volatility.set_index('id').dropna()

# Sorting the DataFrame by percent_change_24h in ascending order
```

```
volatility = volatility.sort_values('percent_change_24h')

# Checking the first few rows
volatility.head()

# Defining a function with 2 parameters, the series to plot and the title
def top10_subplot(volatility_series, title):
    # making the subplot and the figure for nrows and ncolumns
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 6))
    # Plotting with pandas the barchart for the top 10 losers with the color RED
    ax = volatility_series[:10].plot.bar(color="darkred", ax=axes[0])
    # Setting the main title to TITLE
    fig.suptitle(title)
    # Setting the ylabel to "% change"
    ax.set_ylabel('% change')
    # Same as above, but for the top 10 winners and in darkblue
    ax = volatility_series[-10:].plot.bar(color="darkblue", ax=axes[1])
    # Returning this for good practice, might use later
    return fig, ax

DTITLE = "24 hours top losers and winners"

# Calling the function above with the volatility.percent_change_24h series
# and title DTITLE
fig, ax = top10_subplot(volatility.percent_change_24h, DTITLE)
```

```
# Sorting percent_change_7d in ascending order
volatility7d = volatility.sort_values("percent_change_7d")

WTITLE = "Weekly top losers and winners"

# Calling the top10_subplot function
fig, ax = top10_subplot(volatility7d.percent_change_7d, WTITLE);

# Selecting everything bigger than 10 billion
largecaps = cap.query("market_cap_usd > 1E+10")

# Printing out largecaps
largecaps

# Making a nice function for counting different marketcaps from the
# "cap" DataFrame. Returns an int.
# INSTRUCTORS NOTE: Since you made it to the end, consider it a gift :D
def capcount(query_string):
    return cap.query(query_string).count().id

# Labels for the plot
LABELS = ["biggish", "micro", "nano"]

# Using capcount count the not_so_small cryptos
```

```
biggish = capcount("market_cap_usd > 3E+8")


# Same as above for micro ...

micro = capcount("market_cap_usd >= 5E+7 & market_cap_usd < 3E+8")


# ... and for nano

nano =  capcount("market_cap_usd < 5E+7")


# Making a list with the 3 counts

values = [biggish, micro, nano]


# Plotting them with matplotlib

plt.bar(range(len(values)), values, tick_label=LABELS);
```

Name Game: Gender Prediction using Sound

Task 1: Instructions

Explore the NYSIIS algorithm. There are no right or wrong answers in this task, so go nuts! The purpose is just for you to explore the `fuzzy.nysiis` function.

- Import the `fuzzy` module
- Explore the output of the `fuzzy.nysiis` function using any words.
- Use `fuzzy.nysiis` to test the equality of two words that you think sound the same.

There are a number of fuzzy name-matching algorithms. In this project, we will use the NYSIIS algorithm which is part of the `fuzzy` module. The function `fuzzy.nysiis` takes a string and outputs a phonetic (that is, sound) version of that string. For example, both `fuzzy.nysiis('colour')` and `fuzzy.nysiis('color')` outputs `'CALAR'`, which is how the word sounds. The algorithm is quite useful for catching (and correcting!) certain typos; specifically, misspelled words that sound right phonetically. For example, tomorrow is commonly misspelled as tommorow. Does `fuzzy.nysiis` equate them?

Good to know

To complete this project, you should be familiar with `pandas` DataFrames, Numpy for basic statistics, and Matplotlib for plotting. We recommend that you have completed the following courses:

- Intermediate Python for Data Science
- Data Manipulation with pandas

This project uses the `fuzzy` package and you might want to have a look at its documentation.

Task 2: Instructions

Read in the book data and extract the authors' first names.

- Import the `pandas` module.
- Read `datasets/nytkids_yearly.csv` into `author_df`. Note that `nytkids_yearly.csv` uses semicolon (`;`) as `delimiter`.
- Loop through `author_df['Author']` to extract authors' first names and append them to `first_name`.
- Add `first_name` as a column in `author_df`.
- Check out the first rows of `author_df` using `author_df.head()`

Author's full names (`author_df['Author']`) are simply short strings. The words that make up a string can be separated using the `split` method. In our case, names are separated by just a whitespace. See here for some examples of how to use `split`.

Task 3: Instructions

Create an NYSIIS equivalent of authors' first names.

- Import the `numpy` module.
- For each `first_name`, create an nysiis equivalent and append to `nysiis_name`.
- Add `nysiis_name` as a column to `author_df`.
- As a sanity check: Print out the difference between the unique number of first names and unique number of NYSIIS names. Is it greater than 0?

If you've imported `numpy` as `np` you can apply the `np.unique` function to get a list of unique values.

Task 4: Instructions

Read in the baby name dataset and add to it a new column that indicates gender.

- Read in `datasets/babynames_nysiis.csv` as `babies_df`. Note that it is semicolon (`;`) delimited.
- Loop through the indexes (rows) of `babies_df`. Append 'M' (Male), 'F' (Female) or 'N' (Neutral) to `gender` depending on the values of `perc_female` and `perc_male`.
- Add `gender` as a column to `babies_df`.
- To make sure you're on the right track, print out the first few rows of `babies_df`.

Task 5: Instructions

Figure out the genders of the authors.

- Loop through `author_df['nysiis_name']` to find the index of each author's name in the in `babies_df`.
- Use this index to extract `gender` from `babies_df` and append it to `author_gender`. For cases where a name does not exist in `babies_df` append 'Unknown' instead.
- Add `author_gender` to `author_df` and print out the first few rows of `author_df`.
- Use the method `value_counts()` to tally up the different values in `author_df['author_gender']`.

I've provided you with the function `locate_in_list(a_list, element)` that retrieves the index of an `element` in `a_list`. You can use this to match names from the author list to the names in the baby list. If an element does not exist in `a_list`, the function will return a value of -1. `locate_in_list` takes in a list of names, so you'll need to convert your DataFrame column to a list first. You can do that like this:

```
list(babies_df['babynysiis'])
```

Task 6: Instructions

Tally up the gender of the authors over time.

- Create a list `years` containing unique year values (from `author_df`) in ascending order.
- Loop through year values and count the occurrences of each of (`M`, `F`, `Unknown`) per year and append to the lists `males_by_yr`, `females_by_yr`, and `unknowns_by_yr`.
- Print out yearly values to examine changes over time

In a column of `author_df` you can find the number of occurrences like this:

```
len( author_df[ author_df['Gender']=='F' ] )
```

And you can add more conditions using the `&` operator.

Task 7: Instructions

Visualize the foreign-born authors using a bar chart.

- Use `plt.bar` to make a barplot of `unknown_by_yr` by `year`.
- [*Optional*] Add a title and axes labels to your chart.

Using `matplotlib` there are many ways to customize a bar chart. Check out [the official documentation for `plt.bar`] for the full list of options.

Task 8: Instructions

Compare male and female authorship using a grouped bar chart.

- Create a new list `years_shifted`, where 0.25 is added to each element in `years`.
- Make a bar plot for `males_by_yr` by `year`, with `width=0.25` and `color='lightblue'`.
- Make a bar plot for `females_by_yr` by `year_shifted`, with `width=0.25` and `color='pink'`.
- [*Optional*] Add axes labels and a title.

One way to make grouped bar charts is to plot two bar charts on top of each other, but where the x-positions are shifted for one of the charts, and the bars are made more narrow to not overlap. This is the approach you'll take in this task.

If you want to know more

There are a number of different phonetic algorithms; each with its own set of advantages and disadvantages. Check out this blog post if you want to know more.

```
# Importing the fuzzy package

import fuzzy


# Exploring the output of fuzzy.nysiis

fuzzy.nysiis('tufool')


# Testing equivalence of similar sounding words

fuzzy.nysiis('tomorrow') == fuzzy.nysiis('tommorow')


# Importing the pandas module

import pandas as pd


# Reading in datasets/nytkids_yearly.csv, which is semicolon delimited.

author_df = pd.read_csv('datasets/nytkids_yearly.csv', delimiter=';')


# Looping through author_df['Author'] to extract the authors first names

first_name = []

for name in author_df['Author']:

    first_name.append(name.split()[0])
```

```
# Adding first_name as a column to author_df
author_df['first_name'] = first_name

# Checking out the first few rows of author_df
author_df.head()

# Importing numpy
import numpy as np

# Looping through author's first names to create the nysiis (fuzzy) equivalent
nysiis_name = []
for firstname in author_df['first_name']:
    nysiis_name.append(fuzzy.nysiis(firstname))

# Adding nysiis_name as a column to author_df
author_df['nysiis_name'] = nysiis_name

# Printing out the difference between unique firstnames and unique nysiis_names:
diff_names = len(np.unique(author_df.first_name)) - \
    len(np.unique(author_df.nysiis_name))
print('There are ' + str(diff_names) +
      ' more unqiue values for first_name than nysiis_name')

# Reading in datasets/babynames_nysiis.csv, which is semicolon delimited.
```

```
babies_df = pd.read_csv('datasets/babynames_nysiis.csv', delimiter=';')


# Looping through the rows of babies_df to and filling up gender
gender = []
for idx in range(len(babies_df['babynysiis'])):
    if babies_df.perc_female[idx] > babies_df.perc_male[idx]:
        gender.append('F')
    elif babies_df.perc_female[idx] < babies_df.perc_male[idx]:
        gender.append('M')
    else:
        gender.append('N')


# Adding a gender column to babies_df
babies_df['gender'] = gender


# Printing out the first few rows of babies_df
babies_df.head()


# This function returns the location of an element in a_list.
# Where an item does not exist, it returns -1.
def locate_in_list(a_list, element):
    loc_of_name = a_list.index(element) if element in a_list else -1
    return(loc_of_name)


# Looping through author_df['nysiis_name'] and appending the gender of each
```

```
# author to author_gender.
author_gender = []
for name in author_df['nysiis_name']:
    nloc = locate_in_list(list(babies_df['babynysiis']), name)
    if nloc == -1:
        author_gender.append('Unknown')
    else:
        author_gender.append(babies_df['gender'][nloc])

# Adding author_gender to the author_df
author_df['author_gender'] = author_gender

# Counting the author's genders
author_df['author_gender'].value_counts()

# Creating a list of unique years, sorted in ascending order.
years = list(np.unique(author_df.Year))

# Intializing lists
males_by_yr = []
females_by_yr = []
unknown_by_yr = []

# Looping through years to find the number of male, female and unknown authors per year
for yr in years:
```

```
    males_by_yr.append(
        len(author_df[(author_df["author_gender"] == 'M') & (author_df["Year"] == yr)]))
    females_by_yr.append(
        len(author_df[(author_df["author_gender"] == 'F') & (author_df["Year"] == yr)]))
    unknown_by_yr.append(len(
        author_df[(author_df["author_gender"] == 'Unknown') & (author_df["Year"] == yr)]))

# Printing out yearly values to examine changes over time
data = np.array([males_by_yr, females_by_yr, unknown_by_yr])
headers = ['males', 'females', 'unknowns']
pd.DataFrame(data, headers, years)

# Importing matplotlib
import matplotlib.pyplot as plt

# This makes plots appear in the notebook
%matplotlib inline

# Plotting the bar chart
plt.bar(years, unknown_by_yr)

# [OPTIONAL] - Setting a title, and axes labels
plt.title('unknown gender by year')
plt.xlabel('years')
```

```
# Creating a new list, where 0.25 is added to each year
years_shifted = [year + 0.25 for year in years]

# Plotting males_by_yr
plt.bar(years,  males_by_yr,  width=0.25, color='lightblue')

# Plotting females_by_yr by years_shifted
plt.bar(years_shifted,  females_by_yr,  width=0.25, color='pink')

# [OPTIONAL] - Adding relevant Axes labels and Chart Title
plt.xlabel('Years')
plt.ylabel('No. Authors')
plt.title('Authors by Gender')
```

Exploring the Evolution of Linux

Task 1: Instructions

Print out the raw data in the Git log file to show the available format.

- Print out the content of the sample file `datasets/git_log_excerpt.csv`.

Good to know

This Project requires that you know your way around Python and Pandas. We recommend that you have complete these DataCamp courses before doing this project:

- Intermediate Python for Data Science
- Data Manipulation with pandas
- Manipulating Time Series Data in Python

Task 2: Instructions

Read in the Linux Git log file with Pandas.

- Load in the `pandas` module as `pd`.
- Read in the log file `datasets/git_log.gz`. Name the 1st column `"timestamp"` and the 2nd column `"author"`.
- Assign the resulting DataFrame to `git_log`.
- Print out the first five rows of `git_log`.

The `pandas` method `read_csv` can read a CSV file compressed in a `gz` file. You will have to specify the `sep`, `encoding`, `header`, and `names` arguments to `read_csv`.

Task 3: Instructions

Gather some basic metrics about Linux's Git repository.

- Count the number of commits in `git_log`.
- Count the number of all contributing authors. Leave out the entries that don't have an author at all.

Here, use some basic functions of Python, Pandas' `DataFrame` and `Series` to count values and to remove missing data.

Task 4: Instructions

List the ten authors that made the most commits.

- Count how often each author occurs in `git_log`, pick out the top ten authors, and assign the result to `top_10_authors`.

In this task, the result that is stored in `top_10_authors` has to be a `Series` or a `DataFrame` that includes the authors and the number of commits that each author has made.

Task 5: Instructions

Transform the numbers in `timestamp` to time series-based data type.

- Convert the `timestamp` column to a Pandas' `Timestamp` type
- Look at a summary of the converted `timestamp` column to check if the conversion was successful and if the boundary values make sense.

Here is [the official Pandas documentation for how to convert these type of time stamps]() (called *epoch* time stamps) to `Timestamp`. Be sure to set the right `unit` of time (in our case: seconds) to the date conversion method. To summarize the resulting `Timestamp` column you could use the `describe()` method.

Task 6: Instructions

Determine a right time period and keep only those commits within this time period.

- Pick a reasonable *first* timestamp and assign it to `first_commit_timestamp`.
- Pick a reasonable *last* timestamp for this dataset from late 2017 and assign it to `last_commit_timestamp`.
- Create a new `DataFrame` called `corrected_log`.
- Use `describe()` on `corrected_log['timestamp']` to check the data.

A possible valid time period:

- The *first* reasonable entry is the first commit from Linus Torvalds.
- Every commit before the year 2018 would be a reasonable *last* timestamp.

Task 7: Instructions

Count the number of commits of the `corrected_log` for each year:

- Create a new `DataFrame` called `commits_per_year` that sums up all commits annually, starting at January 1st.
- Show the first five rows of the `DataFrame`.

There are many ways to accomplish this with Pandas. Use the `groupby` method with the utility function `Grouper` to group by year:

```
my_data_frame.groupby(
    pd.Grouper(
        key='my_timestamp_column',
        freq='AS'
        )
    )
```

Here, `freq='AS'` makes `groupby` group by year using the 1st of January as starting day.

Task 8: Instructions

Visualize the yearly counts using a suitable plot.

- Plot `commits_per_year` using the `pandas` `plot` method.
- Add a suitable `title`.
- Turn the `legend` off.

The `plot` method in `pandas` takes many options that allow you to customize your plot. Here is the official documentation for `plot` for `Series`. That documentation contains a lot of info, but the arguments you might want to add here are `kind`, `title`, and `legend`.

Task 9: Instructions

Thanks for doing the project! As a last task:

- Set `year_with_most_commits` to the year with the most commits to Linux (as of autumn 2017).

Further Reading

If you are more interested in mining software repositories, take a look at the following books:

- Adam Tornhill: Software X-Ray. Pragmatic Programmers, 2018.
- Christian Bird, Tim Menzies, Thomas Zimmermann: The Art and Science of Analyzing Software Data. Morgan Kaufmann, 2015.
- Tim Menzies, Laurie Williams, Thomas Zimmermann: Perspectives on Data Science for Software Engineering. Morgan Kaufmann, 2016.

```
# Printing the content of git_log_excerpt.csv
with open("./datasets/git_log_excerpt.csv") as f:
    print(f.read())

# Loading in the pandas module as 'pd'
import pandas as pd

# Reading in the log file
git_log = pd.read_csv(
    'datasets/git_log.gz',
    sep='#',
    encoding='latin-1',
    header=None,
    names=['timestamp', 'author']
)

# Printing the first 5 rows
git_log.head()

# calculating number of commits
number_of_commits = len(git_log)

# calculating number of authors
number_of_authors = len(git_log['author'].dropna().unique())
```

```
# printing out the results

print("%s authors committed %s code changes." % (number_of_authors,
number_of_commits))



# Identifying the top 10 authors

top_10_authors = git_log['author'].value_counts().head(10)



# Listing contents of 'top_10_authors'

top_10_authors.head(10)



# converting the timestamp column

git_log['timestamp'] = pd.to_datetime(git_log['timestamp'], unit="s")



# summarizing the converted timestamp column

git_log['timestamp'].describe()



# determining the first real commit timestamp

first_commit_timestamp = git_log.iloc[-1]['timestamp']



# determining the last sensible commit timestamp

last_commit_timestamp = pd.to_datetime('2018')



# filtering out wrong timestamps

corrected_log = git_log[

    (git_log['timestamp'] >= first_commit_timestamp) &

    (git_log['timestamp'] <= last_commit_timestamp)]
```

```
# summarizing the corrected timestamp column
corrected_log['timestamp'].describe()

# Counting the no. commits per year
commits_per_year = corrected_log.groupby(
    pd.Grouper(key='timestamp', freq='AS')).count()

# Listing the first rows
commits_per_year.head()

# Setting up plotting in Jupyter notebooks
%matplotlib inline

# plot the data
commits_per_year.plot(kind='bar', title="Commits per year (Linux kernel)", legend=False)

# calculating or setting the year with the most commits to Linux
year_with_most_commits = 2016
```

Recreating John Snow's Ghost Map

Task 1: Instructions

Import the data that John Snow collected about the cholera epidemic.

- Read about Dr. John Snow to the right.
- Load in the `pandas` module.
- Import the data `datasets/deaths.csv` and assign the resulting DataFrame to `deaths`.
- Print out the first rows of `deaths`.

Good to know

This Project is designed to test also your knowledge of pandas and Bokeh. If you'd like to refresh your memory, the recommended prerequisites for this course are Data Manipulation with pandas and Interactive Data Visualization with Bokeh.

Even if you've finished all the DataCamp Python courses you may still find this project challenging unless you use/read some external *documentation*.
In this case check out Karlijn's Datacamp pandas DataFrame tutorial, Hugo's Hierarchical indices, groupby and pandas tutorial, and pandas' cheat sheet that summarizes the basics of pandas DataFrames. You could also look at the official pandas documentation.

Stack Overflow is also a useful resource. A handy search pattern is **example of ??? in pandas** where **???** is what you need to do.

A big thank you to Robin Wilson from Southampton University who digitized John Snow's original data and georeferenced it to the Ordnance Survey co-ordinate system which will allow us to analyze it and overlay it on modern maps of that area.

Task 2: Instructions

Check, rename columns, and describe the DataFrame.

- Summarize the content of `deaths` (from previous exercise) with `.info()` method.
- Prepare dictionary that will be used to rename the `Death`, `X coordinate`, and `Y coordinate` columns to `death_count`, `x_latitude`, and `y_longitude`, respectively.
- Rename the columns of the dataset with the `.rename()` method.
- Describe the dataset with the `.describe()` method.

The following exercise may be helpful:

- Inspecting DataFrames from `pandas` Foundations

Task 3: Instructions

Prepare and pre-process the data for plotting.

- Create a subset (called `locations`) of the original dataset selecting only `x_latitude` and `y_longitude` columns.
- Transform this subset into list of `x_latitude` and `y_longitude` pairs and name it `deaths_list`.
- Check the length of this list (the number of pairs).

The following links may be helpful:

- Selecting columns using `[]` from Intermediate Python for Data Science
- Subselecting DataFrames with lists from Manipulating DataFrames with `pandas`
- Pandas DataFrame to list on Stack Overflow

Task 4: Instructions

Loop through the pre-processed data to create a map.

- Fill in the `len` function to loop through the data.

More info about on loops can be found here: Loops/LearnPython.org

Basic info about the folium library is found here: Folium 0.5.0 documentation

The map displayed in the notebook is also in the 2nd reprint of *On the Mode of Communication of Cholera (1855)* that is publicly available here.

Task 5: Instructions

Recreate The Ghost Map.

- Import the data `datasets/pumps.csv` and assign the resulting DataFrame to `pumps`.
- Create subset `locations_pumps` of the original dataset (select only `'X coordinate'` and `'Y coordinate'` columns).
- Transform this subset into list of `'X coordinate'` and `'Y coordinate'` pairs and call it `pumps_list`.
- Create `for loop` to plot all the points on a map (we will use `folium/Leaflet library` again).

The following exercises may be helpful:

- Selecting columns using `[]` from Intermediate Python for Data Science
- Dates in DataFrames from `pandas` Foundations
- Subselecting DataFrames with lists from Manipulating DataFrames with `pandas`

Task 6: Instructions

Reanalyze the John Snow's data about the Cholera Outbreak.

- Import the data `datasets/dates.csv` as DataFrame `dates` and parse `date` column as the datetime data type.
- Create new column `day_name` that will contain name of the day (Monday to Sunday) using `dt.weekday_name` attribute.
- Create new column `handle` that will contain a Boolean (True or False) for whether or not the handle was present.

The following links may be helpful:

- pandas.to_datetime
- Selecting columns using `[]` from Intermediate Python for Data Science
- Dates in DataFrames from `pandas` Foundations

Task 7: Instructions

Visualize the data about the Cholera Outbreak using the Bokeh library.

- Plot a line graph for cholera deaths vs. date.
- Plot a circle/point graph for cholera deaths vs. date.
- Plot a line graph for cholera attacks vs. date.

The following exercises may be helpful:

- From Interactive Data Visualization with Bokeh:
  - Plotting data from Pandas DataFrames
  - Lines
  - Customizing glyphs
  - Selection and non-selection glyphs
  - How to create legends
- Dates in DataFrames from `pandas` Foundations

Task 8: Instructions

True or false?

- Given the data John Snow collected and The Ghost Map he created, is it `True` or `False` that he "knows nothing"?

Congratulations, for completing the project! You should now check your project by clicking the "Check project" button.

Good luck on your data science journey! :)

Level Difficulty in Candy Crush Saga

Task 1: Instructions

Load in the packages you're going to need for the project:

- `readr`
- `dplyr`
- `ggplot2`

Good to know

This project assumes you have used the `dplyr` and `ggplot2` packages and that you are familiar with the pipe operator (`%>%`). Before taking on this project we recommend that you have completed the following courses:

- Introduction to the Tidyverse
- Intermediate Data Visualization with ggplot2

RStudio has created some very helpful cheat sheets, including two that will be helpful for this project: Data Wrangling and Data Visualization with ggplot2. We recommend that you keep them open in a separate tab to make it easy to refer to them.

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

Task 2: Instructions

Load in the dataset and display the first couple of rows.

- Load the csv file located at `datasets/candy_crush.csv` using `read_csv` and assign it to the variable `data`.
- Display the first six rows of the `data` with `head()`.

Make sure to use `read_csv` (with an *underscore*) to read in the data. The `read.csv` function, which is built into R, has a number of problems which the new `read_csv` function avoids.

We define the **granularity** of a dataset as the *lowest level of detail* of the observations. Here that means the combination of `level`, `player_id`, and `dt`. The rest of the columns are the *facts* that happened at that *level of detail*. That is, what happened for a given player, at a given day, at a given level. Sometimes we refer to the two types of columns as **id** columns (`level`, `player_id`, `dt`) and **variable** columns (`num_attempts`, `num_success`).

Task 3: Instructions

Count how many players are in the dataset and how many days it spans.

- Count the number of unique players included in the `data`.
- Compute the period for which we have data.

Remember that there might be several rows for each player, but here you should calculate the number of `unique` players.

One of the nice features of the function `read_csv` from the `readr` package is that it uses a heuristic to figure out the types of your columns. That's why the `dt` column has already been parsed as `Date`. This is useful because there are many functions that work with `Date`s, for example `range()`.

Task 4: Instructions

Calculate the probability of winning a level in a single attempt for each level.

- Group the dataset by `level`.
- Compute the total number of attempts and wins for each level by using the `summarise` function.
- Compute the probability to win as the number of wins divided by the number of attempts. Assign the result to a new column called `p_win` using `mutate`.
- The resulting data frame should be assigned to `difficulty` and printed out.

Modeling the probability of winning a level (p_{win}) as a Bernoulli process is, of course, a simplification. In reality, this probability will also depend on the skill of each player and the player could learn from past attempts and play better every time. But to include those assumptions is a refinement that we will leave for another occasion.

Task 5: Instructions

Plot a line graph with the difficulty for each level.

- Use ggplot to plot a line graph with `p_win` on the Y-axis and `level` on the X-axis.
- Set the breaks of the X-axis to show a tick mark for every level.
- Set the Y-axis labels to a nicely formatted percentage using the `scales` package.

For how to set the tick marks of a continuous scale see the ggplot2 documentation for `scale_x_continuous` (the examples could also be useful to look at).

If you don't know how to percent format the Y-axis check out this stack overflow question.

Task 6: Instructions

Add points to the plot and a horizontal dashed line at the 10% value.

- Copy and paste the solution from task 5.
- In addition to the lines *between* the datapoints, add a point *at* each datapoint.
- Add a horizontal *dashed* line to the plot at Y-axis value 10%.

Check out the documentation for geom_point for how to add points to a plot.

Check out the documentation for `geom_hline` for how to annotate a plot with horizontal lines.

Task 7: Instructions

Compute the standard error of the difficulty for each level using the given formula.

- Add the column `error` to `difficulty` which should contain the standard error of p_{win} using the formula `error = sqrt(p_win * (1 - p_win) / attempts)`.

There are many ways we could calculate the uncertainty around the difficulty estimates. We could, for example, have used bootstrap estimation or Bayesian modeling. However, calculating standard errors is a very quick way of getting uncertainty estimates that in many cases are good enough.

Task 8: Instructions

Add error bars to the difficulty profile plot.

- Copy and paste the `ggplot` code used to generate the plot in task 6.
- Use `geom_errorbar` to add error bars that range from `p_win - error` to `p_win + error` for each level.

`ggplot2` has many nice ways of showing vertical intervals: lines, crossbars and error bars. Here you will use error bars. Check out the documentation for `geom_errorbar` for how to do this.

Task 9: Instructions

Calculate how likely is it that a player will complete all the levels in the first attempt.

- Calculate the probability of the average player completing every level in the first attempt. Assign the result to `p`.

The probability of two independent events happening is simply the product of the individual probabilities. So the probability of winning both level 1 *and* level 2 on the first attempt would be

```
p_win[1] * p_win[2]
```

To extend this to all the 15 levels in the episode you can use the `prod` function which multiplies all the numbers in a vector together (that is, takes the *product* of all the vector elements).

Task 10: Instructions

Should our level designer worry that a lot of players will complete the episode in one attempt?

- Set `should_the_designer_worry` to `TRUE` or `FALSE` to indicate your answer.

If you want to know more

Now that you've analyzed some Candy Crush data, maybe you want to try out the actual game? Perhaps you can think of other level metrics you would want to calculate?

Bad passwords and the NIST guidelines

Task 1: Instructions

Load in and inspect the usernames and passwords of the fictional users.

- Load the `pandas` module.
- Load the user data from the file contained in the path `datasets/users.csv` and store it as a DataFrame called `users`.
- Print the number of rows (i.e. users).
- Show the first 12 rows in `users`.

Good to know

To complete this project, you need to know how to manipulate strings in `pandas` DataFrames and be familiar with regular expressions. Before starting this project we recommend that you have completed the following courses:

- Data Cleaning in Python
- Regular Expressions in Python

An excellent companion while doing this project is the *Working with text data* page from the `pandas` documentation which you can find here. At the bottom of that page you'll also find a list of all the string functions `pandas` supports.

Task 2: Instructions

Flag the passwords that are too short.

- Add the column `length` to `users` which should list the number of characters in each `password`.
- Flag the users with too short passwords by adding the column `users['too_short']` which should be `True` when `users['length']` is less than 8.
- Print the count of the number of users with passwords that are too short.
- Show the first 12 rows in `users`.

To solve this task, you need to be able to figure out the number of characters in each password. Check the pandas string documentation for a method that does just that.

Task 3: Instructions

Load in the list with the 10,000 most common passwords.

- Read in `datasets/10_million_password_list_top_10000.txt`as a Series and put it in the variable `common_passwords`.
- Take a look at the top 20 common passwords.

Note: The passwords are stored in a plain text file with one password per row. To read this in as a Series you can use the `read_csv` function from `pandas` but to make it return a Series (rather than a DataFrame) you have to set the arguments `header=None` and `squeeze=True`.

Task 4: Instructions

Flag the `user` passwords that are among the top 10,000 used passwords.

- Flag common `user` passwords by adding the column `users['common_password']` which should be `True` when a `password` is one of the `common_passwords`.
- Count and print the number of users using common passwords.
- Show the first 12 rows in `users`.

Task 5: Instructions

Flag the passwords that are among the 10,000 most common English words.

- Read in `datasets/google-10000-english.txt` as a Series and put it in the variable `words`.
- Flag `user` passwords that are common words by adding the column `users['common_word']` which should be `True` when a `password` is one of the `words`. The comparison should be *case-insensitive*.
- Count and print the number of users using common words as passwords.
- Show the first 12 rows in `users`.

Task 6: Instructions

Flag passwords that are the same as the users first or last name.

- Extract users first names from `users['user_name']` into the new column `users['first_name']`.
- Similarly, extract last names into the new column `users['last_name']`.
- Add the column `users['uses_name']` which should be `True` when a `password` is the same as each users' first or last name.
- Count and print the number of users using names as passwords.
- Show the first 12 rows in `users`.

- To extract the first and last names you can use the `.str.extract()` method. Check out the `pandas` documentation under **Extracting substrings** for more info.

Task 7: Instructions

Flag passwords that contain 4 or more repeated characters.

- Add the column `users['too_many_repeats']` which should be True when a `password` has 4 or more repeated characters.
- Take a look at only the `users` with too many repeats.

This task can be solved using the `.str.contains()` method that is described in [the pandas text documentation](). The regexp you need to craft here is a bit tricky, so you may need to revisit this [video lesson]() in Regular Expressions in Python.

Task 8: Instructions

Flag *all* the bad passwords.

- Add the column `users['bad_password']` which should be `True` when a password is bad according to `too_short`, `common_password`, `common_word`, `uses_name`, or `too_many_repeats`.
- Count and print the number of users with bad passwords.
- Show the first 25 bad passwords in `users`.

Task 9: Instructions

- Assign a new password to `new_password` that passes the NIST rules you've implemented in this project.

If you want to know more

- You can [read the full NIST Special Publication 800-63B online]().
- [This blog post]() also gives you a summary of 800-63B.
- [Here is an article]() explaining where the 10,000 common passwords you used in this project come from.
- Finally, [some advice]() on how to come up with a good password.

The Hottest Topics in Machine Learning

Task 1: Instructions

Load the dataset.

- Import the pandas library.
- Load the `papers.csv` file from `datasets/papers.csv` and assign it to the `papers` variable.
- Print the first rows of the DataFrame with the `head` method to verify the file was loaded correctly.

Good to know

Welcome to the Project! While working on the different tasks in this Project, make sure to first read the narrative for each task in the notebook on the right, before reading the more detailed instructions here!

To complete this Project you need to know some Python, pandas, and Natural Language Processing. We recommend one is familiar with the content in DataCamp's Data Manipulation with pandas, and Introduction to Natural Language Processing in Python courses.

For this exercise in particular, you can find more information about loading a CSV file with the pandas library here. You can also look at a similar exercise here on loading CSV files. If you print the first few rows of data, you should see a table with 7 columns.

Task 2: Instructions

Clean the data for further analysis.

- Remove the `id`, `event_type` and `pdf_name` columns.
- Print the first rows of the DataFrame with the `head` method.

If you print the first few rows of data, you should see a table with only 4 columns.

Note that if you remove the required columns on the `papers` variable in-place, running the same code twice will result in an error! Click "Restart & Clear Output" in this Jupyter Notebook's "Kernel" dropdown menu if you run into this issue. Note that removing columns in-place does have the advantage that you don't create the same variable multiple times, which can have an impact on the available memory.

Task 3: Instructions

Visualise the number of papers per year.

- Group the papers by year.
- Count the number of papers per group (i.e. per year).
- Visualise these counts per year in a bar plot.

All of the instructions above can be achieved with 3 lines of code and with methods in the pandas library. The following is one of the approaches you can use.

- Group the papers by the `year` variable with a `groupby` statement and load the result into `groups`.
- Determine the size of each group of papers by applying the `size` statement and loading the result into `counts`.
- Plot the counts as a barplot with the `plot` statement from the `counts` variable.

You can learn more about grouping data in the 'Manipulating DataFrames with pandas' course. The plot methods in pandas are all based on the matplotlib library. More information on plotting can be found here.

Task 4: Instructions

Preprocess the text data.

- Load the regular expression library (re).
- Convert the titles to lowercase using a `map` operation.
- Print the processed titles to verify the results.

The first preprocessing step is already filled in, i.e., removing the punctuation.

If you want to learn more about constructing regular expressions to manipulate strings, check out this course. A more in-depth explanation about regular expressions is provided in the NLP course.

In the interest of time, we will only analyze the titles of the different papers to identify machine learning trends. Note that analyzing the full texts would provide you with more insights.

Task 5: Instructions

Transform the data and create a word cloud.

- Load the `wordcloud` library.
- Convert all the processed titles to a single string.
- Create a `WordCloud` object.
- Generate a word cloud.

Transforming the preprocessed text data to the correct format is required so that it can be handled by the `wordcloud` library. This library takes a long string as input and outputs a word cloud.

Converting all process titles to a single string can be accomplished with a single line of code using the string `join` method.

Task 6: Instructions

Prepare the text for LDA analysis with sk-learn.

- Create a CountVectorizer object with the `stop_words='english'` argument to remove meaningless words.
- Fit and transform the processed titles with the `fit_transform` method. Save the results in the `count_data` variable.
- Plot the most common words with the helper function (plot*10*most*common*words).

Writing the code to transform the titles into document vectors would require a lot of work. Instead, we will use the `CountVectorizer` method in the `sk-learn` library.

Once your code is correct, verify whether the same words occur in the plot and in the word cloud.

If you want to learn more about document representations and why they are necessary, check out the NLP course.

Task 7: Instructions

Play around with different values of the LDA algorithm.

- Tweak the two parameters of LDA (number of topics and number of words).

`number_topics` defines the total number of topics in the LDA model.

`number_words` is only for debugging purposes. It is the number of words that will be printed for each topic. For each topic, the most important words for the topic are selected.

The details of LDA won't be explored here, but you can find more information in the DataCamp Tutorials page.

A follow-up task would be to build an LDA model for the papers per year and try to determine how the topics in the research field have evolved over time.

Task 8: Instructions

True or false?

- Based on the trend in the number of NIPS submissions, it is likely that the number of submissions for NIPS in 2018 will be higher than the previous year. Answer `True` or `False`.

Congratulations, you've made it to the end! If you haven't tried it already, you can check your Project by clicking the 'Check Project' button.

Good luck and keep on learning!

```
# Importing modules

import pandas as pd
```

```
# Read datasets/papers.csv into papers

papers = pd.read_csv("datasets/papers.csv")
```

```
# Print out the first rows of papers

papers.head()
```

```
# Remove the columns

papers.drop(['id', 'event_type', 'pdf_name'], axis=1, inplace=True)
```

```
# Print out the first rows of papers

papers.head()
```

```
# Group the papers by year
groups = papers.groupby('year')

# Determine the size of each group
counts = groups.size()

# Visualise the counts as a bar plot
import matplotlib.pyplot
%matplotlib inline
counts.plot(kind='bar')

# Load the regular expression library
import re

# Print the titles of the first rows
print(papers['title'].head())

# The following line
papers['title_processed'] = papers['title'].map(lambda x: re.sub('[,\.!?]', '', x))

# Convert the titles to lowercase
papers['title_processed'] = papers['title_processed'].map(lambda x: x.lower())

# Print the processed titles of the first rows
papers['title_processed'].head()
```

```
# Import the wordcloud library
import wordcloud

# Join the different processed titles together.
long_string = ' '.join(papers['title_processed'])

# Create a WordCloud object, generate a wordcloud and visualise it
wordcloud = wordcloud.WordCloud()

# Generate a word cloud
wordcloud.generate(long_string)

# Visualize the word cloud
wordcloud.to_image()

# Load the library with the CountVectorizer method
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

# Helper function
def plot_10_most_common_words(count_data, count_vectorizer):
    import matplotlib.pyplot as plt
    words = count_vectorizer.get_feature_names()
    total_counts = np.zeros(len(words))
```

```
    for t in count_data:
        total_counts+=t.toarray()[0]

    count_dict = (zip(words, total_counts))
    count_dict = sorted(count_dict, key=lambda x:x[1], reverse=True)[0:10]
    words = [w[0] for w in count_dict]
    counts = [w[1] for w in count_dict]
    x_pos = np.arange(len(words))

    plt.bar(x_pos, counts,align='center')
    plt.xticks(x_pos, words, rotation=90)
    plt.xlabel('words')
    plt.ylabel('counts')
    plt.title('10 most common words')
    plt.show()

# Initialise the count vectorizer with the English stopwords
count_vectorizer = CountVectorizer(stop_words='english')

# Fit and transform the processed titles
count_data = count_vectorizer.fit_transform(papers['title_processed'])

# Visualise the 10 most common words
plot_10_most_common_words(count_data, count_vectorizer)
```

```
import warnings
warnings.simplefilter("ignore", DeprecationWarning)

# Load the LDA model from sk-learn
from sklearn.decomposition import LatentDirichletAllocation as LDA

# Helper function
def print_topics(model, count_vectorizer, n_top_words):
    words = count_vectorizer.get_feature_names()
    for topic_idx, topic in enumerate(model.components_):
        print("\nTopic #%d:" % topic_idx)
        print(" ".join([words[i]
                        for i in topic.argsort()[:-n_top_words - 1:-1]]))

# Tweak the two parameters below (use int values below 15)
number_topics = 10
number_words = 10

# Create and fit the LDA model
lda = LDA(n_components=number_topics)
lda.fit(count_data)

# Print the topics found by the LDA model
print("Topics found via LDA:")
print_topics(lda, count_vectorizer, number_words)
```

The historical data indicates that:

more_papers_published_in_2018 = True

Visualizing Inequalities in Life Expectancy

Task 1: Instructions

Load libraries and the dataset.

- Load the `dplyr`, `tidyr` and `ggplot2` packages.
- Read `datasets/UNdata.csv` into a data frame and name it `life_expectancy`.
- Print the first few rows of `life_expectancy`.

Good to know

To complete this project, you should be comfortable with the `ggplot2` package, a popular plotting package in R. You will also need `dplyr` and `tidyr`, both popular data manipulation packages in R. If you're not familiar with them, we recommend that you complete Introduction to the Tidyverse and Data Cleaning in R first.

The `options` function allows the user to set and examine a variety of global options which affect the way in which R computes and displays its results. Here it is used to set image size.

Sometimes you may get warning messages when packages are loaded: don't worry. Most of the time they are just informative and you need not take them into account.

A quick way to read a small CSV is to use the `read.csv` function. Take into account that the CSV file is comma-separated.

Task 2: Instructions

Manipulate the dataset to contain male and female life expectancy for each country.

- Filter `life_expectancy` to obtain all records such as `Year` is equal to `2000-2005`.
- Subset the dataset to include just three columns: `Country.or.Area`, `Subgroup`, and `Value`.
- Convert `Subgroup` into two other columns called `Female` and `Male`, reshaping dataset from long to wide.
- Print the first rows of the resulting dataset.

The `dplyr` package contains handy functions to manipulate data. It may be used for example to filter (rows or columns), add new variables, arrange rows by variables or to reduce multiple values down to a single value. On the other hand, the `tidyr` package is useful to reshape data; that is: to change its format from wide to long (and vice versa).

Working with `dplyr` you will need to manage the famous pipe operator `%>%`; if you want to learn more about it, check out [this DataCamp's tutorial](#).

After completing this task, you should have a new data frame called `subdata` with 195 rows and 3 columns, called `Country.or.Area`, `Female`, and `Male`.

Task 3: Instructions

Create a basic scatter plot for male vs. female life expectancy.

- Use the `ggplot` function to initialize a `ggplot` object. Declare subdata as the input data frame and set the aesthetics to represent `Male` on the x-axis and `Female` on the y-axis.
- Add a layer to represent observations with points using `geom_point`.

Once you map variables to axes with `aes()` function, you will only need to add points to your plot using `geom_point()`.

Task 4: Instructions

Add reference lines and axis limits.

- Copy your code from task 3.
- Add a dashed diagonal line that passes by (0, 0) with slope equal to 1.
- Set limit of x-axis from 35 to 85.
- Set limit of y-axis from 35 to 85.

The limits of both axes are the same so the diagonal line will go from the lower left corner to the upper one. It will make the plot easier to understand.

Task 5: Instructions

Add plot titles and axis labels.

- Add the plot title: "Life Expectancy at Birth by Country".
- Add the next caption: "Source: United Nations Statistics Division".
- Set the x-axis label to "Males".
- Set the y-axis label to "Females".

Check out this [page](#) for more information about labels in `ggplot2`.

Task 6: Instructions

Annotate certain countries on the plot with labels.

- Modify the `ggplot(...)` function to set the `label` parameter to `Country.or.Area`.
- Add a label to countries defined by `top_male`.
- Add a label to countries defined by `top_female`.
- Change the plot theme to `theme_bw`.

Labels can be defined with `aes()` inside the `ggplot` function, so you will have to modify the sentence `ggplot(...)`. This will not yet display any text in the plot. You must use `geom_text()` to display it.

There are a wide variety of themes in `ggplot2` that can be used to control the appearance of all non-data components of plots: check out this page for more information. As can be read in the previous link, `theme_bw` is the classic dark-on-light `ggplot2` theme and may work better for presentations displayed with a projector.

Task 7: Instructions

Manipulate the dataset to contain the *difference* in male and female life expectancy for each country.

- Look at the first transformations already defined in the sample code.
- Convert `Sub_Year` into four other columns called `Female_2000_2005`, `Female_1985_1990`, `Male_2000_2005` and `Male_1985_1990`, reshaping dataset from long to wide.
- Create a new variable called `diff_Female` as the difference between `Female_2000_2005` and `Female_1985_1990`.
- Create another variable called `diff_Male` as the difference between `Male_2000_2005` and `Male_1985_1990`.

The sample code already defined performs the following actions:

- Filters `life_expectancy` to obtain all records such as `Year` is equal to `2000-2005` or `1985-1990`.
- Concatenates `Subgroup` and `Year` to create a new column called `Sub_Year` that uses "_" as a separator.
- Replaces the "-" character with the "_" character in variable `Sub_Year` since "-" shouldn't be used to name a column.
- Removes columns `Subgroup` and `Year`.

As you did in task 2, use `spread` function to reshape the dataset from long to wide. Now, you should use values of column `Sub_Year` to name the new columns. There will be as many new columns as there are values of `Sub_Year`.

You can create more than one column inside `mutate` using "," as a separator.

Task 8: Instructions

Plot the variables on the scatter plot and create axis limits.

- Set `diff_Male` column to x-axis
- Set `diff_Female` column to y-axis.
- Set limits of x-axis from -25 to 25 using `scale_x_continuous`.
- Set limits of y-axis from -25 to 25 using `scale_y_continuous`.

After completing this task, you should see a plot similar to the one we did in task 5.

Task 9: Instructions

Add reference lines to the plot.

- Add a dashed horizontal line that passes through point y=0.
- Add a dashed vertical line that passes through point x=0.

Task 10: Instructions

Annotate certain countries on the plot with labels.

- Create a dataset called `bottom` with bottom three rows of `subdata2` ordered by `diff_Male+diff_Female`.
- Add labels for the top three countries using `top`.
- Add labels for the bottom three countries using `bottom`.

Selecting the bottom three rows ordered by `diff_Male+diff_Female` is similar as selecting the top three rows, which is presented in the sample code already.

Congratulations on reaching the end of the project! During your journey, you have applied a wide set of essential functionalities of `ggplot2`. I hope you learned some interesting things about the world we live in. :)

Scout your Athletics Fantasy Team

Task 1: Instructions

Read in the dataset and extract the events of interest.

- Import the `tidyverse` package.
- Read in the `datasets/athletics.csv` dataset using `read_csv()` and assign it to `data`.
- Filter out the Women's Javelin events, drop the `Male_Female` and `Event` columns, and assign the result to `javelin`.
- Print out the `head` and a `summary` of `javelin`.

Good to know

You will be using `tidyr` and `dplyr` packages throughout this project. Both are part of the `tidyverse` suite of packages. It is recommended that you have completed Introduction to the Tidyverse and Data Cleaning in R prior to starting this project.

Task 2: Instructions

Put the data in a tidy format, where each observation is the result from an individual flight.

- Convert `javelin` from "wide" to "long," using `Flight` : `Distance` as your key:value pair.
- Assign the result to `javelin_long`.
- Remove the string "Flight" from each observation in your `Flight` column, so only the flight number remains as a numeric.
- Take a look at your data by calling `head`.

Refer back to Cleaning Data in R to refresh yourself on how to gather columns into key:value pairs.

Task 3: Instructions

For each meet an athlete competed in, we want to know the total distance covered, the standard deviation of each athlete's successful throws and the number of successful throws.

- Filter for observations where the distance is greater than zero.

- Group the data so you can take summary statistics of each athlete's results at a given event.
- Compute the sum of `Distance`, the standard deviation of `Distance` rounded to three places, and the number of non-zero rows. Assign these to `TotalDistance`, `StandardDev` and `Success`, respectively.
- View any ten rows from the middle of the data frame (i.e., do not use `head` or `tail`).

Because you start by retaining the non-zero observations, computing `Success` means finding the number of rows in each `Athlete, EventID` group. The summary statistics can be computed within a single use of a dplyr verb.

Helpful links:

- `sum()` documentation
- `sd()` documentation
- `round()` documentation
- `n()` documentation

Task 4: Instructions

Use the data frame `javelin` to find the difference between the first three throws and second three throws for each athlete in each event.

- Create two new columns: `early` containing the sum of first three throws and `late` containing the sum of the second three throws.
- Create another new column `diff` containing the difference between the two (late minus early).
- Examine the last ten rows.

This can be accomplished within a single `mutate` call like we did with `summarize` in the previous task.

Task 5: Instructions

Join the `diff` column from `javelin` to the `javelin_totals` data frame that contains the other summary statistics.

- `left_join javelin` to `javelin_totals` by the `EventID` and `Athlete` columns.
- Keep the `Athlete`, `TotalDistance`, `StandardDev`, `Success`, and `diff` columns only.
- View the first ten rows.

You will only need the athletes' names and their summary statistics for the next few tasks, so we can drop all the other columns. The five columns listed above contain the necessary information.

If this is your first time using `dplyr` to combine datasets, get a quick overview at the Tidyverse.

Task 6: Instructions

Apply a function to normalize the data across aggregate statistics then compute each athlete's average for each of the normalized stats.

- Define and assign a function called `norm` that operates on `result`.
- Assign the data frame of normalized data to `javelin_norm`.
- Normalize the data by applying `norm` to the columns listed in `aggstats`.
- Find the mean for each athlete on each normalized statistic.

`mutate_at` and `summarize_all` are called scoped variants on the more familiar `mutate`. They apply functions over multiple columns in the data frame. For our purposes, we pass two arguments to `mutate_at`: a vector of column names and the function. We can then use `summarize_all` to take the `mean` of all the columns. Read more at Tidyverse.

Task 7: Instructions

Choose the weights for your statistics. Use them to calculate a total score for each athlete and choose the top five, who will reveal themselves in `javelin_team`. To ensure you evaluate the trade-offs in selecting your team and choose what is most important to you, the four weights must add to ten.

- Assign your `weights` for `TotalDistance`, `StandardDev`, `Success`, and `diff`.
- Create a new column called `TotalScore` by multiplying each summary statistic by its respective weight and adding the products together.
- Arrange the data frame so the highest `TotalScore` is at the top.
- Save the first five rows and the columns for `Athlete` and `TotalScore`.

`slice()` and `select()` may come in handy for the last subtask in this task. To select rows by their position - such as the first five (rather than based on a predicate condition like we would with `filter()`) use the dplyr function, `slice()`.

Task 8: Instructions

Create a data frame `team_stats` containing your players' average statistics from the `javelin_totals` data frame. For now, keep this in a "wide" view so you can examine how they compare to each other.

- Filter the `javelin_totals` data frame to keep those rows containing your athletes. Use the `%in%` operator in your filter statement.
- Take the average by using `summarize_all` as you did in Task 6.
- Examine `team_stats`. Your boss may ask you about this in the next task!

We've created the `pool_stats` data frame containing the maximum and average values for each statistic across the entire pool of athletes we started with.

A lot is going on in how we created `pool_stats`, so take a minute to understand what we did. We found the `max` and `mean` of each numeric column in `javelin_totals` using `sapply`. Then we used the `do.call` function on `cbind` to combine the maximum and average lists. Then we converted it to a data frame. These are all functions from base R. This [StackOverflow post]() has some good back-and-forth comparing `do.call` and `lapply`.

Task 9: Instructions

Create a 2 x 2 grid of plots, each showing a different aggregate statistic. Each plot should have a bar representing each athlete on your team, and a line showing the maximum and average values for that statistic from `pool_stats`.

- Tidy `team_stats` by gathering the data frame into key:value pairs of `Statistic:Aggregate`. Leave the `Athlete` column out of the gathering.
- Use `ggplot` to plot the aesthetics: `Athlete` on the x-axes, `Aggregate` on the y-axis and fill by `Athlete`.
- Add a layer to create a bar plot, where `stat="identity"` and the bars are in the dodge position.
- Use `facet_wrap` to create a 2 x 2 layout based on each `Statistic`. Set `scales` to `free_y` so each facet has y-axis values appropriate to the data.
- Include your team name in the title of the plot, e.g., Sarah's Athletic Club or Grand Rapids Athletic Club.

We created the horizontal lines showing the maximum and average by creating a new geom and passing the data from `pool_stats`. `geom_hline` creates a horizontal line, so it takes `yintercept` as an aesthetic. `labs` assigns text to our labels. The `color` argument within `labs` assigns a title to the legend for the maximum and average lines. We then gave our plot `theme_minimal`, and modified that using the additional `theme()` to remove the titles of the x- and y-axes and the players' names from the x-axis. If you haven't yet, put [Data Visualization with ggplot2]() high on your to-do list!

Task 10: Instructions

The javelin match will use the average data available in the `team_stats` data frame against the single-event data in the `javelin_totals` data frame. Since you've done all the work, you're the home team. Select which three of your five players you want to send out onto the field. Use their average data in the `team_stats` data frame.

- Complete the vector for `HomeTeam`. Use digits 1-5 to identify which of your three players you want to use.
- Take a look at the `AwayTeam` vector to see how we're selecting three players' results at random from the `javelin_totals` data frame to be the away team.

You can look back at Task 8 to remind yourself of their stats and see what digit identifies them. And when you're done, feel free to share your plots with us on Instagram and see if any of your players are on there, too.

Classify Suspected Infection in Patients

Task 1: Instructions

First, let's take a look at the antibiotic data.

- Load the `data.table` package using `library()`.
- Read in `datasets/antibioticDT.csv` using the `data.table` function `fread()`.
- Look at the first 30 rows.

Good to know

This project assumes you have some experience with R and `data.table`, including assignment using `:=`, grouped aggregations using `by`, and the `shift` function. You can learn about `data.table` in the DataCamp course, Data Manipulation in R with data.table.

Helpful links for the duration of the project:

- `data.table` cheat sheet
- This blog post is really helpful for understanding `shift`. The referenced package version is old, but the idea is still the same.

Task 2: Instructions

Identify rows representing "new" antibiotics.

- Use `setorder()` to sort the data by `patient_id`, `antibiotic_type`, and `day_given`. Print and examine the first 40 rows.
- Use `shift` to calculate the last day the antibiotic was given to a patient. Call the new variable, `last_administration_day`.
- Calculate the number of days since the antibiotic was administered to a patient. Call the new variable, `days_since_last_admin`.
- In a two-step process, create a new variable called `antibiotic_new` that is initialized to one, then reset it to zero in rows where it has only been one or two days since the antibiotic was given.

---

*These criteria are a simplified version of the criteria given in a JAMA article by Rhee *et. al.* (2017).

Rhee C, Dantes R, Epstein L, Murphy DJ, Seymour CW, Iwashyna TJ, Kadri SS, Angus DC, Danner RL, Fiore AE, Jernigan JA, Martin GS, Septimus E, Warren DK, Karcz A, Chan C, Menchaca JT, Wang R, Gruber S, Klompas M. *Incidence and Trends of Sepsis in US Hospitals Using Clinical vs Claims Data, 2009-2014*. JAMA. 2017;318(13):1241-1249.

Task 3: Instructions

Investigate the blood culture data.

- Read in `"datasets/blood_cultureDT.csv"`.
- Print the first 30 rows.

Helpful links:

- `fread()` documentation

Task 4: Instructions

Merge the antibiotic data with the blood culture data.

- Make a combined dataset by merging `antibioticDT` with `blood_cultureDT`.
- Sort by `patient_id`, `blood_culture_day`, `day_given`, and `antibiotic_type`.
- Print and examine the first 30 rows.

Helpful links:

- `merge()` documentation

Task 5: Instructions

Make a new variable indicating whether or not the antibiotic administration and blood culture are within two days of each other.

- Make a new variable called `drug_in_bcx_window` which is `1` if the drug was given in the 2-day window and `0` otherwise.

For indicator functions, it can be handy to use `as.numeric()` to convert logical values (TRUE or FALSE) to 0 or 1. Try running the following to see how it works.

```
print("as.numeric(TRUE, FALSE)")
```

Task 6: Instructions

For each patient/blood culture day combination, determine if at least one I.V. antibiotic was given in the +/-2 day window.

- Create a new variable, `any_iv_in_bcx_window`, indicating whether or not an I.V. drug was given within a +/-2 day window of a blood culture day.
- Exclude rows in which the `blood_culture_day` does not have any I.V. drugs in the window.

Use `any()` to check if there are any rows that are both: (1) in +/-2 day window, and (2) have an I.V. drug administered. Use `by =` to make sure this is calculated within each blood culture day for each patient.

Task 7: Instructions

For each blood culture, find the **first day** of potential 4-day antibiotic sequences. This day will be the first day that is both in the window, and a new antibiotic was given.

- Create a new variable called `day_of_first_new_abx_in_window`.
- Remove rows where the day is before this first qualifying day.

Since we're looking for the **day**, start with `day_given` and index from there. Then select only the first, using `[1]`. Indexing by `[1]` only works if the data are sorted by day, which we did in a previous step. Remember, this will be the first day that is both in the window and a new antibiotic was given.

Task 8: Instructions

Make a new dataset that only contains what we need to check the remaining criteria.

- Create a new data.table containing only `patient_id`, `blood_culture_day`, and `day_given`.
- Remove duplicate rows.

Helpful links:

- `unique()` documentation

Task 9: Instructions

Extract the first four antibiotic days.

- Make a new variable, `num_antibiotic_days`, showing the number of antibiotic days each patient/blood culture day combination had.
- Remove blood culture days with less than four antibiotic days (rows).

- Select the first four days (rows) for each blood culture.

The special symbol `.N` counts the number of observations. When used with `by =`, it counts the number of rows in each `by =` group. You can use this to get the number of antibiotic days in each patient-blood culture day.

Selecting the first four rows for each patient ID/blood culture day combination is a little tricky. Use the `data.table` special symbol `.SD`.

```
print(".SD[1:4]")
```

Task 10: Instructions

Find which four-day sequences qualify.

- Make a new 0/1 variable, `four_in_seq`, indicating whether or not the antibiotic sequence has no skips of more than one day.

`diff()` takes a vector of numbers and calculates the difference between each pair of adjacent numbers. If there is a gap of one day, the difference will be two. `max()` of the `diff()` would be useful here too.

Do not forget `as.numeric()` when making `four_in_seq` a 0/1 indicator

Task 11: Instructions

Create a new data frame with one row for each `patient_id` with suspected infection.

- Select the rows which have `four_in_seq` equal to `1`.
- Retain only the `patient_id` column.
- Get rid of duplicates.
- Make a new indicator,`infection`, setting it to `1` for everyone.

To select one column of a data table as a new data table, use `.()` with the column name inside the paratheses.

Helpful links:

- `unique()` documentation

Task 12: Instructions

Find the percentage of presumed serious infections in the data.

- Use `fread()` to read in `"datasets/all_patients.csv"`, which contains a record of all patients who were in the hospital during the same two-week timeframe.
- Merge this dataset with the infection flag data. Make sure to retain all patients.
- The patients who were not in the antibiotic and blood culture data will have missing values for the infection flag. Set these to 0.
- Calculate the percentage of patients who met the criteria for presumed infection.

In one study of almost 3 million hospitalizations, the incidence of presumed serious infection was 14.6% (Rhee 2018).

Helpful links:

- `is.na()` documentation
- `merge()` documentation

-

Hint

Use `all = TRUE` in the merge to make sure patient ids not in both input files are retained.

```
merge(___, ___,
  by = ___,
  all = TRUE)
```

Use `100 * mean()` to calcuate the percentage of people suspected to have a sever infection.

Load packages

library(data.table)

Read in the data

antibioticDT <- fread("datasets/antibioticDT.csv")

Look at the first 30 rows

antibioticDT[1:30]

Sort the data by id, antibiotic type, day

```
setorder(antibioticDT, patient_id, antibiotic_type, day_given)
#antibioticDT[1:40]

# Use shift to calculate the last day a particular drug was administered
antibioticDT[ , last_administration_day := shift(day_given, 1),
  by = .(patient_id, antibiotic_type)]

# Calculate the number of days since the drug was last administered
antibioticDT[ , days_since_last_admin := day_given - last_administration_day]

# Create antibiotic_new with an initial value of one, then reset it to zero as needed
antibioticDT[ , antibiotic_new := 1]
antibioticDT[days_since_last_admin <= 2, antibiotic_new := 0]
# Read in blood_cultureDT.csv
blood_cultureDT <- fread("datasets/blood_cultureDT.csv")

# Print the first 30 rows
blood_cultureDT[1:30]
# Merge antibioticDT with blood_cultureDT
combinedDT <- merge(
  blood_cultureDT,
  antibioticDT,
  all = FALSE,
  by = 'patient_id')
```

```
# Sort by patient_id, blood_culture_day, day_given, and antibiotic_type
setorder(combinedDT, patient_id, blood_culture_day, day_given, antibiotic_type)

# Print and examine the first 30 rows
#combinedDT[1:40]
# Make a new variable called drug_in_bcx_window
combinedDT[ ,
  drug_in_bcx_window :=
          as.numeric(
              day_given - blood_culture_day <= 2
              &
              day_given - blood_culture_day >= -2)]
# Create a variable indicating if there was at least one I.V. drug given in the window
combinedDT[ ,
  any_iv_in_bcx_window := as.numeric(any(route == 'IV' & drug_in_bcx_window == 1)),
  by = .(patient_id, blood_culture_day)]

# Exclude rows in which the blood_culture_day does not have any I.V. drugs in window
combinedDT <- combinedDT[any_iv_in_bcx_window == 1]

# Create a new variable called day_of_first_new_abx_in_window
combinedDT[ ,
    day_of_first_new_abx_in_window :=
        day_given[antibiotic_new == 1 & drug_in_bcx_window == 1][1],
    by = .(patient_id, blood_culture_day)]
```

```
# Remove rows where the day is before this first qualifying day

combinedDT <- combinedDT[day_given >= day_of_first_new_abx_in_window]

# Create a new data.table containing only patient_id, blood_culture_day, and day_given

simplified_data <- combinedDT[ , .(patient_id, blood_culture_day, day_given)]


# Remove duplicate rows

simplified_data <- unique(simplified_data)

# Count the antibiotic days within each patient/blood culture day combination

simplified_data[ , num_antibiotic_days := .N, by = .(patient_id, blood_culture_day)]


# Remove blood culture days with less than four rows

simplified_data <- simplified_data[num_antibiotic_days >= 4]


# Select the first four days for each blood culture

first_four_days <- simplified_data[ , .SD[1:4], by = .(patient_id, blood_culture_day)]

# Make the indicator for consecutive sequence

first_four_days[ , four_in_seq := as.numeric(max(diff(day_given)) < 3), by = .(patient_id,
blood_culture_day)]

# Select the rows which have four_in_seq equal to 1

suspected_infection <- first_four_days[four_in_seq == 1]


# Retain only the patient_id column

suspected_infection <- suspected_infection[ , .(patient_id)]


# Remove duplicates
```

```
suspected_infection <- unique(suspected_infection)

# Make an infection indicator
suspected_infection[ , infection := 1]
# Read in "all_patients.csv"
all_patientsDT <- fread("datasets/all_patients.csv")

# Merge this with the infection flag data
all_patientsDT <- merge(
  all_patientsDT,
  suspected_infection,
  by = "patient_id",
  all = TRUE
)

# Set any missing values of the infection flag to 0
all_patientsDT[is.na(infection) , infection := 0]

# Calculate the percentage of patients who met the criteria for presumed infection
ans <- all_patientsDT[ , 100*mean(infection == 1)]
```

Mobile Games A/B Testing with Cookie Cats

Task 1: Instructions

Read in and take a look at the Cookie cats AB-test data.

- Import the `pandas` module.
- Read in the AB-test data `datasets/cookie_cats.csv` as a DataFrame and assign it to `df`.
- Take a look at the first few rows of `df`.

Good to know

To complete this project, you need to know some python and be familiar with `pandas` DataFrames and bootstrap analysis. Here are relevant DataCamp exercises if you need to brush up your skills:

- From Intermediate Python for Data Science
  - Reading in a csv-file
  - Selecting columns using `[]`
- From Data Manipulation with pandas
  - Inspecting DataFrames
  - Line plots using pandas
- From Statistical Thinking in Python (Part 2)
  - Bootstrap analysis

Even if you've taken these courses, you will still find this project challenging unless you use some external *documentation*. Here is a pandas cheat sheet summarizing the basics of `pandas` DataFrames. (You could also look at the official pandas documentation but be aware that it is *very technical*.)

Finally, know that *Google is your friend* and a good search pattern is **example of ??? in pandas** where **???** is whatever you need to do. For instance, if you need to read in a csv file you could search for *example of reading a csv file in pandas*.

-

Hint

If you import the pandas module like this:

```
import pandas as pd
```

You can use the `pd.read_csv` function to read in data stored in `csv`-files and to print out the first few rows of a DataFrame use the `.head()` method.

Task 2: Instructions

Count the players in each AB-group.

- `count` and display the number of rows in each AB-group defined by `version`.

For some help, check out the `groupby` and the `count()` methods in the pandas cheat sheet.

-

Hint

Here is some code to get you started:

```
df.groupby('...')['...'].count()
```

Task 3: Instructions

Plot the distribution of game rounds.

- Group by `'sum_gamerounds'` and then `count` `userid`. Assign the result to `plot_df`.
- Use the pandas `.plot()` method to plot the first 100 rows of `plot_df` with `sum_gamerounds` on the X-axis and `userid` on the Y-axis.
- Assign the plot to `ax`.
- [*Optional*] Set the X-label and Y-label to something informative.

To select the first 100 rows of `plot_df` you can use the `.head()` method, but you have to set the first argument to `head` to 100.

-

Hint

Here is some code to get you started:

```
df.groupby('...')['...'].count()
```

and

```
plot_df.head(n=...).plot(x="...", y="...")
```

Task 4: Instructions

Calculate overall 1-day retention.

- Calculate and display the proportion of `True` values in the column `df['retention_1']`.

There are two ways of calculating the proportion of `True` values in a column. Assuming the column is `my_df['a_col']`:

1. `my_df['a_col'].sum() / my_df['a_col'].count()` : That is, `sum` the column and divide by the total number of values. This works because when using `.sum()` the `True`/`False` values will first be converted to `1`/`0`.
2. `my_df['a_col'].mean()` : This works because the mean is calculated by summing the values and dividing by the total number of values.

-

Hint

Some code to get you started:

```
df['...'].sum() / df['...'].count()
```

Task 5: Instructions

Calculate 1-day retention for each AB-group.

- Calculate and display the proportion of `True` values in the column `df['retention_1']` *for each group defined by `'version'`*.

To calculate this, you can simply copy-and-paste the solution from task 4. You will then have to modify the code by adding `.groupby('version')` in the right places.

-

Hint

Here is some code to get you started:

```
df.groupby('...')['...'].sum() / df.groupby('...')['...'].count()
```

Task 6: Instructions

Do a bootstrap analysis of 1-day retention and plot the result.

- Perform 500 bootstrap replications and `append` them to `boot_1d`. Each replication should re-sample from `df` and calculate the `mean` retention for each AB-group defined by `version`.
- Turn the list `boot_1d` into a DataFrame.
- Use the `pandas` `.plot()` method to plot `boot_1d` as a Kernel Density Plot.

In the code, each `boot_mean` should be calculated by first sampling `df` with `.sample`, and then grouping by `version` and calculating the `mean` of `retention_1`. All of this can be done in one line. When using `sample` remember to set `frac=1` and `replace=True`.

To turn `boot_1d` into a DataFrame, you could simply use the pandas `DataFrame` creation function.

For how to create a density plot from a `pandas` DataFrame take a look at [this documentation]().

This is a tricky task, so don't be afraid to look at the ↓ hint ↓ below. There's no penalty!

-

Hint

Here is some code to get you started:

```
boot_mean = df.sample(frac=1, replace=True).groupby('...')['...'].mean()
```

Task 7: Instructions

Calculate and plot the % difference in 1-day retention between the two AB-groups.

- Calculate the bootstrap % difference in 1-day retention by: Calculating the difference between `gate_30` and `gate_40`, dividing by `gate_40`, and multiplying by 100 to turn the proportion into a percentage.
- Assign the resulting column to `boot_1d['diff']`.
- Plot `boot_1d['diff']` as a Kernel Density Plot using the `.plot` method and assign the result to `ax`.
- [*Optional*] Give the plot a nice X-label.

Since mathematical operators are vectorized in `pandas`, we can easily write expressions that calculate values by row. For example, if `my_df` is this DataFrame:

	a	b
0	1.1	1
1	2.4	2
2	2.7	3

You can calculate how much larger (in %) each `a` is compared to `b`:

```
my_df['diff'] = (
    (my_df['a'] - my_df['b']) /
         my_df['b'] * 100 )
```

Which yields this:

	a	b	diff
0	1.1	1	10.0
1	2.4	2	20.0
2	2.7	3	-10.0

-

Hint

here is some code to get you started:

```
boot_1d['diff'] = (boot_1d['...'] - boot_1d['...']) /  boot_1d['...'] * 100
```

Task 8: Instructions

Calculate the probability that a gate at level 30 gives higher 1-day retention.

- Calculate the proportion of `boot_1d['diff']` that is above `0.0`. Assign it to `prob`.
- Pretty print `prob` so that it is formatted as a percentage.

If `vals` is a Series of numbers you can calculate the proportion of `vals` above `0.0` like this:

```
(vals > 0).sum() / len(vals)
```

or like this:

```
(vals > 0).mean()
```

For a convenient way of pretty printing percentages see [this StackOverflow answer](#).

-

Hint

Here is some code to get you started:

```
prob = (boot_1d['...'] > 0).sum() / len(...)
```

Task 9: Instructions

Calculate the 7-day retention for each AB-group.

- Calculate and display the proportion of `True` values in the column `df['retention_7']` *for each group defined by* `'version'`.

The solution to this task is *almost* the same as in task 5.

-

Hint

Just check the hint and your solution for task 5.

Task 10: Instructions

Do a bootstrap analysis for the difference in 7-day retention.

- Copy and paste in the code from task 6, 7, and 8.
- Modify the code to do a bootstrap analysis of 7-day retention (`retention_7`) instead of 1-day retention. The list/DataFrame to hold the bootstrap samples should be named `boot_7d`.

In this task, you will bring everything together that you've coded in task 6, 7, and 8. But now you will apply the bootstrap analysis to a new variable: `retention_7`.

-

Hint

So this task can be solved by the code you've developed in task 6, 7, and 8. But modifying copied code can be tricky. Make sure all the names match up and that you've changed all `retention_1` to `retention_7`, and all `boot_1d` to `boot_7d`.

Task 11: Instructions

Given the data and the analysis should we move the gate to level 40, or keep it at level 30?

- Set `move_to_level_40` to either `True` or `False` to indicate your decision.

If you want to know more

Now that you've analyzed some Cookie Cats data, maybe you want to take the actual game for a spin. Perhaps you can think of more things you would like to AB-test in this game?

```
# Importing pandas

import pandas as pd



# Reading in the data

df = pd.read_csv('datasets/cookie_cats.csv')



# Showing the first few rows

df.head()

# Counting the number of players in each AB group.

df.groupby('version')['userid'].count()

# This command makes plots appear in the notebook

%matplotlib inline



# Counting the number of players for each number of game rounds

plot_df = df.groupby('sum_gamerounds')['userid'].count()



# Plotting the distribution of players that played 0 to 100 game rounds

ax = plot_df.head(n=100).plot(x="sum_gamerounds", y="userid")

ax.set_xlabel("Game Rounds")

ax.set_ylabel("User Count")

# The % of users that came back the day after they installed

df['retention_1'].sum() / df['retention_1'].count()
```

```
# Calculating 1-day retention for each AB-group
df.groupby('version')['retention_1'].sum() / df.groupby('version')['userid'].count()
# Creating an list with bootstrapped means for each AB-group
boot_1d = []
for i in range(500):
    boot_mean = df.sample(frac=1, replace=True).groupby('version')['retention_1'].mean()
    boot_1d.append(boot_mean)

# Transforming the list to a DataFrame
boot_1d = pd.DataFrame(boot_1d)

# A Kernel Density Estimate plot of the bootstrap distributions
boot_1d.plot(kind='kde')
# Adding a column with the % difference between the two AB-groups
boot_1d['diff'] = (boot_1d['gate_30'] - boot_1d['gate_40']) /  boot_1d['gate_40'] * 100

# Ploting the bootstrap % difference
ax = boot_1d['diff'].plot(kind = 'kde')
ax.set_xlabel("% difference in means")
# Calculating the probability that 1-day retention
# is greater when the gate is at level 30.
prob = (boot_1d['diff'] > 0).sum() / len(boot_1d)

# Pretty printing the probability
'{:.1%}'.format(prob)
```

```
# Calculating 7-day retention for both AB-groups
df.groupby('version')['retention_7'].sum() / df.groupby('version')['userid'].count()
# Creating a list with bootstrapped means for each AB-group
boot_7d = []
for i in range(500):
    boot_mean = df.sample(frac=1, replace=True).groupby('version')['retention_7'].mean()
    boot_7d.append(boot_mean)

# Transforming the list to a DataFrame
boot_7d = pd.DataFrame(boot_7d)

# Adding a column with the % difference between the two AB-groups
boot_7d['diff'] = (boot_7d['gate_30'] - boot_7d['gate_40']) /  boot_7d['gate_30'] * 100

# Ploting the bootstrap % difference
ax = boot_7d['diff'].plot(kind = 'kde')
ax.set_xlabel("% difference in means")

# Calculating the probability that 7-day retention is greater when the gate is at level 30
prob = (boot_7d['diff'] > 0).sum() / len(boot_7d)

# Pretty printing the probability
'{:.1%}'.format(prob)
# So, given the data and the bootstrap analysis
# Should we move the gate from level 30 to level 40 ?
```

```
move_to_level_40 = False # True or False ?
```

Who Is Drunk and When in Ames, Iowa?

Task 1: Instructions

First, get the data into your workspace and summarize it by year. Do you notice a pattern over time?

- Read `datasets/breath_alcohol_ames.csv` into your workspace using the `read_csv()` function. Save it as `ba_data`.
- Count how many tests were administered in each `year` using the `count()` function, creating a new data set called `ba_year`.

Good to know

If you have taken the Introduction to the Tidyverse, this project is for you! You may also find this data transformation cheat sheet and this `ggplot2` cheat sheet helpful. For even more, visit the `tidyverse` documentation.

The `count()` function isn't explicity taught in Introduction to the Tidyverse, however all of the tools required to understand `count()` are. This is an excellent opportunity to apply these skills. Read up on `count()` here. If you're stuck afterward, check the hint below.

The column returned from `count()` will be named `n`. Keep this column name for testing purposes.

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Hint

The statement below will count up all rows in `data` that belong to each value of `var` and return a table of counts.

```
data %>% count(var)
```

So will this one.

```
data %>% group_by(var) %>% count()
```

Task 2: Instructions

Count the data by location to see which department administers more breathalyzer tests.

- Group and summarize the data by `location` with the `count()` function. Save this data as `pds`.

The column returned from `count()` will be named `n`. Keep this column name for testing purposes.

Helpful links:

- `count()`
-

Hint

The statement below will count up all rows in `data` that belong to each value of `var` and return a table of counts in descending order.

```
data %>% count(var, sort = TRUE)
```

Task 3: Instructions

Summarize the data and create a bar chart of number of tests by hour of the day.

- Using `count()`, summarize the data by `hour` and sort it according to the total number of arrests from high to low. Save this as `hourly`.
- Create a bar chart of total breath alcohol tests by `hour` of the day using `ggplot2` and the `hourly` data. Use `weight =` in the asethetics

---

The function `geom_bar()` uses the `weight` argument in `aes()` to map the height of the bar to a variable in the dataset.

-

Hint

Use the `sort = TRUE` argument in `count()` to arrange the data in descending order, then use `weight =` in the `aes()` to map the height of the bar to a variable in the dataset.

```
new_data  <- data %>%
 count(var, sort = TRUE)

ggplot(hourly, aes(x = ...., weight = n)) +
    geom_bar()
```

Task 4: Instructions

We'll look at the `month` variable to determine the most popular time of year for breathalyzer tests.

- Using `count()` and the `sort =` argument to summarize the data by `month` and sort it according to the total number of arrests from high to low. Save this as `monthly`.
- Create a bar chart of total breath alcohol tests by month of the year using `ggplot2` and the `monthly` data.

The values of the `month` variable are (1, 2, …, 12) and correspond to the months (January, February, …, December).

The `month` variable of `monthly` is made a factor variable to make the bar chart easier to read. For more information on factor variables, check out Chapter 4 of the Introduction to R course.

-

Hint

Use the `sort = TRUE` argument in `count()` to arrange the data in descending order, then use `weight =` in the `aes()` to map the height of the bar to a variable in the dataset.

```
new_data  <- data %>%
 count(var, sort = TRUE)

ggplot(hourly, aes(x = ...., weight = n)) +
    geom_bar()
```

Task 5: Instructions

Compare test frequency and results for men vs. women.

- Count the number of tests by gender using `count()` to see which gender took more tests.
- Remove the `NA` values in the `gender` variable with `filter()` and save the results as `clean_gender`.
- In `clean_gender`, use `mutate()` to create a new variable called `meanRes`, the mean of the two tests `Res1` and `Res2`. Save the result as a new data set called `mean_bas`.
- Using `mean_bas`, create boxplots of mean results for men and women. Use `meanRes` on the y-axis and `gender` on the x-axis.

- The function `is.na(x)` returns `TRUE` if the object `x` is `NA`.

- The `mutate()` function can create new columns in a data set from existing columns.
-

Hint

The following command subsets data to include only values of `var` that are not `NA`.

```
data %>% filter(!is.na(var))
```

Compute the mean the old-fashioned way! i.e. `meanRes = (Res1 + Res2) / 2`

`geom_boxplot()` is the function for boxplots.

Task 6: Instructions

Determine what percent of the breathalyzer tests in the data are above the legal limit.

- Filter the `ba_data` to include only tests where one or both of `Res1`, `Res2` are greater than 0.08. Call this filtered data `duis`.
- Create a variable, `p_dui`, the proportion of all tests that would have resulted in a DUI.

Check out this video for information on Logical Operators in R (e.g. `|`, which means "or") and this video for information on Relational Operators in R (e.g. `>`, which means "greater than").

- `nrow(dat)` returns the number of rows in `dat`.
- `data %>% filter(var1 == 1 | var2 ==2 )` returns the rows of `data` where `var1` is 1 OR `var2` is 2.
-

Hint

Use the logical OR operator (`|`) in the `filter` statement with the greater than operator (`>`).

```
ba_data %>% filter(Res1 > 0.08 | Res2 > ....)
```

Task 7: Instructions

Create a date variable and determine the week in the year each test occurred.

- Create a new column in `ba_data` called `date` by using `paste()` to combine the date variables. Do this using `mutate()` and the `lubridate` function `ymd()` to make a `date` column.

- Using the new `date` variable, create another new column in `ba_data` called `week` with the `lubridate` function `week()`

The function `ymd()` from `lubridate` can take values like `'YYYY-MM-DD'` and make date objects, and `week()` takes date variables and gets the integer value of the week in the year.

For more on working with dates and times in R, see the course [Working with Dates and Times in R](#).

-

Hint

Try wrapping the functions around each other! For example, the following code creates a date column in `dat` called `da` that is the date parsing of the variables `y`, `m`, `d` that have been pasted together into a character vector. Then create the week data.

```
# Create a date column
dat <- dat %>% mutate(da = ymd(paste(y, m, d, sep = '-')))

# Create a column that is the week in year in da
dat <- dat %>% mutate(week = week(da))
```

Task 8: Instructions

Create a time series plot to compare weeks across years.

- Using `count()`, count up the number of arrests per week in *each* year. Save the summarized data as `weekly`.
- Create a line plot with `week` on the x-axis and the count variable on the y-axis, colored by `year`.

`year` is made a factor to make seeing the year groups in the plot easier. For more information on factor variables, check out [Chapter 4](#) of the Introduction to R course.

The `mutate()` function can overwrite variables by using the same column name.

`geom_point()` and `scale_x_continuous` were added to make the plot more readable.

-

Hint

Group by both the `week` and the `year` variables when using `count()`.

To count a dataset called `data` by both `var1` and `var2`, the code would look like this:

```
data %>% count(var1, var2)
```

Task 9: Instructions

Run the provided code to plot the previous time series chart with annotations pointing to the last two VEISHEA weeks in Iowa State's history.

- In your opinion, `TRUE` or `FALSE`: canceling VEISHEA was the right decision based on the amount of breathalyzer tests alone.

Congratulations on making it to the end! If you haven't checked your project yet, you can do so by clicking the yellow "Check Project" button.

Good luck! :)

-

Hint

Run the provided `ggplot()` code then make a decision by inputting `TRUE` or `FALSE`.

A New Era of Data Analysis in Baseball

Task 1: Instructions

Load the CSV files, which hold the Statcast data for each player, into pandas DataFrames.

- Load `datasets/judge.csv` into a DataFrame and assign it to the variable `judge`.
- Load `datasets/stanton.csv` into a DataFrame and assign it to the variable `stanton`.

Good to know

This Project requires that you know your way around Python, pandas, and data visualization. We recommend the following courses as prerequisites:

- Intermediate Python for Data Science
- Introduction to Data Visualization with Python

MLB.com's Statcast glossary (MLB stands for Major League Baseball) may be helpful at various points *throughout* the Project. Through accessible text and video, they explain baseball concepts in more detail than the Project Notebook. Links to specific glossary pages will be provided throughout the Project.

-

Hint

If you load in the pandas module:

```
import pandas as pd
```

You can use pandas' `read_csv` function (documentation) to read in data stored in CSV files like so:

```
my_data = pd.read_csv('my_data.csv')
```

Task 2: Instructions

Display the last five rows of the `judge` DataFrame.

- Use pandas' `tail` method to display the last five rows of `judge`.

The last five rows of the `judge` DataFrame are displayed instead of the first five because they contain more interesting data.

Helpful links:

- pandas' `tail` method (documentation)
-

Hint

pandas' `tail` method prints out the last five rows by default. If `df_1` is a DataFrame, printing the last five rows can be done like so:

```
df_1.tail()
```

Task 3: Instructions

Isolate each player's batted ball events for the 2017 season.

- Filter `judge` to include pitches from 2017 only and select the `events` column. Store the result in a variable called `judge_events_2017`.
- Using the `value_counts` method, print out the count of unique values for `judge_events_2017`.
- Filter `stanton` to include pitches from 2017 only and select the `events` column. Store the result in a variable called `stanton_events_2017`.
- Using the `value_counts` method, print out the count of unique values for `stanton_events_2017`.

Helpful links:

- pandas' `value_counts` method (documentation)
-

Hint

Each file has a column named `game_year`.

The structure of the first line of code to complete could be (where capitalized words should be replaced):

```
# All of Aaron Judge's batted ball events in 2017
judge_events_2017 = judge.loc[judge['COLUMN NAME'] == YEAR].events
```

The structure of the second line of code to complete could be:

```
print(judge_events_2017._____________())
```

Task 4: Instructions

Isolate each player's home runs then plot exit velocity vs. launch angle.

- Filter the `judge` and `stanton` DataFrames to include home runs only.
- Create a figure using seaborn's `regplot` function with two scatter plots of launch speed vs. launch angle, one for each player's home runs.
- Create a figure using seaborn's `kdeplot` function with two KDE plots of launch speed vs. launch angle, one for each player's home runs.

Exit velocity is also known as launch speed, where `launch_speed` is the name of the column in each file.

Helpful links:

- seaborn's `regplot` function (documentation)
- seaborn's `kdeplot` function (documentation)
-

Hint

The event "home_run" is found in the `events` column.

The structure of the first line of code to complete could be (where capitalized words should be replaced):

```
# Filter to include home runs only
judge_hr = judge.loc[judge['COLUMN NAME'] == 'EVENT']
```

Task 5: Instructions

Plot the pitch velocities of each player's home runs on box plots.

- Concatenate `judge_hr` and `stanton_hr` using pandas' `concat` function and store the result in a variable called `judge_stanton_hr`.
- Create a boxplot using seaborn's `boxplot` function that describes the pitch velocity of each player's home runs. Make the color argument 'tab:blue'.

Pitch velocity is also known as release speed, where `release_speed` is the name of the column in each file.

Helpful links:

- pandas' `concat` function (documentation)

- seaborn's `boxplot` function (documentation)
-

Hint

If `df_1` is a DataFrame and `df_2` is another DataFrame, concatenating these two DataFrames is done like so (note the list used as the `objs` argument):

```
pd.concat([df_1, df_2])
```

The boxplot parameters required for this plot are `x`, `y`, `data`, and `color`.

Task 6: Instructions

Create a function that returns the x-coordinate of a pitch zone.

- Return the x-coordinate for the left third of strike zone.
- Return the x-coordinate for the middle third of strike zone.
- Return the x-coordinate for the right third of strike zone.

While you should ignore zones 11, 12, 13, and 14 for this plotting task, setting up conditionals to filter these out now isn't necessary. That will come in an upcoming task!

`zone` is the name of the column that holds each pitch's zone data.

It may be helpful to draw the zone and label the x- and y-coordinates by hand.

-

Hint

The left third of the zone (zone numbers 1, 4, and 7) should have an x-coordinate of 1 since the origin of the plot (i.e. point (0,0)) is the bottom left corner.

Task 7: Instructions

Create a function that returns the y-coordinate of a pitch zone.

- Return the y-coordinate for the upper third of strike zone.
- Return the y-coordinate for the middle third of strike zone.
- Return the y-coordinate for the lower third of strike zone.

While you should ignore zones 11, 12, 13, and 14 for this plotting task, setting up conditionals to filter these out now isn't necessary. That will come in an upcoming task!

`zone` is the name of the column that holds each pitch's zone data.

It may be helpful to draw the zone and label the x- and y-coordinates by hand.

-

Hint

The lower third of the zone (zone numbers 7, 8, and 9) should have a y-coordinate of 1 since the origin of the plot (i.e. point (0,0)) is the bottom left corner.

Task 8: Instructions

Assign Cartesian coordinates to the strike zone and plot pitches that resulted in Judge home runs as a 2D histogram.

- Apply `assign_x_coord` to `judge_strike_hr` to create a new column called `zone_x`.
- Apply `assign_y_coord` to `judge_strike_hr` to create a new column called `zone_y`.
- Plot Judge's home run zone as a 2D histogram (using matplotlib's `hist2d` function) with a colorbar.

Helpful links:

- pandas' `apply` method (documentation)
- How to use pandas' `apply` method (StackOverflow answer)
- matplotlib's `hist2d` function (documentation)

-

Hint

The structure of the first line of code to complete could be (where capitalized words should be replaced):

```
judge_strike_hr['zone_x'] = judge_strike_hr.apply(FUNCTION, axis=1)
```

The arguments for `x` and `y` in `hist2d` should be the `zone_x` and `zone_y` columns represented as pandas Series.

Task 9: Instructions

Assign Cartesian coordinates to the strike zone and plot pitches that resulted in Stanton home runs as a 2D histogram.

- Apply `assign_x_coord` to `stanton_strike_hr` to create a new column called `zone_x`.
- Apply `assign_y_coord` to `stanton_strike_hr` to create a new column called `zone_y`.
- Plot Stanton's home run zone as a 2D histogram (using matplotlib's `hist2d` function) with a colorbar.

Helpful links:

- pandas' `apply` method (documentation)
- How to use pandas' `apply` method (StackOverflow answer)
- matplotlib's `hist2d` function (documentation)
-

Hint

The structure of the first line of code to complete could be (where capitalized words should be replaced):

```
stanton_strike_hr['zone_x'] = stanton_strike_hr.apply(FUNCTION, axis=1)
```

The structure of the third line of code to complete could be (where capitalized words should be replaced):

```
plt.hist2d(stanton_strike_hr.COLUMN, stanton_strike_hr.COLUMN, bins = 3,
cmap='Blues')
```

Task 10: Instructions

Answer the following question: "Should opposing pitchers be wary of Aaron Judge and Giancarlo Stanton?"

- Store a Boolean value (True or False) in `should_pitchers_be_scared`.

These editorial images (Judge, Stanton) from Getty Images are awesome. The best are the ones with each player next to normal sized humans.

If you'd like more Statcast content to digest, this video (A culmination of special Statcast records in 2017 from MLB's YouTube channel) and this article (Major League Baseball's Statcast Can Break Sabermetrics by Emma Baccellieri) are excellent.

Hint

Aaron Judge and Giancarlo Stanton are pretty scary…

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline


# Load Aaron Judge's Statcast data

judge = pd.read_csv('datasets/judge.csv')


# Load Giancarlo Stanton's Statcast data

stanton = pd.read_csv('datasets/stanton.csv')

# Display all columns (pandas will collapse some columns if we don't set this option)

pd.set_option('display.max_columns', None)


# Display the last five rows of the Aaron Judge file

judge.tail()

# All of Aaron Judge's batted ball events in 2017

judge_events_2017 = judge.loc[judge['game_year'] == 2017].events

print("Aaron Judge batted ball event totals, 2017:")

print(judge_events_2017.value_counts())


# All of Giancarlo Stanton's batted ball events in 2017

stanton_events_2017 = stanton.loc[stanton['game_year'] == 2017].events

print("\nGiancarlo Stanton batted ball event totals, 2017:")

print(stanton_events_2017.value_counts())
```

```
# Filter to include home runs only

judge_hr = judge.loc[judge['events'] == 'home_run']

stanton_hr = stanton.loc[stanton['events'] == 'home_run']



# Create a figure with two scatter plots of launch speed vs. launch angle, one for each player's
home runs

fig1, axs1 = plt.subplots(ncols=2, sharex=True, sharey=True)

sns.regplot(x='launch_angle', y='launch_speed', fit_reg=False, color='tab:blue',
data=judge_hr, ax=axs1[0]).set_title('Aaron Judge\nHome Runs, 2015-2017')

sns.regplot(x='launch_angle', y='launch_speed', fit_reg=False, color='tab:blue',
data=stanton_hr, ax=axs1[1]).set_title('Giancarlo Stanton\nHome Runs, 2015-2017')



# Create a figure with two KDE plots of launch speed vs. launch angle, one for each player's
home runs

fig2, axs2 = plt.subplots(ncols=2, sharex=True, sharey=True)

sns.kdeplot(judge_hr.launch_angle, judge_hr.launch_speed, cmap="Blues", shade=True,
shade_lowest=False, ax=axs2[0]).set_title('Aaron Judge\nHome Runs, 2015-2017')

sns.kdeplot(stanton_hr.launch_angle, stanton_hr.launch_speed, cmap="Blues", shade=True,
shade_lowest=False, ax=axs2[1]).set_title('Giancarlo Stanton\nHome Runs, 2015-2017')

# Combine the Judge and Stanton home run DataFrames for easy boxplot plotting

judge_stanton_hr = pd.concat([judge_hr, stanton_hr])



# Create a boxplot that describes the pitch velocity of each player's home runs

sns.boxplot(x='player_name', y='release_speed', color='tab:blue',
data=judge_stanton_hr).set_title('Home Runs, 2015-2017')

def assign_x_coord(row):

    """

    Assigns an x-coordinate to Statcast's strike zone numbers. Zones 11, 12, 13,

    and 14 are ignored for plotting simplicity.
```

```
    """
    # Left third of strike zone
    if row.zone in [1, 4, 7]:
        return 1
    # Middle third of strike zone
    if row.zone in [2, 5, 8]:
        return 2
    # Right third of strike zone
    if row.zone in [3, 6, 9]:
        return 3
def assign_y_coord(row):
    """
    Assigns a y-coordinate to Statcast's strike zone numbers. Zones 11, 12, 13,
    and 14 are ignored for plotting simplicity.
    """
    # Upper third of strike zone
    if row.zone in [1, 2, 3]:
        return 3
    # Middle third of strike zone
    if row.zone in [4, 5, 6]:
        return 2
    # Lower third of strike zone
    if row.zone in [7, 8, 9]:
        return 1
```

```
# Zones 11, 12, 13, and 14 are to be ignored for plotting simplicity
judge_strike_hr = judge_hr.copy().loc[judge_hr.zone <= 9]

# Assign Cartesian coordinates to pitches in the strike zone for Judge home runs
judge_strike_hr['zone_x'] = judge_strike_hr.apply(assign_x_coord, axis=1)
judge_strike_hr['zone_y'] = judge_strike_hr.apply(assign_y_coord, axis=1)

# Plot Judge's home run zone as a 2D histogram with a colorbar
plt.hist2d(judge_strike_hr.zone_x, judge_strike_hr.zone_y, bins = 3, cmap='Blues')
plt.title('Aaron Judge Home Runs on\n Pitches in the Strike Zone, 2015-2017')
plt.gca().get_xaxis().set_visible(False)
plt.gca().get_yaxis().set_visible(False)
cb = plt.colorbar()
cb.set_label('Counts in Bin')

# Zones 11, 12, 13, and 14 are to be ignored for plotting simplicity
stanton_strike_hr = stanton_hr.copy().loc[stanton_hr.zone <= 9]

# Assign Cartesian coordinates to pitches in the strike zone for Stanton home runs
stanton_strike_hr['zone_x'] = stanton_strike_hr.apply(assign_x_coord, axis=1)
stanton_strike_hr['zone_y'] = stanton_strike_hr.apply(assign_y_coord, axis=1)

# Plot Stanton's home run zone as a 2D histogram with a colorbar
plt.hist2d(stanton_strike_hr.zone_x, stanton_strike_hr.zone_y, bins = 3, cmap='Blues')
plt.title('Giancarlo Stanton Home Runs on\n Pitches in the Strike Zone, 2015-2017')
```

```
plt.gca().get_xaxis().set_visible(False)

plt.gca().get_yaxis().set_visible(False)

cb = plt.colorbar()

cb.set_label('Counts in Bin')

# Should opposing pitchers be wary of Aaron Judge and Giancarlo Stanton

should_pitchers_be_scared = True
```

A Visual History of Nobel Prize Winners

Task 1: Instructions

Load the required libraries and the Nobel Prize dataset.

- Load the `tidyverse` library.
- Use `read_csv` (*not* `read.csv`) to read in `datasets/nobel.csv` and save it into `nobel`.
- Show the `head` of `nobel`, that is, the first couple of prize winners.

Make sure to use `read_csv` (with an underscore) to read in the data. The `read.csv` function, which is built into R, has a number of problems which the new `read_csv` function avoids.

Good to know

This Project assumes you have used the `dplyr` and `ggplot2` packages and that you are familiar with the pipe operator (`%>%`). Before taking on this Project, we recommend that you have completed the course Introduction to the Tidyverse.

RStudio has created some very helpful cheat sheets, including two that will be helpful for this Project: Data Wrangling and Data Visualization with ggplot2. We recommend that you keep them open in a separate tab to make it easy to refer to them.

-

Hint

If you've loaded in the `tidyverse`

```
library(tidyverse)
```

you can read in `path_to/my_data.csv` like this:

```
my_data <- read_csv("path_to/my_data.csv")
```

Task 2: Instructions

Count up the Nobel Prizes. Also, split by `sex` and `birth_country`.

- Count and display the number of rows/prizes using the `count()` function.
- Count and display the number of rows/prizes, grouped by `sex`.

- Count the number of rows/prizes, grouped by `birth_country`. Arrange the result by no. prizes in descending order and display the first 20 rows using `head(20)`.

For how to use the `group_by` function to group by a column check out *Group Cases* in [the dplyr cheat sheet](). For how to arrange rows, take a look at *Arrange Cases* in the same cheat sheet.

-

Hint

Here is how to solve the most complicated part of this task:

```
nobel %>%
    group_by(birth_country) %>%
    count()  %>%
    arrange(desc(n))  %>%
    head(20)
```

See if you can figure out the two easier parts yourself!

Task 3: Instructions

Calculate the proportion of USA born winners per decade starting from the `nobel` dataset and put the result into `prop_usa_winners`.

- Add a `usa_born_winner` column to `nobel`, where the value is `TRUE` when `birth_country` is `"United States of America"`.
- Add a `decade` column to `nobel` showing the decade the prize was awarded (`1953` should become `1950`, for example).
- Group by `decade` and use `summarize` to add the column `proportion` to `nobel`. `proportion` should contain the proportion of `usa_born_winners` for each `decade`.
- Display / print out `prop_usa_winners`.

You can use `mutate` for the first two bullet points.

To calculate the proportion of `TRUE` values you can use the `mean` function. If the column includes `NA` values, you would have to use the `na.rm` argument. Here's how you could use `mean` together with `summarize`:

```
my_data  %>%
  group_by(my_categorical_variable)  %>%
  summarize(proportion =
      mean(is_winner, na.rm = TRUE))
```

-

Hint

To add the column `usa_born_winner`, you can use `mutate` like this:

```
prop_usa_winners <- nobel %>%
    mutate(usa_born_winner = birth_country == "United States of America")
```

To go from year to decade, you can fiddle around with multiplication, division, and the `floor` function. For example, here is one way you could calculate the decade of `year`:

```
year = 1873
decade = floor(year / 10) * 10
# decade is now 1870
```

Task 4: Instructions

Plot the proportion of USA born winners per decade.

- Use `ggplot` to plot `prop_usa_winners` with `decade` on the x-axis and `proportion` on the y-axis as a line-and-dot-plot. That is, add both `geom_line()` and `geom_point()`.
- Fix the y-scale to that it shows percentages, its limits go from 0.0 to 1.0, and extra spacing is removed above and below 0.0 and 1.0.

To change the y-axis use `scale_y_continuous` and set the `labels`, `limits`, and `expand` arguments. Check [the ggplot2 documentation]() for how to use `limits` and `expand`. Here is [a StackOverflow question]() that shows how to set `labels` correctly.

-

Hint

To fix the y-axis, you can add (+) the following to your ggplot:

```
scale_y_continuous(labels = scales::percent,
                   limits = 0:1, expand = c(0,0))
```

Task 5: Instructions

Plot the proportion of female laureates by decade split by prize category.

- Add `female_winner` column, where the value is `TRUE` when `sex` is `"Female"`.
- Add the column `decade` showing the decade the prize was awarded (`1953` should become `1950`, for example).

- Group by `decade` and `category` and summarize the proportion of `female_winner` into the `proportion` column.
- Copy and paste your ggplot code from task 4, except plot the `prop_female_winners` data and map the `category` variable to the `color` parameter.

This task can be solved by copying and modifying the code from task 3 and 4.

-

Hint

Here is how you would map `color_variable` to `color`:

```
ggplot(data,
    aes(x_variable, y_variable,
        color = color_variable)) +
    ....
```

Task 6: Instructions

Extract and display the row showing the first woman to win a Nobel Prize.

- Use `filter` to filter away all non-`"Female"` laureates.
- Use `top_n` to pick out the row with the earliest `year`.

`top_n(x, n, wt)` is a useful function that takes a table `x` and picks out the top `n` rows as ordered by the column `wt`. By default `top_n`sort highest-to-lowest so to pick out five best offers in the bargain bin, you would have to use `desc()`:

```
bargain_bin  %>%
    top_n(5, desc(price))
```

-

Hint

Here is how you would use `top_n` to pick out the row with the earliest `year` in a pipeline (`%>%`):

```
top_n(1, desc(year))
```

Task 7: Instructions

Extract and display the names of repeat Nobel Prize winners.

- Use `count` to count the number of wins grouped by `full_name`.
- Filter away all winners that "only" won one time.

-

Hint

`count()` works well together with `group_by()`. For example here is how you would filter away all country names that won less than ten times:

```
nobel %>%
    group_by(birth_country) %>%
    count()  %>%
    filter(n >= 10)
```

Task 8: Instructions

Calculate and plot the age of each winner when they won their Nobel Prize.

- Load the `lubridate` package (you'll find the `year()` function useful).
- `mutate` the `nobel` table to include the column `age` which should be how old people were when they got their price. Assign the resulting table to `nobel_age`.
- Use `ggplot` to plot `age` as a function of `year` as a scatter plot (`geom_point()`) with a smooth trend (`geom_smooth()`).

The `year()` function from `lubridate` takes a date and extracts the year:

```
dates <- as.Date(
    c("1985-04-02",
      "1988-07-25"))
year(dates)
## [1] 1985 1988
```

-

Hint

You can solve this task by using the columns `year` and `birth_date` like this:

```
nobel %>%
    mutate(age = year - year(birth_date))
```

Task 9: Instructions

Plot how old winners are within the different price categories.

- Use `ggplot` to plot `age` as a function of `year` as a scatter plot (`geom_point()`) with a smooth trend (`geom_smooth()`) *and* facet by `category` using `facet_wrap`.
- Optional: Remove the confidence band in `geom_smooth` by setting `se = FALSE`.

This is the same plot as in task 8, except faceted by `category`.

Removing the confidence band in `geom_smooth` is not strictly necessary, but the bands are not that meaningful and removing them makes the plot more focused.

If you don't remember how `facet_wrap` works then look under *Faceting* on the second page of the `ggplot2` cheat sheet.

-

Hint

Copy and paste your solution from task 8 and then add:

```
geom_smooth(se = FALSE) +
facet_wrap(~category)
```

Task 10: Instructions

Pick out the rows of the oldest and the youngest winner of a Nobel Prize.

- Use `top_n`to pick out and display the row of the oldest winner.
- Use `top_n`to pick out and display the row of the youngest winner.

Remember that you can use `desc` to reverse the sorting:

```
# The most expensive item
bargain_bin  %>% top_n(1, price)

# The cheapest item
bargain_bin  %>% top_n(1, desc(price))
```

-

Hint

You almost got the answer in the instructions, so if you clicked on the hint you must be desperate! :) So here is half of the answer:

```
# The oldest winner of a Nobel Prize as of 2016
```

```
nobel_age %>% top_n(1, age)
```

Task 11: Instructions

- Assign the name of the youngest winner of a Nobel Prize to `youngest_winner`. The first name will suffice.

If you want to know more

The Nobel Prize dataset is rich and there and this project just scratched the surface -- there is much more to explore! After you have completed this project you can download it and continue exploring on your own computer! To do that you will have to install Jupyter notebooks with support for R. Here are instructions for how to install the Jupyter Notebook interface and here are instructions for how to add support for R. Good luck!

-

Hint

If you *really* can't figure this out you can always use Google search:

[*youngest nobel prize laureate*]

Naïve Bees: Image Loading and Processing

Task 1: Instructions

First, we need to import the Python libraries with which we will work.

- Import `numpy` as `np`.
- Import the class `Image` from the library `PIL`.
- Create a `numpy` array called `test_data` that is a random array drawn from the beta distribution. The size should be `(100, 100, 3)` and the `a` and `b` parameters should both be `1`. The distribution is in `np.random.beta`.
- Call `plt.imshow` on the test data. You should see some colorful looking random noise!

Good to know

Welcome to the first project in a series on working with image data. We will be working through a Driven Data Competition to identify Honey Bees and Bumble Bees given an image of these insects! To learn more about the background, you can explore the competition page.

For this project, the documentation for Pillow, matplotlib, and numpy will be helpful resources! For more information about bees, see the BeeSpotter project or the DrivenData competition.

The recommended prerequisites for this Project are Intermediate Python for Data Science and Introduction to Data Visualization with Python.

-

Hint

Here's a little help making the test data and displaying it! Try something like the following:

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

test_data = np.random.beta(1, 1, size=(100, 100, 3))
plt.imshow(test_data)
```

Task 2: Instructions

Load an image to introduce you to PIL.

- Assign the variable `img` to an `Image` loaded with the `open` method. The path is `'datasets/bee_1.jpg'`.

- Assign the variable `img_size` to the `size` property of the `img` that you opened.

-

Hint

`Image.open` is the function that loads an image from a path. Once the image is loaded, `img.size` will tell you the width and the height.

Task 3: Instructions

We'll try some of the transformation methods available in Pillow and look at the results.

- Assign `img_cropped` to the image cropped with the `crop` method to the bounding box `25, 25, 75, 75`.
- Assign `img_rotated` to the image rotated by 45 degrees and expanded by 25 pixels with the `rotate` method.
- Assign `img_flipped` to the image flipped left-to-right with the `transpose` method and the `Image.FLIP_LEFT_RIGHT` transposition.

`crop`, `rotate`, and `transpose` are all methods on our existing `img` variable.

-

Hint

Does your `crop` code look something like this?

```
img_cropped = img.crop([..., ..., ..., ...])
```

Does your `rotate` code look something like this?

```
img_rotated = img.rotate(..., expand=...)
```

Does your `transpose` code look something like this?

```
img_flipped = img.transpose(...)
```

Task 4: Instructions

Now we'll get the image as a NumPy array, and with that data, we'll use matplotlib to display the image.

- Assign `img_data` to the result of calling `np.array` on our `Image` object.

- Assign `img_data_shape` equal to the shape of `img_data`, which should be a NumPy array.
- Plot the NumPy array using matplotlib's `imshow` function.
- Plot each of the color channels, red, green, and blue by accessing them in the last dimension of the NumPy array.

Completing this task shows us that at the end of the day, our images are just arrays of numbers.

-

Hint

Simply calling `np.array()` on our `img` data will get us the NumPy array. we can then call `plt.imshow(img_data)` to display it.

You can access each channel with the indices 0, 1, and 2 for the third dimension. For example: `img_data[:, :, 0]`

Task 5: Instructions

Create a kernel density estimate (KDE) plot for each of the color channels on the same plot.

- Assign `channels` to a list of strings that contains the first letter of each color channel ('r', 'g', and 'b').
- In the loop that uses `enumerate`, again use the `[:, :, ix]` selection to take one channel from our `img_data`.
- Pass that data as to `plot_kde` along the `color` variable.

With the KDE plot, we can understand how the color channels in the image differ. We'll want to create our plots in a for-loop for every color. Once this is done, we can call the `plot_kde` function that is provided. We'll do this inside a function called `plot_rgb` so that we can use it again later.

-

Hint

The for loop can be tricky if you haven't seen `enumerate` before. Does yours look something like this?

```
for ix, color in enumerate(channels):
    plot_kde(img_data[:, :, ix], color)
```

Task 6: Instructions

Now we'll load the honey bee image and inspect its colors with a plot.

- Use PIL's `Image.open` method to assign `honey` to the image at the path `'datasets/bee_12.jpg'`.
- Display the `honey` image.
- For `honey`, create a NumPy array of the data and assign it to `honey_data`.
- Plot the channels in `honey_data` with `plot_rgb`.

-

Hint

You can load the images with `Image.open('PATH')`.

Don't forget to turn the PIL `Image` objects into NumPy arrays with `np.array` before calling our `plot_rgb` function.

Task 7: Instructions

Now we'll load the bumble bee image and inspect its colors with a plot.

- Use PIL's `Image.open` method to assign `bumble` to the image at the path `'datasets/bee_3.jpg'`.
- Display the `bumble` image.
- For `bumble`, create a NumPy array of the data and assign it to `bumble_data`.
- Plot the channels in `bumble_data` with `plot_rgb`.

-

Hint

You can load the images with `Image.open('PATH')`.

Don't forget to turn the PIL `Image` objects into NumPy arrays with `np.array` before calling our `plot_rgb` function.

Task 8: Instructions

Now we'll convert the image to grayscale and explore the results of this transformation.

- Assign the variable `honey_bw` to the result of using the `convert` method on `honey` with the parameter `"L"`.
- Assign `honey_bw_arr` to the result of converting `honey_bw` to a NumPy array.
- Assign `honey_bw_arr_shape` to the shape of `honey_bw_arr`.
- Plot the NumPy array version using `plt.imshow` and plot the kde of the new single-channel NumPy array version with `plot_kde`.

-

Hint

Calling `img.convert("L")` will convert the Image object `img` to grayscale from RGB mode. We can then convert this to an array with `np.array`.

Task 9: Instructions

Now we'll look at saving images so we can store the changes we have made.

- Assign `honey_bw_flip` to the result of transposing the image left-right.
- Call the `save` function on `honey_bw_flip` to save it to the path `"saved_images/bw_flipped.jpg"`.
- Assign `honey_hc_arr` to the result of calling `np.maximum` passing our black and white array `honey_bw_arr` and the cutoff value, `100`.
- Convert `honey_hc_arr` back to an `Image` object using `Image.fromarray`, and call that `honey_bw_hc` and save to the path `"saved_images/bw_hc.jpg"`.

-

Hint

None of the calls we make here are new except for `np.maximum`, `Image.fromarray`, and `Image.save`. All of the other ones you can find elsewhere in this project!

Task 10: Instructions

Fill out the pipeline that we can use to process our images.

- Convert the loaded image, `img`, to grayscale with the `convert` function and call that `bw`.
- Save `bw` to the path `bw_path` with the `.save()` function.
- Chain together the commands `.rotate().crop().resize()` and call that `rcz`. Rotate the image 45 degrees (no need to expand as above). Crop the image with the same box as earlier. Resize the image to `(100, 100)`.

- Save `rcz` to the path `rcz_path` with the `.save()` function.

Excellent work on making it to the end of the Project. You're ready to go, so let's get to work!

-

Hint

The trickiest part is realizing that you can chain together transformations in PIL. Does your `rcz` look like `rcz = bw.rotate(45).crop([25, 25, 75, 75]).resize((100, 100))`?

```
# Used to change filepaths

from pathlib import Path


# We set up matplotlib, pandas, and the display function

%matplotlib inline

import matplotlib.pyplot as plt

from IPython.display import display

import pandas as pd


# import numpy to use in this cell

import numpy as np


# import Image from PIL so we can use it later

from PIL import Image


# generate test_data

test_data = np.random.beta(1, 1, size=(100, 100, 3))


# display the test_data
```

```
plt.imshow(test_data)

# open the image

img = Image.open('datasets/bee_1.jpg')


# Get the image size

img_size = img.size


print("The image size is: {}".format(img_size))


# Just having the image as the last line in the cell will display it in the notebook

Img

# Crop the image to 25, 25, 75, 75

img_cropped = img.crop([25, 25, 75, 75])

display(img_cropped)


# rotate the image by 45 degrees

img_rotated = img.rotate(45, expand=25)

display(img_rotated)


# flip the image left to right

img_flipped = img.transpose(Image.FLIP_LEFT_RIGHT)

display(img_flipped)

# Turn our image object into a NumPy array

img_data = np.array(img)
```

```
# get the shape of the resulting array
img_data_shape = img_data.shape

print("Our NumPy array has the shape: {}".format(img_data_shape))

# plot the data with `imshow`
plt.imshow(img_data)
plt.show()

# plot the red channel
plt.imshow(img_data[:, :, 0], cmap=plt.cm.Reds_r)
plt.show()

# plot the green channel
plt.imshow(img_data[:, :, 1], cmap=plt.cm.Greens_r)
plt.show()

# plot the blue channel
plt.imshow(img_data[:, :, 2], cmap=plt.cm.Blues_r)
plt.show()
def plot_kde(channel, color):
    """ Plots a kernel density estimate for the given data.

        `channel` must be a 2d array
        `color` must be a color string, e.g. 'r', 'g', or 'b'
```

```
    """
    data = channel.flatten()
    return pd.Series(data).plot.density(c=color)

# create the list of channels
channels = ['r', 'g', 'b']

def plot_rgb(image_data):
    # use enumerate to loop over colors and indexes
    for ix, color in enumerate(channels):
        plot_kde(image_data[:, :, ix], color)

    plt.show()

plot_rgb(img_data)
# load bee_12.jpg as honey
honey = Image.open('datasets/bee_12.jpg')

# display the honey bee image
display(honey)

# NumPy array of the honey bee image data
honey_data = np.array(honey)

# plot the rgb densities for the honey bee image
```

```
plot_rgb(honey_data)

# load bee_3.jpg as bumble

bumble = Image.open('datasets/bee_3.jpg')


# display the bumble bee image

display(bumble)


# NumPy array of the bumble bee image data

bumble_data = np.array(bumble)


# plot the rgb densities for the bumble bee image

plot_rgb(bumble_data)

# convert to grayscale

honey_bw = honey.convert("L")

display(honey_bw)


# convert the image to a NumPy array

honey_bw_arr = np.array(honey_bw)


# get the shape of the resulting array

honey_bw_arr_shape = honey_bw_arr.shape

print("Our NumPy array has the shape: {}".format(honey_bw_arr_shape))


# plot the array using matplotlib

plt.imshow(honey_bw_arr, cmap=plt.cm.gray)
```

```
plt.show()

# plot the kde of the new black and white array
plot_kde(honey_bw_arr, 'k')
# flip the image left-right with transpose
honey_bw_flip = honey_bw.transpose(Image.FLIP_LEFT_RIGHT)

# show the flipped image
display(honey_bw_flip)

# save the flipped image
honey_bw_flip.save("saved_images/bw_flipped.jpg")

# create higher contrast by reducing range
honey_hc_arr = np.maximum(honey_bw_arr, 100)

# show the higher contrast version
plt.imshow(honey_hc_arr, cmap=plt.cm.gray)

# convert the NumPy array of high contrast to an Image
honey_bw_hc = Image.fromarray(honey_hc_arr)

# save the high contrast version
honey_bw_hc.save("saved_images/bw_hc.jpg")
image_paths = ['datasets/bee_1.jpg', 'datasets/bee_12.jpg', 'datasets/bee_2.jpg',
'datasets/bee_3.jpg']
```

```
def process_image(path):

    img = Image.open(path)


    # create paths to save files to

    bw_path = "saved_images/bw_{}.jpg".format(path.stem)

    rcz_path = "saved_images/rcz_{}.jpg".format(path.stem)


    print("Creating grayscale version of {} and saving to {}.".format(path, bw_path))

    bw = img.convert("L")

    bw.save(bw_path)


    print("Creating rotated, cropped, and zoomed version of {} and saving to {}.".format(path,
rcz_path))

    rcz = bw.rotate(45).crop([25, 25, 75, 75]).resize((100, 100))

    rcz.save(rcz_path)


# for loop over image paths

for img_path in image_paths:

    process_image(Path(img_path))
```

Generating Keywords for Google Ads

Task 1: Instructions

Create a list of words to pair with products.

- Create a list of six to ten strings named `words` that contain words you think would work well with the products in the brief. If you're stuck, here are six words that would work: buy, price, discount, promotion, promo, and shop.
- Print `words` to inspect your newly-created list.

Good to know

Note: the hint contains all acceptable words for this task.

Welcome to the Project! This Project requires that you know your way around Python and pandas. We recommend the following courses as prerequisites:

- Intro to Python for Data Science
- Intermediate Python for Data Science

Remember the words have to signify purchase intent and should be focused on a price-sensitive audience.

Helpful links:

- Search engine marketing campaign for DataCamp: A tutorial, showing a full case study, where you go through the process of creating an entire campaign for DataCamp. You will be going through part of the process here.
- Python Data Structures DataCamp Tutorial
- advertools: Once you are comfortable with the basic concepts, you can check the advertools package, which has several functions and tools for online marketing productivity and analysis.
- Search Engine Marketing cheat sheet

-

Hint

Think about what words people might use while they are planning to buy sofas at good prices. Here are several that we thought of:

```
words = ['buy', 'price', 'discount', 'promotion', 'promo', 'shop',
         'buying', 'prices', 'pricing', 'shopping', 'discounts',
         'promos', 'ecommerce', 'e commerce', 'buy online',
         'shop online', 'cheap', 'best price', 'lowest price',
         'cheapest', 'best value', 'offer', 'offers', 'promotions',
         'purchase', 'sale', 'bargain', 'affordable',
```

```
        'cheap', 'low cost', 'low price', 'budget', 'inexpensive',
'economical']
```

Task 2: Instructions

Combine words and products.

- Create an empty list named `keywords_list`.
- Loop through all `products`.
- Loop through all `words`.
- Append `product`, and `word` and `product` joined by a space.

An example of an appended list would be: `['sofas', 'sofas buy']`. Note that you can concatenate two strings with the + operator in Python.

To inspect `keyword_list`, a module named `pprint` is used that "pretty-prints" Python data structures like lists (or lists of lists!) and dictionaries. After importing the module, `pprint()` is used like Python's regular `print()` statement. Here is the `pprint` documentation, which has plenty of examples. Here is a specific example with lists in an answer on Stack Overflow.

Helpful links:

- List exercises in Intro to Python for Data Science
- Loop exercises in Intermediate Python for Data Science
-

Hint

The following code puts the product first, and then the product together with the word, separated by a space:

```
for product in products:
    for word in words:
        keywords_list.append([product, product + ' ' + word])
```

The last line of code in this task should be the same as the second last line, except with product and word switched around in the `... + ' ' + ...` part of the list.

Task 3: Instructions

Convert the list into a DataFrame.

- Load the `pandas` library aliased as `pd`.
- Create a DataFrame named `keywords_df` from `keywords_list` using the pandas `from_records()` method. Don't specify column names yet.
- Inspect the contents of `keywords_df` using the `head()` method.

Helpful links:

- pandas documentation for the `from_records()` method
-

Hint

It is common to import the pandas library like this:

```
import pandas as pd
```

If you were to convert a list named `random_list` into a DataFrame, you could do so using `pd.DataFrame.from_records` like so:

```
random_df = pd.DataFrame.from_records(random_list)
```

Task 4: Instructions

Give `keyword_df` more descriptive column names.

- Rename column `0` to `Ad Group` and column `1` to `Keyword`.

Helpful links:

- pandas `rename` documentation
- Stack Overflow answer for "Renaming columns in pandas"
-

Hint

If you wanted to rename the columns of DataFrame named `df`, you could do so as follows using the `rename()` method:

```
df = df.rename(columns={'oldName1': 'newName1', 'oldName2': 'newName2'})
```

Task 5: Instructions

Add a column to include the campaign name.

- Add a new column called `Campaign` with the value 'SEM_Sofas' in every row.

Helpful links:

- Stack Overflow answer for "Add column to dataframe with default value"
-

Hint

If you wanted to add a column named `column_name` with the value 'abc' in every row of that column, you could do so as follows:

```
df['column_name']='abc'
```

Task 6: Instructions

Add a column to include the match criterion.

- Add a new column called `Criterion Type` with the value 'Exact' in every row.

Helpful links:

- Stack Overflow answer for "Add column to dataframe with default value"
- AdWords Keyword match type documentation
-

Hint

If you wanted to add a column named `column_name` with the value 'abc' in every row of that column, you could do so as follows:

```
df['column_name']='abc'
```

Task 7: Instructions

Create phrase match keywords.

- Change the `Criterion Type` column values of the newly copied DataFrame `keyword_phrase` to 'Phrase'.
- Append the two DataFrames vertically so you can have one final DataFrame containing both. Name the new DataFrame `keywords_df_final`.

Appending DataFrames exercises are covered in the Merging DataFrames with pandas course. The appending task may be a bit outside of your comfort zone since it is not covered in the recommended prerequisites, so don't hesitate to check the hint if you're stuck!

-

Hint

If you wanted to add a column named `column_name` with the value 'abc' in every row of that column, you could do so as follows:

```
df['column_name']='abc'
```

Since appending DataFrames is not covered in the recommended prerequisites, here is how to append the two DataFrames together:

```
keywords_df_final = keywords_df.append(keywords_phrase)
```

Task 8: Instructions

Save the `DataFrame` to a CSV file.

- Save `keywords_df_final` to a CSV file named 'keywords.csv' using the pandas `to_csv()` method. Exclude the DataFrame index in the saved file by specifying `index=False`.

Saving DataFrames to CSV files isn't explicitly covered in Intermediate Python for Data Science, but it is the opposite of these "CSV to DataFrame" exercises. Don't hesitate to check out the hint if you're stuck!

If you'd like to learn how to group by and count to generate the campaign summary, check out the Manipulating DataFrames with pandas course. The pandas split-apply-combine documentation is also excellent!

If you're curious about pasting the data into the AdWords or BingAds editor, check out the pandas `to_clipboard` documentation.

Helpful links:

- `pd.to_csv()` documentation
- DataCamp tutorial: How To Write a Pandas DataFrame to a File
- Stack Overflow answer for "How to avoid Python/Pandas creating an index in a saved csv?"

-

Hint

If you have a DataFrame named `df` and you want to save it to a file named 'file_name.csv' and exclude the DataFrame index, you can do so as follows:

```
df.to_csv('file_name.csv', index=False)

# List of words to pair with products
words = ['buy', 'price', 'discount', 'promotion', 'promo', 'shop']

# Print list of words
print(words)
products = ['sofas', 'convertible sofas', 'love seats', 'recliners', 'sofa
beds']

# Create an empty list
keywords_list = []

# Loop through products
for product in products:
    # Loop through words
    for word in words:
        # Append combinations
        keywords_list.append([product, product + ' ' + word])
        keywords_list.append([product, word + ' ' + product])

# Inspect keyword list
from pprint import pprint
pprint(keywords_list)

# Load library
import pandas as pd

# Create a DataFrame from list
keywords_df = pd.DataFrame.from_records(keywords_list)

# Print the keywords DataFrame to explore it
print(keywords_df)

# Rename the columns of the DataFrame
keywords_df = keywords_df.rename(columns={0: 'Ad Group', 1: 'Keyword'})

# Add a campaign column
keywords_df['Campaign'] = 'SEM_Sofas'

# Add a criterion type column
keywords_df['Criterion Type'] = 'Exact'

# Make a copy of the keywords DataFrame
keywords_phrase = keywords_df.copy()

# Change criterion type match to phrase
keywords_phrase['Criterion Type'] = 'Phrase'

# Append the DataFrames
keywords_df_final = keywords_df.append(keywords_phrase)

# Save the final keywords to a CSV file
keywords_df_final.to_csv('keywords.csv', index=False)

# View a summary of our campaign work
```

```
summary = keywords_df_final.groupby(['Ad Group', 'Criterion Type'])
['Keyword'].count()
print(summary)
```

Naïve Bees: Predict Species from Images

Task 1: Instructions

First, we need to import the Python libraries with which we will work.

- Import the class `Image` from the library `PIL`.
- Import the function `train_test_split` from the `model_selection` module of `sklearn`.
- Import the function `SVC` from the `svm` module of `sklearn`. This is the model we'll use.
- Import the function `accuracy_score` from the `metrics` module of `sklearn`. This is the metric we'll use.

Good to know

Welcome to the second project in a series on working with image data. We will be working through a Driven Data Competition to identify Honey Bees and Bumble Bees given an image of these insects! To learn more about the background, you can explore the competition page.

For this project, the documentation for scikit-learn, scikit-image, and numpy will be helpful resources! For more information about bees, see the BeeSpotter project or the DrivenData competition.

The recommended prerequisites for this Project are Intermediate Python for Data Science, Introduction to Data Visualization with Python, Supervised Learning with scikit-learn, and Naïve Bees: Image Loading and Processing.

-

Hint

Here's a little help getting the import structure right! Try something like the following:

```
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, auc, accuracy_score
```

Task 2: Instructions

Load the dataframe of labels and image names. Using that, display the sixth image of a Bombus (bumble bee).

- Display the first five rows of the labels dataframe using `.head()`.

- Subset the dataframe to just Bombus (i.e. where `genus == 1.0`) and assign the sixth item of the index of the subsetted dataframe to the variable `bombus_row`.
- Call the function `get_image` on `bombus_row` to get the image of the Bombus and display it using `plt.imshow`.

-

Hint

The equivalent operations for Apis are:

```
apis_row = labels[labels.genus == 0.0].index[5]
plt.imshow(get_image(apis_row))
```

To convert these two rows for Bombus, change `genus == 0.0` to `genus == 1.0` and replace `apis_row` with `bombus_row`.

Task 3: Instructions

Now we'll use the `rgb2grey` function to convert our image to greyscale.

- Load our bombus image using the `get_image` function and `bombus_row` from the previous cell and assign it to `bombus`.
- Print the shape of the bombus image to see that it has three color channels.
- Convert the bombus image to greyscale using `rgb2grey`.
- Print the shape of the greyscale image to see that it only has one channel.

-

Hint

Did you load your bombus image with `get_image(bomus_row)`?

Does you convert the bombus image to greyscale with `rgb2grey(bombus)`?

Task 4: Instructions

Using our greyscale image, we'll calculate the HOG feature vector and image using `hog` from scikit-image.

- Call `hog` on the greyscale bombus image.
- Plot the `hog_image` using matplotlib's `imshow` function.

-

Hint

Remember `hog` takes in a greyscale image. We want to use the greyscale bombus image (`grey_bombus`) from above.

To show the image, try something like `plt.imshow(hog_image, cmap=mpl.cm.gray)`.

Task 5: Instructions

Write and call a function to create features for and flatten the bombus image.

Within the `create_features` function:

- Using the method `flatten()`, flatten the original image into a one-dimensional array and assign it to `color_features`.
- Using the function `hstack()`, combine the `color_features` and the `hog_features` arrays and assign it to `flat_features`.

Then,

- Call the `create_features` on our `bombus` image and assign it to `bombus_features`.
- Print the shape of `bombus_features`.

There should be 31,296 features in the one-dimensional array for the bombus image.

Helpful links:

- `numpy.ndarray.flatten()` documentation
- `numpy.hstack()` documentation

-

Hint

Since `img` is already a numpy array, we can use `flatten` to collapse the three-dimensional array into a one-dimensional one. Does your `color_features` calculation look something like `img.flatten()`?

To combine two arrays sequentially, we can use `np.hstack`. Does your `flat_features` calculation look something like `np.hstack([color_features, hog_features])`?

Task 6: Instructions

Write a function to generate the features we care about for each image in our dataframe and stack the resulting arrays into a feature matrix.

- Load each image in the dataframe using the `get_image` function from earlier.
- Create the features for each image using the `create_features` function from above.

Then, use this function to create a feature matrix for our dataframe.

- Call `create_feature_matrix` on our labels dataframe and assign it to `feature_matrix`.

Note: it may take a few seconds for the cell to finish running as it's calculating the features for all 500 images in our dataset.

-

Hint

This function combines many of the functions we wrote earlier in the notebook so be sure to reference those.

Keep in mind:

- `get_image` takes in the value of the dataframe index, `img_id`
- `create_features` takes in the color image, `img`
- our dataframe is called `labels`

Task 7: Instructions

Now we'll prepare our features for modeling by first scaling them and then reducing the number of features using PCA.

- Print the shape of our `feature_matrix`.
- Use the `fit_transform()` method of the StandardScaler object to scale our `feature_matrix` and assign it to `bees_stand`.
- Use the `fit_transform()` method of the PCA object to condense `bees_stand`, our standardized feature matrix, and assign it to `bees_pca`.
- Look at the new shape of our final `bees_pca` matrix.

Note: it may take about 10 seconds for the cell to finish running as PCA is computationally intensive.

-

Hint

We want to scale `feature_matrix` and then perform PCA on `bees_stand` (our standardized feature matrix).

Remember that the syntax for both StandardScaler and PCA, our two methods, is:

```
x = Method()
transformed_matrix = x.fit_transform(matrix)
```

Task 8: Instructions

Split the data into train and test sets for modeling and look at the distribution of labels in the train set.

- Call `train_test_split` where X is the `bees_pca` matrix (the condensed features matrix that was the result of PCA), and y is `labels.genus.values`, a series containing the label for each image (from our dataframe).
- Convert `y_train` to a pandas series using `pd.Series` and then call `value_counts` to see the distribution of Apis (`0.0`) and Bombus (`1.0`) labels in our training data. In this example, we have a perfectly balanced dataset so we will see an equal number of honey and bumble bees.

-

Hint

`train_test_split` has the syntax: `train_test_split(X, y, test_size, random_state)`.

Did you know that you can chain pandas operations together, such as `pd.Series(y_train).value_counts()`?

Task 9: Instructions

Train a support vector classifier (SVC).

- Define our model using the `SVC` object with a linear kernel, set `probability=True`, and `random_state=42`.
- Train the model on `X_train` and `y_train` using `.fit()`.

In this task, we're calculating predictions for each image (i.e. whether an image is a honey bee or a bumble bee). However, we set `probability=True` within our

model object `SVC` in order to later be able to look at the probability that each image is a honey or bumble bee. We'll use these probabilities in the next step.

We set `random_state=42` to ensures that the data is shuffled in a consistent way, meaning our model results won't change if re-run the cell.

For an excellent visualization of how SVMs work, check out [this video](#).

-

Hint

Did you define your model with `SVC(kernel='linear', probability=True, random_state=42)`?

Does your model fitting look something like `svm.fit(X_train, y_train)`?

Task 10: Instructions

Using our trained model, predict on the test set and calculate accuracy.

- Predict the labels for `X_test` using `.predict()`.
- Calculate the accuracy by calling `accuracy_score` on the predicted values from the previous step and the true values (`y_test`).

-

Hint

Did you generate your predictions with `svm.predict(X_test)`?

Remember that `accuracy_score` takes in the true values of y (`y_test`) and the predicted values of y (`y_pred`).

Task 11: Instructions

Finally, we'll get the predicted probabilities for each image in the test set, plot the ROC curve, and calculate the AUC.

- Use the `predict_proba` method from our SVM model to get the probabilities for `X_test` and assign them to `probabilities`.
- Assign `y_proba` to the probabilities for label 1.0 (the second column of `probabilities`).
- Calculate the AUC from `false_positive_rate` and `true_positive_rate` and assign it to `roc_auc`.

- Plot the false positive rate on the x axis and the true positive rate on the y axis to show the ROC curve.

We now have a fully trained computer vision model that can be used to identify honey bees and bumble bees in images. The next step in the data science process is to explore improving the model by using more data, adding new features, and trying different methods!

-

Hint

The trickiest part here is selecting only the probabilities that each image has a true label of `1.0`. Remember that probabilities is an array with two columns, where the first column reflects the probability of having a true label of `0.0` and the second columns reflects the probability of having a true label of `1.0`. To select the second column, we can do `probabilities[:, 1]` and assign this to `y_proba`.

```
# used to change filepaths

import os



import matplotlib as mpl

import matplotlib.pyplot as plt

from IPython.display import display

%matplotlib inline



import pandas as pd

import numpy as np



# import Image from PIL

from PIL import Image



from skimage.feature import hog

from skimage.color import rgb2grey
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# import train_test_split from sklearn's model selection module
from sklearn.model_selection import train_test_split

# import SVC from sklearn's svm module
from sklearn.svm import SVC

# import accuracy_score from sklearn's metrics module
from sklearn.metrics import roc_curve, auc, accuracy_score
# load the labels using pandas
labels = pd.read_csv("datasets/labels.csv", index_col=0)

# show the first five rows of the dataframe using head
display(labels.head())

def get_image(row_id, root="datasets/"):
    """
    Converts an image number into the file path where the image is located,
    opens the image, and returns the image as a numpy array.
    """
    filename = "{}.jpg".format(row_id)
    file_path = os.path.join(root, filename)
```

```
    img = Image.open(file_path)

    return np.array(img)


# subset the dataframe to just Apis (genus is 0.0) get the value of the sixth item in the index

apis_row = labels[labels.genus == 0.0].index[5]


# show the corresponding image of an Apis

plt.imshow(get_image(apis_row))

plt.show()


# subset the dataframe to just Bombus (genus is 1.0) get the value of the sixth item in the
index

bombus_row = labels[labels.genus == 1.0].index[5]


# show the corresponding image of a Bombus

plt.imshow(get_image(bombus_row))

plt.show()

# load a bombus image using our get_image function and bombus_row from the previous cell

bombus = get_image(bombus_row)


# print the shape of the bombus image

print('Color bombus image has shape: ', bombus.shape)


# convert the bombus image to greyscale

grey_bombus = rgb2grey(bombus)
```

```
# show the greyscale image
plt.imshow(grey_bombus, cmap=mpl.cm.gray)

# confirm greyscale bombus image only has one channel
print('Greyscale bombus image has shape: ', grey_bombus.shape)
# run HOG using our greyscale bombus image
hog_features, hog_image = hog(grey_bombus,
                              visualize=True,
                              block_norm='L2-Hys',
                              pixels_per_cell=(16, 16))

# show our hog_image with a grey colormap
plt.imshow(hog_image, cmap=mpl.cm.gray)

def create_features(img):
    # flatten three channel color image
    color_features = img.flatten()
    # convert image to greyscale
    grey_image = rgb2grey(img)
    # get HOG features from greyscale image
    hog_features = hog(grey_image, block_norm='L2-Hys', pixels_per_cell=(16, 16))
    # combine color and hog features into a single array
    flat_features = np.hstack([color_features, hog_features])
    return flat_features
```

```
bombus_features = create_features(bombus)


# print shape of bombus_features
bombus_features.shape
def create_feature_matrix(label_dataframe):
    features_list = []


    for img_id in label_dataframe.index:
        # load image
        img = get_image(img_id)
        # get features
        image_features = create_features(img)
        features_list.append(image_features)


    # convert list of arrays into a matrix
    feature_matrix = np.array(features_list)
    return feature_matrix


# run create_feature_matrix on our dataframe of images
feature_matrix = create_feature_matrix(labels)
# get shape of feature matrix
print('Feature matrix shape is: ', feature_matrix.shape)


# define standard scaler
ss = StandardScaler()
```

```
# run this on our feature matrix
bees_stand = ss.fit_transform(feature_matrix)

pca = PCA(n_components=500)
# use fit_transform to run PCA on our standardized matrix
bees_pca = pca.fit_transform(bees_stand)
# look at new shape
print('PCA matrix shape is: ', bees_pca.shape)

X_train, X_test, y_train, y_test = train_test_split(bees_pca,
                                                    labels.genus.values,
                                                    test_size=.3,
                                                    random_state=1234123)

# look at the distrubution of labels in the train set
pd.Series(y_train).value_counts()
# define support vector classifier
svm = SVC(kernel='linear', probability=True, random_state=42)

# fit model
svm.fit(X_train, y_train)
# generate predictions
y_pred = svm.predict(X_test)

# calculate accuracy
```

```
accuracy = accuracy_score(y_test, y_pred)
print('Model accuracy is: ', accuracy)
# predict probabilities for X_test using predict_proba
probabilities = svm.predict_proba(X_test)

# select the probabilities for label 1.0
y_proba = probabilities[:, 1]

# calculate false positive rate and true positive rate at different thresholds
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_proba, pos_label=1)

# calculate AUC
roc_auc = auc(false_positive_rate, true_positive_rate)

plt.title('Receiver Operating Characteristic')
# plot the false positive rate on the x axis and the true positive rate on the y axis
roc_plot = plt.plot(false_positive_rate,
                    true_positive_rate,
                    label='AUC = {:0.2f}'.format(roc_auc))

plt.legend(loc=0)
plt.plot([0,1], [0,1], ls='--')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate');
```

Partnering to Protect You from Peril

Task 1: Instructions

Import the network ties and the characteristics of each health department in the network.

- Import the network edges data (`naccho2016clean.csv`) from the `datasets` folder using `read_csv()`.
- Import the node attributes data (`naccho2016att.csv`) from the `datasets` folder with `read_csv()`.
- Use `graph_from_data_frame()` to merge the edge data and attribute data.
- Show the network object.

Good to know

The project will use `igraphm` `readr` and `dplyr` to import and examine a network made up of an edgelist and an attribute file. Most commands are covered in the DataCamp courses: **Network Analysis in R** and **Network Analysis in the Tidyverse**.

Commands will be used to import data and for subsetting vertices, measuring network size and density, measuring vertex importance with different centralization indices, examining the connections of two large urban health departments, and visualization including node color, size, and layout. You should have an intermediate knowledge of R and the package, `igraph`.

Helpful links:

- Learn about the survey by the National Association of County and City Health Officials (NACCHO) here.
-

Hint

Load edges and attributes with `read_csv()`. Be sure to use quote marks around the dataset folder and file names.

Merge edges and attributes with `graph_from_data_frame()`, like this:

```
graph_from_data_frame(d = name of edges file, vertices = name of attributes
file, directed = FALSE)
```

Task 2: Instructions

Check the network object for loops and multiple ties between health departments. Remove any loops or multiple ties that you find:

- Use `is_simple()` to determine if the network contains loops or multiple ties.
- If `is_simple()` returns `FALSE`, use `simplify()` to remove loops and multiple ties.
- Confirm that the network has no loops or multiple ties using `is_simple()`.

Helpful links:

- The `igraph` documentation for simplify
-

Hint

Use `is_simple()` to check for loops and multiples.

`simplify()` is also a function in `purrr`. If that package was loaded after `igraph`, you would have to use `igraph::simplify()` to avoid conflicts.

Task 3: Instructions

Get to know your network via a few standard network descriptive statistics.

- Use `vcount()` to count the number of vertices in the health department network.
- Use `ecount()` to count the number edges in the health department network.
- Compute the network density with `edge_density()`.

Helpful links:

- The igraph documentation for ecount.
- The igraph documentation for vcount.
- The igraph documentation for edge_density.
-

Hint

Use `ecount()` for edges and `vcount()` for vertices.

Task 4: Instructions

Identify the key central health departments that share information and coordinate national emergency preparedness and response.

- Use the `degree()` to add a `health.dep.degree` variable to the `health.dep.nodes` attributes data frame.
- Use `arrange()` to list the health departments with the highest degree.
- Use the `betweenness()` to add a `health.dep.between` variable to the `health.dep.nodes` attributes data frame.
- Use `arrange()` to list the health departments with the highest betweenness.

Helpful links:

- Learn more about degree and betweenness centrality in networks from [this DataCamp article](#).
- Find the arguments for using `arrange()` in the [documentation](#).
-

Hint

Degree is a node attribute, so the degree of each node is added to the node attributes file `health.dep.nodes` like this,

```
health.dep.nodes$health.dep.degree <- degree(health.dep.net)
```

To list the health departments with the highest centrality using `arrange()`, put the betweenness measures in descending order by using `-health.dep.degree` as the second argument like this,

```
arrange(health.dep.nodes, -health.dep.between)
```

Task 5: Instructions

Examine the network of local health departments across Texas and Louisiana:

- Use `induced_subgraph()` from `igraph` to create `region.net` that includes the nodes and links for Texas (TX) and Louisiana (LA) health departments.
- Find the number of vertices (i.e., network size) using `vcount()`.
- Use `edge_density()` to find the density of `region.net`.
- Use `ggraph()` to plot the network with nodes colored by state and the `theme_graph` theme.

Note that the network is disconnected with no ties between Texas and Louisiana health departments. This is a potential area for improvement for the states.

Helpful links:

- Read about `induced_subgraph()` in the `igraph` documentation.
- Find information on `theme_graph` here.
-

Hint

Subset the network with `induced_subgraph()` to pul out the states LA and TX like so:

```
region.net <- induced_subgraph(graph = health.dep.net, vids =
which(V(health.dep.net)$state %in% c('LA', 'TX')))
```

In the call to `ggraph()`, `with_kk` is assigned to the `layout=` argument.

Use `theme_graph()` layer to change the look of the graph.

Task 6: Instructions

Find the health departments in each state that are most well connected and are bridging others:

- Use `degree()` to find the degree of nodes in `region.net`.
- Use `sort()` to find the health departments with the highest degree in each state.
- Use `betweenness()` to find the betweenness of nodes `region.net`.
- Use `sort()` to find the health departments with the highest betweenness in each state.

Helpful links:

- Learn more about degree and betweenness centrality in networks from this DataCamp article.
-

Hint

The top `LA` health departments by **degree** are found with this code:

```
head(sort(region.net$degree[V(region.net)$state == "LA"], decreasing =
TRUE))
```

Use this to help fill in the others.

Task 7: Instructions

Examine the central nodes using network plots:

- Add `degree` to the node attributes.
- Plot the network with nodes sized by degree, color by state, use Kamada Kawai layout, theme_graph, and add node labels using `label = name` in the `geom_node_text()` layer.
- Add `betweenness` to the node attributes.
- Plot the network with nodes sized by betweenness, color by state, use Kamada Kawai layout, theme_graph, and add node labels using `label = name` in the `geom_node_text()` layer.

Helpful links:

- Learn more about degree and betweenness centrality in networks from this DataCamp article.
- Explore the `geom_node_point` layer documentation.
- Review the `geom_node_text` layer documentation.

-

Hint

Here is the code for the first graph:

```
region.plot.degree <- ggraph(data = region.net, layout = "with_kk") +
    geom_edge_link() +
    geom_node_point(aes(colour = state, size = degree)) +
    geom_node_text(aes(label = name, size = 1), nudge_y = .25) +
    theme_graph()
```

Try to figure out the second one on your own!

Task 8: Instructions

Examine the network of California health departments:

- Use `induced_subgraph()` to subset the network so it includes only California health departments.
- Find the number of vertices (i.e., network size) using `vcount()`.
- Use `edge_density()` to find the density.
- Find and sort degree centrality for each health department.
- Find and sort betweenness centrality for each health department.

Helpful links:

- Learn more about degree and betweenness centrality in networks from this DataCamp article.

- Read about subsetting in `igraph` here.
-

Hint

Use `induced_subgraph()` and the `vids=` argument to get the California health departments from the health department network, like so:

```
cali.net <- induced_subgraph(graph = health.dep.net, vids =
which(V(health.dep.net)$state %in% "CA"))
```

Use `head(sort(cali.net$degree, decreasing = TRUE))` to get the top six California health department with the highest degrees.

Task 9: Instructions

Visualize `cali.net` with nodes sized by degree centrality and node color representing `rurality`, `population`, and `fte`.

- Fill in the `colour` parameter with the rurality attribute and the `size` parameter with degree to visualize rurality in cali.net.
- Fill in the `colour` parameter with the population attribute and the `size` parameter with degree to visualize population in cali.net.
- Fill in the `colour` parameter with the fte attribute and the `size` parameter with degree to visualize fte in cali.net.

Helpful links:

- Learn more about degree and betweenness centrality in networks from this DataCamp article.
- Explore the `geom_node_point` layer documentation.
- Review the `geom_node_text` layer documentation.
-

Hint

Here is the rurality graph.

```
cali.net.rurality <- ggraph(graph = cali.net, layout = "with_kk") +
  geom_edge_link() +
  geom_node_point(aes(colour = rurality, size = degree(cali.net))) +
  geom_node_text(aes(label = name, size = 3), nudge_y = .2) +
  theme_graph()
```

See if you can fill in the other two on your own.

Task 10: Instructions

Visualize `cali.net` with node color representing different network characteristics and node size representing `betweenness`:

- Assign `betweenness` as a node attribute called `between` for the `cali.net` networks.
- Visualize the networks with nodes sized by `betweenness` and colored by `rurality`, `population`, and `fte`.

Helpful links:

- Learn more about degree and betweenness centrality in networks from this DataCamp article.
- Explore the `geom_node_point` layer documentation.
- Review the `geom_node_text` layer documentation.
-

Hint

Here is the rurality graph.

```
cali.net.rural.bet <- ggraph(graph = cali.net, layout = "with_kk") +
  geom_edge_link() +
  geom_node_point(aes(colour = rurality, size = between)) +
  geom_node_text(aes(label = name, size = 3), nudge_y = .2) +
  theme_graph()
```

See if you can fill in the other two on your own.

A Visual History of Nobel Prize Winners

Task 1: Instructions

Load the required libraries and the Nobel Prize dataset.

- Import the `pandas` library as `pd`.
- Import the `seaborn` library as `sns`.
- Import the `numpy` library as `np`.
- Use `pd.read_csv` to read in `datasets/nobel.csv` and save it into `nobel`.
- Show at least the first six entries of `nobel` using the `head()` method, setting `n=6` or greater.

Good to know

Note: a test in this task relies on the expected last output of this cell. In this case, the first six entries of `nobel` being displayed. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

This project assumes you are familiar with the `pandas` and `seaborn` libraries and before taking on this project, we recommend that you have completed the courses Data Manipulation with pandas and Intermediate Data Visualization with Seaborn.

Two cheat sheets that will be useful throughout this project: DataCamp's Seaborn cheat sheet and Data Wrangling with pandas cheat sheet. We recommend that you keep them open in separate tabs to make it easy to refer to them.

-

Hint

If you've loaded in `pandas` like this:

```
import pandas as pd
```

you can read in `path_to/my_data.csv` like this:

```
my_data = pd.read_csv("path_to/my_data.csv")
```

Task 2: Instructions

Count up the Nobel Prizes. Also, split by `sex` and `birth_country`.

- Count the number of rows/prizes using the `len()` function. Use the `display()` function to display the result.
- Count and `display` the number of prizes for each `sex` using the `value_counts()` method.
- Count the number of prizes for each `birth_country` using `value_counts()` and show the top 10 using `head()`. **Do not use `display()`.**

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

By default, a Jupyter Notebook (which is where you are working right now) will only show the final output in a cell. If you want to show intermediate results, you will have to use the `display()` function. See here for an example of how to use `value_counts()`.

Why `display()` over `print()`? Try them both out for yourself. You'll find that the output of `display()` is prettier. :)

-

Hint

Here is how to solve the most complicated part of this task:

```
nobel['birth_country'].value_counts().head(10)
```

See if you can figure out the two easier parts yourself!

Task 3: Instructions

Create a DataFrame with two columns: decade and proportion of USA-born Nobel Prize winners that decade.

- Add a `usa_born_winner` column to `nobel`, where the value is `True` when `birth_country` is `"United States of America"`.
- Add a `decade` column to `nobel` for the decade each prize was awarded. Here, `np.floor()` will come in handy. Ensure the decade column is of type `int64`.
- Use `groupby` to group by `decade`, setting `as_index=False`. Then isolate the `usa_born_winner` column and take the `mean()`. Assign the resulting DataFrame to `prop_usa_winners`.
- Display `prop_usa_winners`.

For the `decade` column, 1953 should become 1950, for example. Calculating this column is a bit tricky, but try to see if you can solve it using the `np.floor` function. If not, check the hint!

By setting `as_index=False`, you make sure that both the grouping variable and the calculated mean are included in the resulting DataFrame.

-

Hint

You can add the column `usa_born_winner` like this:

```
nobel['usa_born_winner'] = nobel['birth_country'] == 'United States of America'
```

To go from year to decade, you can fiddle around with multiplication, division, the `np.floor` function, and the pandas `astype()` method. For example, here is one way you could calculate the decade of `year`:

```
year = pd.Series([1843, 1877, 1923])
decade = (np.floor(year / 10) * 10).astype(int)
# decade is now 1840, 1870, 1920
```

The grouping code should look something like this:

```
df.groupby('COLUMN 1', as_index=False)['COLUMN 2'].mean()
```

Task 4: Instructions

Plot the proportion of USA born winners per decade.

- Use seaborn to plot `prop_usa_winners` with `decade` on the x-axis and `usa_born_winner` on the y-axis as an `sns.lineplot`. Assign the plot to `ax`.
- Fix the y-scale so that it shows percentages using `PercentFormatter`.

See here for a Stack Overflow answer on how `PercentFormatter` works and here for the documentation of `lineplot`.

-

Hint

To percent format the y-axis of a plot assigned to `ax` you can do the following:

```
from matplotlib.ticker import PercentFormatter
ax.yaxis.set_major_formatter(PercentFormatter(1.0))
```

Task 5: Instructions

Plot the proportion of female laureates by decade split by prize category.

- Add the `female_winner` column to `nobel`, where the value is `True` when `sex` is `"Female"`.
- Use `groupby` to group by both `decade` and `category`, setting `as_index=False`. Then isolate the `female_winner` column and take the `mean()`. Assign the resulting DataFrame to `prop_female_winners`.
- Copy and paste your `seaborn` plot from task 4 (including axis formatting code), but plot `prop_female_winners` and map the `category` variable to the `hue` parameter.

This task can be solved by copying and modifying the code from task 3 and 4.

-

Hint

Here is how you would map `category` to `hue`:

```
ax = sns.lineplot(x='decade', y='female_winner', hue='category',
data=prop_female_winners)
```

Task 6: Instructions

Extract and display the row showing the first woman to win a Nobel Prize.

- Select only the rows of `'Female'` winners in `nobel`.
- Using the `nsmallest()` method with its `n` and `columns` parameters, pick out the first woman to get a Nobel Prize.

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

See here for the documentation of `nsmallest()`.

-

Hint

Here is how you would use `nsmallest` to pick out the row with the earliest `year` of *all* the Nobel Prizes:

```
nobel.nsmallest(1, 'year')
```

Task 7: Instructions

Extract and display the rows of repeat Nobel Prize winners.

- Use `groupby` to group `nobel` by `'full_name'`.
- Use the `filter` method to keep only those rows in `nobel` with winners with 2 or more prizes.

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

See here for how to use the `filter` method.

-

Hint

Here is an example of how to use `groupby` together with `filter`. This would keep only those rows with birth countries that have had 50 or more winners:

```
nobel.groupby('birth_country').filter(lambda group: len(group) >= 50)
```

Task 8: Instructions

Calculate and plot the age of each winner when they won their Nobel Prize.

- Convert the `nobel['birth_date']` column to datetime using `pd.to_datetime`.
- Add a new column `nobel['age']` that contains the age of each winner when they got the prize. That is, year of prize win minus birth year.
- Use `sns.lmplot` (**not** `sns.lineplot`) to make a plot with `year` on the x-axis and `age` on the `y-axis`.

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

To get the year from a datetime column you need to use access the `dt.year` value. Here is an example:

```
a_data_frame['a_datatime_column'].dt.year
```

Seaborn's `lmplot` is a 2D scatterplot with an optional overlaid regression line. This type of plot is useful for visualizing linear relationships.

To make the plot prettier, add the arguments `lowess=True`, `aspect=2`, and `line_kws={'color' : 'black'}`.

-

Hint

Here's how to calculate `nobel['age']`:

```
nobel['age'] = nobel['year'] - nobel['birth_date'].dt.year
```

Task 9: Instructions

Plot how old winners are within the different price categories.

- As before, use `sns.lmplot` to make a plot with `year` on the x-axis and `age` on the `y-axis`. But this time, make one plot per prize category by setting the `row` argument to `'category'`.

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

This is the same plot as in task 8, except with the added `row=` argument (examples in the official Seaborn documentation here).

-

Hint

Copy and paste your solution from task 8 and then add the argument `row='category'` to the function call.

Task 10: Instructions

Pick out the rows of the oldest and the youngest winner of a Nobel Prize.

- Use `nlargest()` to pick out and display the row of the oldest winner.
- Use `nsmallest()` to pick out and display the row of the youngest winner.

Note: a test in this task relies on the expected last output of this cell based on the order of the instructions. If this is not the case, you may run into the following error after ten seconds of the cell being processed: "Your code exceeded the maximum run time permitted."

As before, you will need to use `display()` to display more than the last output of the cell. Here is the documentation for nsmallest and n_largest.

-

Hint

Here's how to display the oldest winner:

```
# The oldest winner of a Nobel Prize as of 2016
display(nobel.nlargest(1, 'age'))
```

Task 11: Instructions

- Assign the name of the youngest winner of a Nobel Prize to `youngest_winner`. The first name will suffice.

If you want to know more

The Nobel Prize dataset is rich, and this project just scratched the surface -- there is much more to explore! After you have completed this project, you can download it and continue exploring on your own! To do that you will have to install Jupyter Notebooks. Here are instructions for how to install the Jupyter Notebook interface. Good luck!

-

Hint

If you *really* can't figure this out you can always use Google search:

[*youngest Nobel Prize laureate*]

What Your Heart Rate Is Telling You

Task 1: Instructions

Load the data into our notebook.

- Use the `read.csv()` function to read in our dataset ("datasets/Cleveland_hd.csv") and save as `hd_data`.
- Use the `head()` function to print out the top 5 rows of the data.

Good to know

In this project, you'll brush up on the skills you learned in Multiple and Logistic Regression and Introduction to the Tidyverse, including basic data explorations with ggplot2 and dplyr.

Helpful links:

- RStudio's Data Wrangling cheat sheet
- RStudio's ggplot2 cheat sheet
- `read.csv()` documentation

-

Hint

The relative path to the dataset is 'datasets/Cleveland_hd.csv'. To view the top N rows of a data frame, you need to add the N as an argument in the head() function.

```
dataframe <- read.csv("relativepath/datafilename")
head(dataframe, n)
```

Task 2: Instructions

Recode the class and sex variables.

- Load the tidyverse package.
- Use the `mutate()`function to recode any > 0 value in the `class` variable to be 1 and all 0 values to remain 0. Save this new variable as `hd`.
- Use the `mutate()` function along with `factor()` to update sex (originally 0 or 1) to a factor with labels "Female" and "Male".

Helpful links:

- Mutate verb from Introduction to the Tidyverse
- `ifelse()` documentation
- `factor()` documentation

-

Hint

The `ifelse(condition, value1, value2)` can be used to recode an existing variable.

To convert a numerical variable to factor, we can use the `factor(var, levels = c(...), labels = c(...))` function:

```
new <- ifelse(old > 0, 1, 0)
factor(A, levels = 0:1, labels = c("No", "Yes"))
```

Task 3: Instructions

Explore bi-variate correlations using a t-test or a chi-squared test.

- Use `chisq.test()` to assess the relationship between `sex` and `hd` and save the output as hd_sex.
- Use `t.test()` to assess the relationship between `age` and `hd` and save the output as hd_age.
- Use `t.test()` to assess the relationship between `thalach` and `hd` and save the output as hd_heartrate.
- Check the above results using `print()`.

Helpful links:

- `chisq.test()` documentation
- `t.test()` documentation

-

Hint

If `x` and `y` are variables, the code to run a chi-squared test and a t-test, respectively, looks like this:

```
chisq.test(x, y) # x and y are both factors
t.test(y ~ x) # y is the continuous vector; x is the grouping variable
```

Task 4: Instructions

Recode the outcome `hd` variable and plot it vs. age on a boxplot.

- Label `hd` (0/1) as "No Disease" and "Disease" using the `ifelse()` function to create the new variable, `hd_labelled`.
- Generate a boxplot with `hd_labelled` on the x-axis and `age` on the y-axis.

Helpful links:

- Mutate verb from Introduction to the Tidyverse
- Boxplot exercises from Introduction to the Tidyverse
-

Hint

We will be using the same `mutate()` function in tidyverse to create the `hd_labelled` variable and overwrite the `hd_data` data frame.

```
# simple boxplot
ggplot(data = dataset, aex(x = groupingvar, y = continuousvar)) +
geom_boxplot()
```

Task 5: Instructions

Plot `hd_labelled` vs. `sex` on a barplot.

- Generate a barplot with `hd_labelled` on the x-axis and `sex` as the fill. Set `position="fill"` in `geom_bar()`. Add a `ylab()` with the label "Sex %" (one space between Sex and %).

Helpful links:

- Barplot exercises from Introduction to the Tidyverse
-

Hint

Does your plotting code look something like this?

```
# stacked barplot
ggplot(data = dataset, aes(x = groupingvar, fill = fillingvar)) +
geom_bar(position = "fill") + ylab("...")
```

Task 6: Instructions

Plot `hd_labelled` vs. `thalach` on a boxplot.

- Generate a boxplot with `hd_labelled` on the x-axis and `thalach` on the y-axis.

Helpful links:

- Boxplot exercises from Introduction to the Tidyverse
-

Hint

Does your plotting code look something like this?

```
# simple boxplot
ggplot(data = dataset, aex(x = groupingvar, y = continuousvar)) +
geom_boxplot()
```

Task 7: Instructions

Fit a logistic regression model with all three variables.

- Use `glm()` to fit the model (predicting `hd` using `age`, `sex`, and `thalach`) to save to `model`. Set the `family` parameter to 'binomial'.
- Get the summary from the `model` object.

We specify the correct error distribution using the `family` argument. In this case, 'binomial' is appropriate because we are working with a binary outcome.

Helpful links:

- `glm()` documentation
- How to fill out the `formula` parameter in `glm()`, via the `formula()` documentation
-

Hint

Does your model fitting code look something like this?

```
logistic_model <- glm(data = data, y ~ x1 + x2, family = 'binomial')
```

The `summary()` function returns the model summary.

Task 8: Instructions

Use the broom package to tidy up the model output.

- Load the broom package.
- Apply the `tidy()` function on the `model` object created previously and save as `tidy_m`.
- Calculate Odds Ratios (ORs) by exponentiating the `estimate` column.
- Calculate the normal approximation for the upper bound of the 95% Confidence Interval (CI) for ORs.

Helpful links:

- Introduction to broom
- Normal approximation CI
-

Hint

The `tidy()` function takes a model object as input. The `estimate` and `std.error` columns in the `tidy_m` data frame will be used to calculate OR and the 95% CI.

Task 9: Instructions

Extract the predicted probability in the current dataset. In addition, apply the model to predict outcomes on new cases.

- Apply the `predict()` function.
- Using `ifelse()`, create a decision rule for `pred_prob >= 0.5` and save the predicted decision (either 1 or 0) as `pred_hd`.
- Using `predict()`, predict probability of HD on the provided `newdata` case and save as `pred_new`.

We include the argument `type="response"` in `predict()` to get prediction probability.

Helpful links:

- `predict()` documentation
- `ifelse()` documentation
-

Hint

The predict function takes a model object and a data frame in which to look for variables with which to predict.

```
predict(model, data, type='response')
```

Task 10: Instructions

Create four model metrics, including AUC, Accuracy, Classification Error, and the confusion matrix.

- Load the Metrics package.
- Use `auc()`, `accuracy()`, and `ce()` to get the first three metrics.
- Obtain the confusion matrix for `hd_data$hd` and `hd_data$pred_hd` using the `table()` function. Use the `dnn` argument to customize the table row/column names to be 'True Status' and 'Predicted Status', respectively.

`auc()`, `accuracy()`, and `ce()` all take the true label as the first argument and predicted outcome as the second argument

Helpful links:

- Metrics package documentation
-

Hint

You can extract the relevant columns from the dataset for functions in Metrics like so (using `auc()` as an example):

```
auc <- auc(data$truestatus, data$predictedstatus)
```

Classify Song Genres from Audio Data

Task 1: Instructions

Read in the data using `pandas` and merge the DataFrames into one usable dataset.

- Using the pandas `read_csv()` function, read in the file with the track metadata (`datasets/fma-rock-vs-hiphop.csv`) and name the DataFrame `tracks`.
- Using the pandas `read_json()` function, read in the JSON file with the track acoustic metrics (`datasets/echonest-metrics.json`) and name the DataFrame `echonest_metrics`. Set the `precise_float` argument to `True` when reading in your data.
- Merge the DataFrames on matching `track_id` values. Only retain the `track_id` and `genre_top` columns of `tracks`. `echonest_metrics` should be the first (left) data frame in the merge.
- Inspect the DataFrame using the `.info()` method.

Good to know

This project lets you apply what you learned in Supervised Learning with scikit-learn, plus data preprocessing, dimensionality reduction, and machine learning using the `scikit-learn` package. We recommend you are familiar with these topics before starting this project.

Helpful links:

- Documentation for pandas `read_csv()`, `read_json()` and `pd.merge()` functions
- Variance of the PCA features exercise
- Train/test/split + Fit/Predict/Accuracy exercise
-

Hint

You can select columns of a DataFrame by providing a list to the indexer i.e. using the `[]`. A correct solution for the merge looks like this:

```
echo_tracks = echonest_metrics.merge(tracks[['column1_name',
'column2_name']], on='column2_name')
```

Task 2: Instructions

Explore correlations in our dataset using pandas `corr` function.

- Visually inspect the correlation table generated from `DataFrame.corr()` for any strong correlations.

Helpful links:

- pandas DataFrame `corr` method [documentation]

-

Hint

Does your code look something like this?

```
corr_metrics = df.corr()
```

Task 3: Instructions

Prepare our dataset for training a model, and standardize the data.

- Define our features from `echo_tracks` by removing `genre_top` and `track_id` from the DataFrame using `DataFrame.drop()` along axis 1.
- Define our labels -- in this case, the `genre_top` column from `echo_tracks`.
- Import the `StandardScaler` from the `sklearn.preprocessing` module
- Define an instance of the `StandardScaler` called `scaler` without passing any arguments and use the `fit_transform` method to scale `features` and save to a new variable called `scaled_train_features`

Helpful links:

- pandas `DataFrame.drop()` method [documentation]
- Square brackets in Pandas [exercise]

-

Hint

The `axis` argument of the `drop()` method by convention considers `0` for rows and `1` for columns to look through to drop. You can drop columns named `col1` and `col2` from the dataframe `df` and save to a new dataframe called `df_drop` like so:

```
df_drop = df.drop(columns=['col1', 'col2'])
```

You can use the `StandardScaler` to standardize the data in `my_data` like so after you define an instance, here called `scaler` using the `fit_transform` method of the `StandardScaler`.

```
scaler = StandardScaler()
```

```
scaler.fit_transform(my_data)
```

Task 4: Instructions

Use PCA to determine the explained variance of our features.

- Import the `matplotlib.pyplot` module as `plt`, and our `PCA()` class from `sklearn.decomposition`
- Create our PCA class using `PCA()`, fit the model on our `scaled_train_features` using `PCA.fit()`, and retrieve the explained variance ratio
- Make a scree plot of the variance explained by each component

---

We run PCA on all our features at first, which is done by default if `n_components` is not specified.

Helpful links:

- sklearn PCA documentation
- matplotlib `bar` plot documentation

-

Hint

You can get the number of components and the explained variance ratio of an array of features named `my_features` from a `PCA` object called `pca` like this after fitting `pca`.

```
pca.fit(my_features)
print(pca.explained_variance_ratio_)
print(pca.n_components_)
```

You can create a barplot by passing first the x-values which hold x-coordinates of each bar and then the corresponding y-values which hold the value of each bar, like so:

```
fig, ax = plt.subplots()
ax.bar(range(6), [5,1,0,2,3,0])
```

When creating the plot, keep in mind that the number of components goes on the x-axis and the explained variance on the y-axis.

Task 5: Instructions

Plot the cumulative explained variance of our PCA.

- Import the `numpy` package as `np`.

- Calculate the cumulative sums of our explained variance using `np.cumsum()`.
- Plot the cumulative explained variances using `ax.plot` and look for the number of components at which we can account for >85% of our variance; assign this to `n_components`.
- Perform PCA using `n_components` and project our data onto these components.

Helpful links:

- numpy `cumsum()` function documentation
- sklearn PCA documentation

-

Hint

Don't forget that Python indexing starts at 0 when looking at your plot.

You can use a `PCA` object that has been trained on an array named `my_features` to then project the data onto the components like this:

```
pca.transform(my_features)
```

Task 6: Instructions

Prepare our training and test sets and train our first classifier.

- Import the `train_test_split()` function from `sklearn.model_selection` module
- Import the `DecisionTreeClassifier` from `sklearn.tree` module
- Split our projected data into train and test, features and labels, respectively using `train_test_split()` with `random_state=10`.
- Create our decision tree classifier using `DecisionTreeClassifier()` and `random_state=10` and train the model using the `model.fit()` notation
- Find the predicted labels of the `test_features` from our trained model using the `model.predict()` notation.

Helpful links:

- scikit-learn `train_test_split()` function documentation
- scikit-learn `DecisionTreeClassifier()` class documentation

-

Hint

The `train_test_split()` function can be used to split `features` and `labels` as training and test sets like this after importing it from `sklearn.model_selection`:

```
from sklearn.model_selection import train_test_split
train_features, test_features, train_labels, test_labels =
train_test_split(features, labels)
```

You can fit your decision tree classifier named `tree` and train on your `train_features` and `train_labels` like this:

```
tree = DecisionTreeClassifier()
tree.fit(train_features, train_labels)
```

Don't forget to set your `random_state` for any function that might use some randomness in order to make your result reproducible.

Task 7: Instructions

Train our logistic regression and compare the performance with our decision tree.

- Create our logistic regression model using LogisticRegression() and set `random_state` to `10`.
- Train the model using the `model.fit()` notation and assign the predicted labels for the `test_features` to `pred_labels_logit`.
- Import the `classification_report` from the `sklearn.metrics` package
- Print the classification reports for our trained Decision Tree and Logistic Regression models.

Helpful links:

- scikit-learn `LogisticRegression()` class documentation
- scikit-learn `classification_report()` function documentation

-

Hint

You can fit your `LogisticRegression` named `my_logreg` on `features` and `labels` like this after instantiating an instance of `LogisticRegression`:

```
my_logreg = LogisticRegresion()
my_logreg.fit(features, labels)
```

You can get the classification report from the labels of each data point called `labels` and the predicted labels called `predicted` like this:

```
report = classification_report(labels, predicted)
```

Task 8: Instructions

Balance our dataset such that the number of tracks for each genre is the same.

- Subset only the hip-hop tracks from `echo_tracks` using `df.loc[]`, and the same for the rock tracks
- Sample `rock_only` such that there is the same number of data points as there are hip-hop data points. Set the `random_state` to 10.
- Concatenate the `rock_only` and `hop_only` (in that order) DataFrames using the `pd.concat()` function by passing a list of these DataFrames.
- Redefine our train and test sets using `train_test_split` with the PCA projection of the balanced dataframe.

Helpful links:

- pandas `DataFrame.loc[]` indexing documentation
- pandas `concat()` function documentation
- pandas `DataFrame.sample()` method documentation

-

Hint

You can filter rows of a dataframe named `df` which satisfy a boolean statement where the value in a column named `col1` is greater than 10, and then sample only 20 of these rows like this:

```
df_filter_sample = df.loc[df['col1'] > 10].sample(20, random_state=0)
```

You can concatenate two DataFrames named `df1` and `df2` like this:

```
df_concat = pd.concat([df1, df2])
```

Whenever you're using a function that takes a random sample, or shuffles data, you should set the `random_state` if you want your result to be reproducible.

Task 9: Instructions

Compare the two model performances on the balanced data.

- Create and train your decision tree using `DecisionTreeClassifier()` and a random state of `10`, then predict on the `test_features`.
- Create and train your logistic regression using `LogisticRegression()` and a random state of `10`, then predict on the `test_features`.
- Compare the performance of the two models using `classification_report()`.

-

Hint

You can predict labels on features called `my_features` using a scikit-learn object called `my_model` after fitting `my_model` on the training data, `train_features` and `train_labels` like this:

```
my_model.fit(train_features, train_labels)
predicted = my_model.predict(my_features)
```

Most of the classifiers initialize or run with some degree of randomness, you should set your `random_state` to make the result reproducible.

Task 10: Instructions

Use cross-validation to get a better sense of your model performance.

- Create a variable called `kf` to store your cv using `KFold()` with `10` folds.
- Train each of your models using `cross_val_score()` on the original `pca_projection` and `labels` variables.
- Print the mean of the cross-validation scores for each model using `np.mean()`.

Helpful links:

- `KFold()` documentation
- `cross_val_score()` documentation

-

Hint

You can train and cross-validate a scikit-learn model named `my_model` after defining our KFold cross-validation named `kf` like this:

```
kf = KFold(5)
cv_score = cross_val_score(my_model, features, labels, cv=kf)

import pandas as pd

# Read in track metadata with genre labels
tracks = pd.read_csv('datasets/fma-rock-vs-hiphop.csv')

# Read in track metrics with the features
echonest_metrics = pd.read_json('datasets/echonest-metrics.json',
precise_float=True)
```

```
# Merge the relevant columns of tracks and echonest_metrics
echo_tracks = echonest_metrics.merge(tracks[['genre_top', 'track_id']],
on='track_id')

# Inspect the resultant dataframe
echo_tracks.info()

# Create a correlation matrix
corr_metrics = echonest_metrics.corr()
corr_metrics.style.background_gradient()

# Define our features
features = echo_tracks.drop(columns=['genre_top', 'track_id'])

# Define our labels
labels = echo_tracks['genre_top']

# Import the StandardScaler
from sklearn.preprocessing import StandardScaler

# Scale the features and set the values to a new variable
scaler = StandardScaler()
scaled_train_features = scaler.fit_transform(features)

# This is just to make plots appear in the notebook
%matplotlib inline

# Import our plotting module, and PCA class
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Get our explained variance ratios from PCA using all features
pca = PCA()
pca.fit(scaled_train_features)
exp_variance = pca.explained_variance_ratio_

# plot the explained variance using a barplot
fig, ax = plt.subplots()
ax.bar(range(pca.n_components_), exp_variance)
ax.set_xlabel('Principal Component #')

import numpy as np

# Calculate the cumulative explained variance
cum_exp_variance = np.cumsum(exp_variance)

# Plot the cumulative explained variance and draw a dashed line at 0.85.
fig, ax = plt.subplots()
ax.plot(cum_exp_variance)
ax.axhline(y=0.85, linestyle='--')

# choose the n_components where about 85% of our variance can be explained
n_components = 6

# Perform PCA with the chosen number of components and project data onto
components
pca = PCA(n_components, random_state=10)
pca.fit(scaled_train_features)
pca_projection = pca.transform(scaled_train_features)
```

```
# Import train_test_split function and Decision tree classifier
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Split our data
train_features, test_features, train_labels, test_labels =
train_test_split(
    pca_projection, labels, random_state=10)

# Train our decision tree
tree = DecisionTreeClassifier(random_state=10)
tree.fit(train_features, train_labels)

# Predict the labels for the test data
pred_labels_tree = tree.predict(test_features)

# Import LogisticRegression
from sklearn.linear_model import LogisticRegression

# Train our logisitic regression
logreg = LogisticRegression(random_state=10)
logreg.fit(train_features, train_labels)
pred_labels_logit = logreg.predict(test_features)

# Create the classification report for both models
from sklearn.metrics import classification_report
class_rep_tree = classification_report(test_labels, pred_labels_tree)
class_rep_log = classification_report(test_labels, pred_labels_logit)

print("Decision Tree: \n", class_rep_tree)
print("Logistic Regression: \n", class_rep_log)

# Subset a balanced proportion of data points
hop_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Hip-Hop']
rock_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Rock']

# subset only the rock songs, and take a sample the same size as there are
hip-hop songs
rock_only = rock_only.sample(hop_only.shape[0], random_state=10)

# concatenate the dataframes hop_only and rock_only
rock_hop_bal = pd.concat([rock_only, hop_only])

# The features, labels, and pca projection are created for the balanced
dataframe
features = rock_hop_bal.drop(['genre_top', 'track_id'], axis=1)
labels = rock_hop_bal['genre_top']
pca_projection = pca.fit_transform(scaler.fit_transform(features))

# Redefine the train and test set with the pca_projection from the balanced
data
train_features, test_features, train_labels, test_labels =
train_test_split(
    pca_projection, labels, random_state=10)

# Train our decision tree on the balanced data
tree = DecisionTreeClassifier(random_state=10)
tree.fit(train_features, train_labels)
pred_labels_tree = tree.predict(test_features)

# Train our logistic regression on the balanced data
```

```
logreg = LogisticRegression(random_state=10)
logreg.fit(train_features, train_labels)
pred_labels_logit = logreg.predict(test_features)

# compare the models
print("Decision Tree: \n", classification_report(test_labels,
pred_labels_tree))
print("Logistic Regression: \n", classification_report(test_labels,
pred_labels_logit))

from sklearn.model_selection import KFold, cross_val_score

# Set up our K-fold cross-validation
kf = KFold(10)

tree = DecisionTreeClassifier(random_state=10)
logreg = LogisticRegression(random_state=10)

# Train our models using KFold cv
tree_score = cross_val_score(tree, pca_projection, labels, cv=kf)
logit_score = cross_val_score(logreg, pca_projection, labels, cv=kf)

# Print the mean of each array o scores
print("Decision Tree:", np.mean(tree_score), "Logistic Regression:",
np.mean(logit_score))
```

Explore 538's Halloween Candy Rankings

Task 1: Instructions

In this task you'll load all the packages you need and take a first glance at the data you'll be using.

- Use the `library()` function to load the packages `dplyr`, `tidyr`, `ggplot2`, `broom`, `corrplot`, and `fivethirtyeight`.
- Make the `candy_rankings` dataset available using the `data()` function.
- Use `glimpse()` to take a look at `candy_rankings`.

Good to know

In this project you'll brush up on the skills you learned in Multiple and Logistic Regression, along with some basic data explorations with `ggplot2`, `dplyr`, and `tidyr`.

Helpful links:

- RStudio's Data Wrangling cheat sheet
- RStudio's `ggplot2` cheat sheet

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Hint

You can load packages using the `library()` function. For example, `library(tidyverse)` will load the `tidyverse` package. To put the `candy_rankings` dataset in the global environment, just call `data()` with `candy_rankings` as its argument. Finally, the `glimpse()` function just takes the dataset as its first argument.

Task 2: Instructions

Now you'll do some data wrangling and make your first plot.

- Create `candy_rankings_long` by `gather()`-ing all the logical columns into the columns `feature` and `value`. Select all the columns from `chocolate` to `pluribus`.
- Make a bar plot of `candy_rankings_long` faceted by `feature` by adding `facet_wrap()` to your plot.

Helpful links:

- `gather()` documentation
-

Hint

To `gather()` the data write code like

```
gather(data, "key", "value", columns)
```

This will create the columns `key` and `value` containing the information stored in the `columns` of `data`.

Once you have `candy_rankings_long` you can plot is using `ggplot2`. To do this use `geom_bar()` and `facet_wrap()`.

Task 3: Instructions

Make a lollipop chart to visualize the values of `pricepercent` using `ggplot2`.

- Using `candy_rankings` plot `competitorname` (reordered by `pricepercent`) on the x-axis and `pricepercent` on the y-axis.
- Use `geom_segment()` with aesthetics `xend=` and `yend=` to draw the lollipop "sticks".
- Next, use `geom_point()` to draw the ends of the lollipops.
- Finish up with `coord_flip()` to make the chart easier to read.

If you haven't seen a lollipop chart before, you can take a look at some examples (with code!) here.

Helpful links:

- `geom_segment()`documentation
-

Hint

The correct structire of the code should look similar to this:

```
# Make a lollipop chart of pricepercent
ggplot(candy_rankings, aes(reorder(...., ....), ....)) +
  geom_segment(aes(xend = reorder(...., ....), yend = 0)) +
  geom_point() +
```

```
  coord_flip()
```

Task 4: Instructions

To explore the values of `winpercent` with a histogram.

- Use `ggplot2` to make a histogram of the values of `winpercent` from `candy_ranksings.

Helpful hints:

- ggplot2 histograms
-

Hint

To make a histogram, use `geom_histogram()`. You only need to specify the `x` aesthetic here.

Task 5: Instructions

Make a lollipop chart very similar to the one you already created.

- Make a lollipop chart of `winpercent`, just like you did for `pricepercent` in Task 3. Don't forget about reordering the aesthetics and flipping the coordinates.

Helpful links:

- `geom_segment()`documentation.
-

Hint

The correct structure of the code looks something like this:

```
# Make a lollipop chart of winpercent
ggplot(candy_rankings, aes(reorder(...., ....), ....)) +
  geom_segment(aes(xend = reorder(...., ....), yend = 0)) +
  geom_point() +
  coord_flip()
```

Task 6: Instructions

Now that you have a sense of how the variables behave individually, make a correlation plot of how they interact with one another.

- Using the pipe operator, compute the correlation matrix of all the variables **except** `competitorname`, and plot the correlation matrix using `corrplot()`.

Helpful links:

- `cor()` documentation
- `corrplot()` documentation

-

Hint

To plot the correlation matrix, write a pipe like:

```
data %>%
  select(columns) %>%
  cor() %>%
  corrplot()
```

Task 7: Instructions

Model your data using a linear model!

- Use the `lm()` function to fit a linear model of the `winpercent` as a function of all the variables except `competitorname`.

To use all the variables in a dataset you can include a `.` in your model. Excluding variables can be done by placing a `-` (minus sign) before them in the formula. So a formula might look something like this: `response ~ . -excluded_variable`.

-

Hint

Call the `lm()` function with the formula `winpercent ~ . -competitorname`. Don't forget to include the `data` argument.

Task 8: Instructions

Take a look at the results of your linear model.

- Print the `summary()` of `win_mod` to look at the coefficients, the R-squares, and the p-values.
- To check for violations of the assumptions of a linear model, use `augment()` from the `broom` package on the model output to plot the fitted values vs. the residuals. Add a horizontal line at `y = 0`.

In a good linear model, the residuals should look like they're scattered around zero without much structure.

Helpful links:

- `broom`'s `augment()` documentation
-

Hint

To see which coefficients are significant, along with the coefficient values, call the `summary()` function on the model object.

You can use `augument()` within the call to `ggplot()` like:

```
ggplot(augument(lm_model_output), aes(...., ....)) +
    geom_point()
```

The fitted values are stored in the `.fitted` column and the residuals are stored in the `.resid` column. Add a horizontal line using `geom_hline()` with the `yintercept =` argument.

Task 9: Instructions

Create a logistic regression model of the variable `chocolate`.

- Using `glm()` fit a logistic regression of `chocolate` modeled by all the other variables except `competitorname`. `chocolate` is a binary variable - don't forget to choose the correct family.

Helpful links:

- `glm()` documentation
- `glm()` families
-

Hint

To fit a logistic regression, use the `glm()` function. The formula should be very similar to the one you used in the linear model. Don't forget to include the `family` argument.

Task 10: Instructions

Evaluate the performance of your logistic regression model.

- Print the `summary()` of `choc_mod` and take a look at the coefficients.
- Create a new data frame, `preds`, of predicted probabilities by `augment()`-ing the model output with `type.predict = "response"`. Convert these probabilities to predictions (stored in a new `prediction` column) using a threshold of 0.5 on the `.fitted` values.
- Select `chocolocate` and `prediction` and create a confusion matrix of the predictions using `table()`.
- Calculate and print the accuracy (the percent of predictions that were correct).

Helpful links:

- `broom`'s `augment()` documentation
-

Hint

To view the coefficient estimates, call the `summary()` function on `choc_mod()`.

To make the predictions from the raw probabilities use `mutate(prediction = .fitted > 0.5)`. Then you can `select()` the `chocolate` and `prediction` columns and use `table()` to make a confusion matrix.

To calculate the accuracy of the confusion matrix, divide the sum of the diagonal values (`diag()`) by the sum of all the values.

Reducing Traffic Mortality in the USA

Task 1: Instructions

Explore your current folder and view the main dataset file.

- Check the name of the current folder using `!pwd`.
- List all files in this folder using `!ls`.
- List all files in the `datasets\` folder using `!ls` and the name of the folder.
- View the first 20 lines of `road-accidents.csv` in the `datasets\` folder using `!head`.

Good to know

This project lets you apply skills from:

- Introduction to Shell, including how to navigate the file system and view files
- Data Manipulation with pandas, including reading, exploring, filtering, grouping, and reshaping data
- Merging DataFrames with pandas, including how two merge two DataFrames
- Unsupervised Learning in Python, including KMeans clustering, dimensionally reduction through PCA, and visualizations using `matplotlib`
- Supervised Learning with scikit-learn, including multivariate regression
- Intermediate Python for Data Science, including visualizations using `matplotlib`
- Intermediate Data Visualization with Seaborn, including statistical visualizations using `seaborn`

We recommend that you review the appropriate sections of those courses before starting this project.

Here are three charts illustrating the road accident fatality situation described in the notebook's first paragraph.

Helpful links:

- Manipulating Files and Directories
- How to run shell commands in a Jupyter Notebook (through the underlying IPython interpreter)
- For viewing the first few lines of a file using shell commands, exercise number 3 and number 5 in the Manipulating Files and Directories

-

Hint

Preface the shell command with `!` so that the Jupyter Notebook knows to interpret it as a shell command rather than a Python variable, e.g. `!ls` to list files in the directory.

Does your code for the third bullet look like this:

```
accidents_head = !head -n <NUMBER_OF_LINES> <PATH/TO_FILE>
```

Task 2: Instructions

Read in the main dataset file and start exploring the data.

- Import the `pandas` module aliased as `pd`.
- Read in `road-accidents.csv` (which is in the `datasets/` folder) using `read_csv()` from `pandas`. Set the `comment` and `sep` parameters based on the output from task 1.
- Save the number of rows columns as a tuple, using the `shape` attribute.
- Generate an overview of the DataFrame using the `info()` method.

Does the output from these commands make sense? A quick data type sanity check can save us major headaches down the line.

Look into the documentation of `read_csv` (with `pd.read_csv?`), to find out how to use the `comment` and `sep` parameters and specify `'#'` for comments and `'|'` as the separator.

Helpful links:

- pandas cheat sheet
- pandas `read_csv()` function documentation
- Reading a flat file exercise in the pandas Foundations course

-

Hint

Remember that the dataset is located within the `datasets` folder, so the full path from the current directory to the dataset is `'datasets/road-accidents.csv'`.

From the output of the `head` command in the previous task, we can see that there are comments in the CSV file that are prefixed with `#` and the separators for the values is `|` rather than `,`.

A correct version of the `read_csv()` code looks like this:

```
car_acc = pd.read_csv('datasets/road-accidents.csv', comment='<COMMENT>',
sep='<SEPARATOR>')
```

Task 3: Instructions

Create a textual and graphical overview of the data.

- Compute the summary statistics of all columns in the `car_acc` DataFrame, using the `describe()` method.
- Create a pairwise scatter plot to explore the data, using `sns.pairplot()`.

Helpful links:

- sns.pairplot lecture
- seaborn pairplot documentation
-

Hint

`sns.pairplot` takes one argument: the variable name of DataFrame to be plotted.

Task 4: Instructions

Explore the correlation between all column pairs in the DataFrame.

- Compute the correlation coefficient for all column pairs in `car_acc`, using the `corr()` method.

By default, the Pearson correlation coefficient will be computed.

Helpful links:

- pandas corr method
-

Hint

Call the `corr()` method on the DataFrame without any arguments.

Task 5: Instructions

Fit a multivariate linear regression model using the fatal accident rate as the outcome.

- Import the `linear_model` function from `sklearn`.

- Create the features and target DataFrames, by subsetting the DataFrame `car_acc`.
- Create a linear regression object, using `linear_model.LinearRegression()`.
- Fit a multivariate linear regression model, using `fit()`.
- Retrieve the regression coefficients from the `coef_` attribute of the fitted regression object.

Helpful links:

- scikit-learn linear regression
-

Hint

The `features` DataFrame should contain the following columns `'perc_fatl_speed'`, `'perc_fatl_alcohol'`, `'perc_fatl_1st_time'`. These columns need to be passed as a list to subset the original DataFrame.

The `'target'` DataFrame should contain the `'drvr_fatl_col_bmiles'` column.

To see the help documentation for how to fit the regression, use `fit.reg?`. Remember that the `X` parameter corresponds to the features and the `y` parameter is the target variable.

Task 6: Instructions

Perform a principal component analysis on the standardized data.

- Standardize and center the feature columns, using the `StandardScaler` from `sklearn` and its `fit_transform()` method.
- Import the PCA class from `sklearn`.
- Fit the standardized data to the PCA class using its `fit()` method.
- Compute the cumulative proportion of variance explained by the first two principal components, either by adding them together or by using the cumulative summation method (`cumsum`) of the explained variance array.

Helpful links:

- scikit-learn standard scaler
- scikit-learn PCA
- Visualizing the PCA variance explained
- PCA variance explained exercise
-

Hint

To standardize the features, the `features` DataFrame should be passed to `scaler.fit_transform()`.

The proportion of variance explained is stored in `pca.explained_variance_ratio_`.

If using the `cumsum()` method of `pca.explained_variance_ratio_`, remember that Python indexing starts at `0` so the sum of the first two components corresponds to position `1` in the array returned from `cumsum()`.

Task 7: Instructions

Transform the data and visualize the first two principal components in a scatter plot.

- Create a `PCA` object with two components. Assign the result to the variable, `pca`.
- Transform the scaled features using two principal components and the `fit_transform()` method of the `PCA` object.
- Extract the first and second component to use for the scatter plot. Assign the results to `p_comp1` and `p_comp2`, respectively.
- Plot the first two principal components in a scatter plot, using `plt.scatter`.

The `n_components` parameter controls the number of components with which to initialize the `PCA` class.

The scatter plot should look something like this.

Helpful links:

- Subsetting numpy arrays
-

Hint

All values from an axis in a numpy array can be extracted using `:`. Combine this with `0` or `1` to subset `p_comps` for the first and second principal component, respectively.

The `p_comp1` code should look like this:

```
p_comp1 = p_comps[:, 0]
```

Task 8: Instructions

Cluster the states using the KMeans algorithm and visualize the explanatory power for different numbers of clusters.

- Import `KMeans` from `sklearn.cluster`.
- Initialize the `KMeans` object using the current number of clusters (k).
- Fit the scaled features to the `KMeans` object.
- Append the inertia for `km` to the list of inertias.
- Plot the results in a line plot using `matplotlib.pyplot.plot`. This type of plot is also called a scree plot.

The line plot should look something like this.

Helpful links:

- scikit-learn KMeans
- Evaluating a clustering

-

Hint

The `k` variable will be updated with the next value in `ks` for each iteration in the loop, so the setting `n_clusters=k` means that the clustering will be performed for all the numbers in `ks`. To be able to access the results after all iterations of the loop are done, the inertias are appended to a list instead of overwriting the list with each new iteration.

For the line plot, use `ks` for the x-axis and `inertias` for the y-axis.

Task 9: Instructions

Highlight the clusters of the K-means fit with three clusters in the PCA scatter plot.

- Create a `KMeans` object with 3 clusters, setting `random_state` to 8 as in the previous task.
- Fit the data to the `km` object.
- Create a scatter plot of the first two principal components and color it according to the KMeans cluster assignment.

The scatter plot should look something like this.

Helpful links:

- matplotlib scatter plot

-

Hint

When plotting the data, use the same subsetting technique as in task 7, but instead of assigning the value to a new variable, subset the array within the call to `plt.scatter()`.

Since the order of `km.labels_` is the same as the order of `p_comps`, the color of the scatter plot can be set by specifying `c=km.labels_`. Matplotlib has both names and numbers assigned to colors, so in this case, it will use the colors '0', '1', and '2'.

The plotting code should look something like this:

```
plt.scatter(p_comps[:, 0], p_comps[:, 1], c=km.labels_)
```

Task 10: Instructions

Visualize the distribution of speeding, alcohol influence and percentage of first-time accidents in a direct comparison of the clusters.

- Create a new column with the labels from the KMeans clustering, using `km.labels_`.
- Reshape the DataFrame to the long format, using `pd.melt()`. Use the features as the value variables and give them the name `'measurement'` in the new DataFrame. Name the value column `'percent'`.
- Create a violin plot splitting and coloring the results according to the km-clusters using the `hue` parameter. Plot the measurements along the y-axis and the percent values along the x-axis.

The violin plot should look something like this.

Helpful links:

- Creating long DataFrames in pandas
- seaborn violin plots
-

Hint

Just as in the previous task, remember that the order of `km.labels_` is the same as the order of observations in the DataFrame. Therefore, `km.labels_` can be assigned to `car_acc['cluster']` without worrying about reordering the values.

Use `pd.melt?` to see how to specify the arguments for this functions. Importantly, the DataFrame is passed as an unnamed argument, while the remaining arguments are passed to named parameters. An example of how this could look would be something like this (note that the words surrounded by `[` and `]` need to be replaced by the correct variable or column names):

```
pd.melt(car_acc, id_vars='<name_of_cluster_column>',
var_name='<name_of_measurement_column>',
```

```
        value_name='<name_of_percent_column>',
value_vars=<list_of_the_three_feature_columns>)
```

In the violin plot, the `hue` parameters should be set to `'clusters'` to group and color points according to their cluster membership.

Task 11: Instructions

Add data on the number of miles driven per state to compute total number of fatal accidents and total accidents for each cluster.

- Merge the `car_acc` DataFrame with the `miles_driven` DataFrame. Merge on the common column `state`.
- Create a new column for the number of drivers involved in fatal accidents. Use the columns `'drvr_fatl_col_bmiles'` and `'million_miles_annually'`, note that these are in billions and million respectively.
- Calculate the number of states in each cluster and their average and total number of drivers involved in fatal accidents using the DataFrame `agg()` method.
- Using `sns.barplot`, create a bar plot of the total number of fatal accidents per cluster, setting the `estimator` parameter to `sum`.

The bar plot should look something like this.

The confidence intervals in the bar plot can be removed by setting `ci=None`, which is done in the provided sample code.

Helpful links:

- Exercise on merging pandas DataFrames on a column
- Performing multiple aggregations in pandas DataFrames

-

Hint

`pd.merge()` has an `on` parameter that can be used to specify which column to merge the datasets on. Specifying this as `'state'` ensures that the values for the states line up when merging even if the two DataFrames are not in the same order.

For the new column `'num_drvr_fatl_col'` remember to divide by 1000 to convert from billions to millions.

To perform multiple aggregations with the `agg()` method, pass the aggregations as a list, e.g., `['count', 'mean', 'sum']`.

For the barplot, the total number of accidents per cluster can be found by setting `estimator=sum`.

Task 12: Instructions

Decide which cluster to focus your resources on.

- Which cluster would you choose: 1, 2, or 3? Assign one of these integers to `cluster_num`.

-

Hint

Assign 0, 1, or 2 to `cluster_num` and think about how to motivate the choice.

Check the name of the current directory

current_dir = !pwd

print(current_dir)

List all files in this directory

file_list = !ls

print(file_list)

List all files in the datasets directory

dataset_list = !ls datasets

print(dataset_list)

View the first 20 lines of datasets/road-accidents.csv

accidents_head = !head -n 20 datasets/road-accidents.csv

accidents_head

Import the `pandas` module as "pd"

import pandas as pd

```
# Read in `road-accidents.csv`
car_acc = pd.read_csv('datasets/road-accidents.csv', comment='#', sep='|')

# Save the number of rows columns as a tuple
rows_and_cols = car_acc.shape
print('There are {} rows and {} columns.\n'.format(
    rows_and_cols[0], rows_and_cols[1]))

# Generate an overview of the DataFrame
car_acc_information = car_acc.info()
print(car_acc_information)

# Display the last five rows of the DataFrame
car_acc.tail()
# import seaborn and make plots appear inline
import seaborn as sns
%matplotlib inline

# Compute the summary statistics of all columns in the `car_acc` DataFrame
sum_stat_car = car_acc.describe()
print(sum_stat_car)

# Create a pairwise scatter plot to explore the data
sns.pairplot(car_acc)
```

```
# Compute the correlation coefficent for all column pairs
corr_columns = car_acc.corr()
corr_columns
# Import the linear model function from sklearn
from sklearn import linear_model

# Create the features and target DataFrames
features = car_acc[['perc_fatl_speed', 'perc_fatl_alcohol', 'perc_fatl_1st_time']]
target = car_acc['drvr_fatl_col_bmiles']

# Create a linear regression object
reg = linear_model.LinearRegression()

# Fit a multivariate linear regression model
reg.fit(features, target)

# Retrieve the regression coefficients
fit_coef = reg.coef_
fit_coef

# Standardize and center the feature columns
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)
```

```
# Import the PCA class from sklearn

from sklearn.decomposition import PCA

pca = PCA()


# Fit the standardized data to the pca

pca.fit(features_scaled)


# Plot the proportion of variance explained on the y-axis of the bar plot

import matplotlib.pyplot as plt

plt.bar(range(1, pca.n_components_ + 1),  pca.explained_variance_ratio_)

plt.xlabel('Principal component #')

plt.ylabel('Proportion of variance explained')

plt.xticks([1, 2, 3])


# Compute the cumulative proportion of variance explained by the first two principal
components

two_first_comp_var_exp = pca.explained_variance_ratio_.cumsum()[1]

print("The cumulative variance of the first two principal componenets is {}".format(

    round(two_first_comp_var_exp, 5)))


# Transform the scaled features using two principal components

pca = PCA(n_components=2)

p_comps = pca.fit_transform(features_scaled)


# Extract the first and second component to use for the scatter plot
```

```
p_comp1 = p_comps[:, 0]
p_comp2 = p_comps[:, 1]

# Plot the first two principal components in a scatter plot
plt.scatter(p_comp1, p_comp2)

# Import KMeans from sklearn
from sklearn.cluster import KMeans

# A loop will be used to plot the explantory power for up to 10 KMeans clusters
ks = range(1, 10)
inertias = []
for k in ks:
    # Initialize the KMeans object using the current number of clusers (k)
    km = KMeans(n_clusters=k, random_state=8)
    # Fit the scaled scaled features to the KMeans object
    km.fit(features_scaled)
    # Append the inertia for `km` to the list of inertias
    inertias.append(km.inertia_)

# Plot the results in a line plot
plt.plot(ks, inertias, marker='o')

# Create a KMeans object with 3 clusters
km = KMeans(n_clusters=3, random_state=8)
```

```
# Fit the data to the `km` object

km.fit(features_scaled)

# Create a scatter plot of the first two principal components

# and color it according to the KMeans cluster assignment

plt.scatter(p_comps[:, 0], p_comps[:, 1], c=km.labels_)


# Create a new column with the labels from the KMeans clustering

car_acc['cluster'] = km.labels_


# Reshape the DataFrame to the long format

melt_car = pd.melt(car_acc, id_vars='cluster', var_name='measurement',
value_name='percent',

                   value_vars=['perc_fatl_speed', 'perc_fatl_alcohol', 'perc_fatl_1st_time'])


# Create a violin plot splitting and coloring the results according to the km-clusters

sns.violinplot(y='measurement', x='percent', data=melt_car, hue='cluster')


# Read in the `miles-drives.csv`

miles_driven = pd.read_csv('datasets/miles-driven.csv', sep='|')


# Merge the `car_acc` DataFrame with the `miles_driven` DataFrame

car_acc_miles = car_acc.merge(miles_driven, on='state')


# Create a new column for the number of drivers involved in fatal accidents

car_acc_miles['num_drvr_fatl_col'] = car_acc_miles['drvr_fatl_col_bmiles'] *
car_acc_miles['million_miles_annually'] / 1000
```

```
# Create a barplot of the total number of accidents per cluster

sns.barplot(x='cluster', y='num_drvr_fatl_col', data=car_acc_miles, estimator=sum, ci=None)



# Calculate the number of states in each cluster and their 'num_drvr_fatl_col' mean and sum.

count_mean_sum = car_acc_miles.groupby('cluster')['num_drvr_fatl_col'].agg(['count',
'mean', 'sum'])

count_mean_sum

# Which cluster would you choose?

cluster_num = 0
```

Reducing Traffic Mortality in the USA

Task 1: Instructions

Explore your current folder and have a look at the main dataset file.

- Check the name of the current folder using `getwd()` .
- List all files in the current folder using `list.files()`.
- List all files in `datasets/` folder using `list.files()` by setting the `path` parameter to the folder.
- View the first 20 lines of `road-accidents.csv` in the `datasets/` folder using `readLines()` and setting the `n` parameter.

Good to know

This project lets you practice the skills from Introduction to the Tidyverse, which includes the pipe operator, summarizing data, and visualizing with ggplot2; from Correlation and Regression, which includes computing correlations and regression coefficients; and from Unsupervised Learning in R, which includes PCA and KMeans clustering. We recommend that you take these courses before starting this project.

Here are three charts illustrating the road accident fatality situation described in the notebook's first paragraph.

Helpful links:

- For this task it is useful to know how to work with files and folders in R
- tidyverse cheat sheet
- ggplot cheat sheet

-

Hint

Does your code look something like this for the second and fourth bullet in the instructions?

```
file_list <- list.files()
accidents_head <- readLines('PATH/TO_FILE', n=20)
```

Task 2: Instructions

Import the main data file and start exploring the data.

- Start by loading the `tidyverse` collection of packages.

- Read in `road-accidents.csv`, which is in the `datasets/` folder, using `read_delim()`.
- Save the number of rows columns with the `dim()` function.
- Generate an overview of the data frame with the useful `str()` function.
- Display the last six rows of the data frame using `tail()`.

Read the documentation of `read_delim()` on how to use the `comment` and `delim` parameters. Is the output of `tail()` what you would expect? A quick data sanity check like this can save us major headaches down the line.

Helpful links:

- readr cheat sheet
- `read_delim()` documentation

-

Hint

Does your code look something like this?

```
library(tidyverse)
car_acc <- read_delim(file = 'PATH_TO/FILE.csv', comment = '<COMMENT>',
delim = '<DELIMITER>')
rows_and_cols <- dim(....)
car_acc_structure <- str(....)
tail(....)
```

Task 3: Instructions

Create a textual and graphical overview of data

- Use `summary()` to create summary statistics of each data frame column.
- Use the pipe operator `%>%` on the `car_acc` data frame to first deselect the `state` column and then use `ggpairs()` (from the `GGally` package) to create a pairwise scatterplot.

Helpful links:

- `summary()` documentation
- `ggpairs()` documentation

-

Hint

Does your plotting code look like this?

```
car_acc %>%
    select(-<COLUMN TO DESELECT>) %>%
    ggpairs()
```

Task 4: Instructions

Explore the correlation between all column-pairs in the data frame

- Using the pipe operator, remove the state column and then compute the correlation coefficient for all column-pairs using `cor()`. By default, the Pearson correlation coefficient will be computed.
- Print the correlation coefficient.

- `cor()` documentation
-

Hint

Does your code look something like this?

```
corr_col <- car_acc %>%
    select(-<COLUMN TO DESELECT>) %>%
    cor()
```

Task 5: Instructions

Fit a multivariate linear regression model using the fatal accident rate as the outcome.

- Use `lm()` to fit a multivariate linear regression model for `drvr_fatl_col_bmiles` as a function of `perc_fatl_speed`, `perc_fatl_alcohol`, and `perc_fatl_1st_time`.
- The fit object contains various types of information. Use the `coef()` function to obtain the model coefficients.

Helpful links:

- `lm()` documentation
- `coef()` documentation
-

Hint

Does your code look something like this?

```
r fit_reg <- lm(drvr_fatl_col_bmiles ~ <EXPLANATORY_VAR_1> +
<EXPLANATORY_VAR_2> + <EXPLANATORY_VAR_3>, data=<DATA_FRAME>) fit_coef <-
coef(fit_reg)
```

Task 6: Instructions

Perform a Principle Component Analysis on the standardized data.

- Center and standardise the three feature columns using `scale()`.
- Perform PCA on standardized features (columns 3, 4, and 5 of `car_acc_standised`) using `princomp()`.
- Add point and line geoms to the ggplot corresponding to the axis labels provided.
- Compute the cumulative proportion of variance explained by applying `cumsum()` to the `pve` vector, and extract and print the variance explained by the first two principal components.

Helpful links:

- `princomp()` documentation
- PCA lecture video of Unsupervised Learning in R
- `scale()` documentation

-

Hint

The `perc_fatl_speed` column can be scaled like this:

```
perc_fatl_speed=scale(perc_fatl_speed)
```

Does your code for instruction bullets 2 and 4 look something like this?

```
pca_fit <- princomp(car_acc_standised[, c(....,....,....)])
cve <- cumsum(pve)
cve_pc2 <- cve[2]
```

Task 7: Instructions

Plot the individual states using the first 2 principle components

- Extract the principle component scores from `pca_fit`.
- Create a data frame of the extracted scores and plot the first two principle components using a point geom from `ggplot()`. Order of the principle component scores does not matter.

-

Hint

The code to calculate `pcomp1` looks like this:

```
pcomp1 <- pca_fit$scores[,1]
```

Does your plotting code look like this?

```
ggplot(aes(x=....,y=....)) + geom_point() +
labs(x="Principle Component 1",
    y="Principle Component 2")
```

Task 8: Instructions

Apply the K-mean method using a range of clusters and plot the within-cluster sum-of-squares.

- Create a vector, `k_vec`, of one to ten. These will be the clusters.
- Initialise a vector of inertias of the same length as `k_vec` filled with NAs by using `rep()`.
- For each k, fit a K-mean model using `kmeans()` with centers set to 'k' and save them in the `mykm` list. Then obtain the within-cluster sum-of-squares from the K-means fit object using `tot.withinss` and save them to `inertias`.
- Finally, plot the within-cluster sum-of-squares against the different numbers of clusters with point and line geoms.

Helpful links:

- kmeans documentation

-

Hint

Does your `k_vec` and `inertias` code look like this?

```
k_vec <- 1:10
inertias <- rep(NA, length(k_vec))
```

Task 9: Instructions

Highlight the clusters of the K-means fit with three clusters in the PCA scatterplot.

- Obtain cluster-ids from the kmeans fit with k=3, using the `cluster` handle.
- Then color the points of the principle component plot according to their cluster number by setting the `col` argument equal to the `cluster_id` vector.

-

Hint

Setting the `col` argument can be done like this: `col=cluster_id`

Task 10: Instructions

Visualise the distribution of speeding, alcohol influence, and precentage of first time accidents in a direct comparison of the clusters.

- Add the `cluster_id` vector to the original data frame.
- Remove the `drvr_fatl_col_bmiles` column and use `gather()` to create a long format of the data frame.
- Visualise the distribution of the three features using `geom_violin()`, set the `fill` aesthetic to show separate violin plots for each cluster.

Though not *necessary*, flipping the coordinates of the plot is provided in the sample code.

Helpful links:

- dplyr's gather() function [documentation]
- tidyr's [cheat sheet]

-

Hint

Adding a column to a data frame can be done like this:

```
data_frame$new_col_name <- col_to_add
```

Task 11: Instructions

Add state-wise information about miles driven to compute total number of fatal accidents and total accidents across clusters

- Use the `left_join` function to join the new data frame miles_driven to `car_acc` by the state variable, then create a new variable

`num_drvr_fatl_col` by multiplying `drvr_fatl_col_bmiles` with `million_miles_annually` and dividing by 1000.
- Creating a summary per cluster of the total number of fatal accidents
- Plot the sum for each cluster using `geom_bar` by setting the `stat` to "identity". Set the `fill` aesthetic to the cluster. You can also drop the legend using `show.legend` argument.

Helpful links:

- dplyr's `left_join()` function documentation
- `left_join()` lecture video from Joing Data with dplyr in R

-

Hint

The calculation for `num_drvr_fatl_col` is as follows:

```
num_drvr_fatl_col = drvr_fatl_col_bmiles*million_miles_annually / 1000
```

Task 12: Instructions

Decide which cluster to focus your resources on.

- Which cluster would you choose: 1, 2, or 3? Assign one of these integers to `cluster_num`.

-

Hint

Assign 1, 2, or 3 to `cluster_num` and think about how to motivate the choice.

Who's Tweeting? Trump or Trudeau?

Task 1: Instructions

Import the tools you'll need from scikit-learn.

- Import `CountVectorizer` and `TfidfVectorizer` from `sklearn.feature_extraction.text`.
- Import `train_test_split` from `sklearn.model_selection`.
- Import `MultinomialNB` from `sklearn.naive_bayes`.
- Import `LinearSVC` from `sklearn.svm`.
- Import `metrics` from `sklearn`.

Good to know

This project lets you apply the skills taught in [Natural Language Processing Fundamentals in Python]().

Having trouble locating where you can import a module from? Take a look at the [scikit-learn API documentation](), which lists all of the important modules.

-

Hint

To import a function or module `foo` from a package `bar` you would write:

```
from bar import foo
```

Task 2: Instructions

Import and prepare your data for machine learning.

- Create a new pandas DataFrame with CSV `datasets/tweets.csv`.
- Create target labels `y` equal to the author column of your DataFrame.
- Use `train_test_split()` with the imported DataFrame's `status` column as your data and the target (`y`). Use `random_state=53` and `test_size=.33`.

Remember to follow conventions of naming your output variables in `train_test_split`.

-

Hint

You can read `path_to/my_data.csv` into a DataFrame named `my_data` like this after importing the pandas library:

```
import pandas as pd
my_data = pd.read_csv("path_to/my_data.csv")
```

To select a column from a pandas dataframe, you can use similar syntax to a Python dictionary. To select column `status` from your dataframe, you would use:

```
my_data['status']
```

Task 3: Instructions

Vectorize the data to train a model.

- Initialize a `CountVectorizer` object called `count_vectorizer` with English stop words removed, a minimum frequency of 0.05, and a maximum frequency of 0.9.
- Create `count_train` and `count_test` variables using `fit_transform` and `transform` respectively.
- Initialize a `TfidfVectorizer` object called `tfidf_vectorizer` with English stop words removed, a minimum frequency of 0.05, and a maximum frequency of 0.9.
- Set up `tfidf_train` and `tfidf_test` variables using `fit_transform` and `transform` with the `tfidf_vectorizer` object.

Having trouble remembering how to run `fit_transform` or `transform`? Take a look at the [CountVectorizer documentation].

-

Hint

Stuck on vectorization?

Remember, first you initialize an object with parameters.

```
count_vectorizer = CountVectorizer(stopwords='english', ...)
```

Then, you call `count_vectorizer.fit_transform` and pass it the training data (`X_train`).

Finally, you call `count_vectorizer.transform` and pass it the test data (`X_test`).

After, do the same with the `tfidf_vectorizer`.

Task 4: Instructions

Train and test a Bayesian models using the TF-IDF vectors and count vectors to see how they perform.

- Create `tfidf_nb`, a Multinomial Naive Bayes Classifier with TfidfVectorizer data.
- Fit the model and save the test data predictions as `tfidf_nb_pred` and the accuracy score as `tfidf_nb_score`.
- Create `count_nb`, a Multinomial Naive Bayes Classifier with CountVectorizer data.
- Fit the model and save the test predictions as `count_nb_pred` and the accuracy score as `count_nb_score`.

You can use the `metrics.accuracy_score` method to return accuracy metrics.

-

Hint

Remember, first, you must initialize your model:

```
count_nb = MultinomialNB()
```

Then, you can use the `fit` method of the model to fit your data by passing in the training data and labels (`X_train, y_train`).

Finally, you can use the `predict` method of the model to predict the labels by passing it the test data (`X_test`).

Task 5: Instructions

Plot confusion matrices using the provided helper function and the built-in `metrics.confusion_matrix` function from scikit-learn.

- Create confusion matrices `tfidf_nb_cm` and `count_nb_cm` using the `metrics.confusion_matrix` function with `y_test`, and `tfidf_nb_pred` and `count_nb_pred` respectively. Labels for both matrices are a list of the names in this order: `['Donald J. Trump', 'Justin Trudeau']`.
- Plot `tfidf_nb_cm` using the `plot_confusion_matrix` function by passing in the confusion matrix, the list of classes in the correct order, and a title for clarity.
- Plot the `count_nb_cm` same as above, making sure to also pass in the parameter `figure=1` so the first plot is not overwritten.

-

Hint

The `metrics.confusion_matrix` function takes two arguments: `y_test` and your predicted labels. You can also pass a keyword argument labels, which should be a list of the labels, like so:

```
cm = metrics.confusion_matrix(y_test, my_predictions, labels=['LabelOne',
'LabelTwo'])
```

Task 6: Instructions

Create, train, and test a LinearSVC model to see how it compares to the Bayesian model.

- Create `tfidf_svc`, a Linear Support Vector Classifier with TfidfVectorizer data.
- Fit the model and save the test data predictions as `tfidf_svc_pred` and the accuracy score as `tfidf_svc_score`.
- Create a confustion matrix, `svc_cm`, with the `metrics.confusion_matrix` function, `y_test`, and `tfidf_svc_pred`. Again, the labels need to be in order.
- Plot the confusion matrix and pass in the classes as a list in the correct order and title for clarity.

To get the accuracy score, use the predicted labels with `y_test` labels by passing both to the `metrics.accuracy_score` function. This function will test each prediction and give you an overall accuracy of the predicted labels.

-

Hint

Remember, first, you must initialize your model:

```
tfidf_svc = LinearSVC()
```

Then, you can use the `fit` method of the model to fit your data by passing in the training data and labels (`X_train, y_train`).

Finally, you can use the `predict` method of the model to predict the labels by passing it the test data (`X_test`).

Task 7: Instructions

Plot the features from most Trump-like to most Trudeau-like using `plot_and_return_top_features`.

- Import `pprint` from module `pprint`.
- Use `plot_and_return_top_features` and save the output as `top_features`.

- Print `top_features` to see the tokens and their weights. Analyze the resulting graph. What tokens are most Trump-like? Most Trudeau-like? Do you notice anything that we could have caught in preprocessing?

Make sure to read the docstring documentation to know what order to pass in the classifier and vectorizer. Remember to use the top performing model!

-

Hint

The vectorizer for your top model is the `tfidf_vectorizer` and the classifier is your top model (the LinearSVC model).

Task 8: Instructions

Create one tweet to classify as Trump and one tweet to classify as Trudeau. Test them with the model.

- Write a tweet you think will be classified as Trump and save it as `trump_tweet`.
- Write a tweet you think will be classified as Trudeau and save it as `trudeau_tweet`.
- Using `tfidf_vectorizer`, transform the two tweets you created and save the transformed tweets as `trump_tweet_vectorized` and `trudeau_tweet_vectorized`. Remember, the vectorizer expects a list of strings, so make sure to put your tweet inside a list.
- Using the `tfidf_svc` model, predict the label for each vectorized tweet and save the predictions as `trump_tweet_pred` and `trudeau_tweet_pred`.

Remember to use a list when giving the string of the tweet to the vectorizer `transform` method.

-

Hint

You can vectorize the tweet by using the `tfidf_vectorizer` and calling the `transform` method with a list. For example, if you wnated to transform the tweet `my_tweet` and save it as `tweet_vectorized`, you could use the following:

```
tweet_vectorized = tfidf_vectorizer.transform([my_tweet])
```

Then, you can call predict on a model `my_model` using the vectorized tweet like so:

```
tweet_pred = my_model.predict(tweet_vectorized)
```

```
# Set seed for reproducibility
import random; random.seed(53)

# Import all we need from sklearn
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn import metrics

import pandas as pd

# Load data
tweet_df = pd.read_csv('datasets/tweets.csv')

# Create target
y = tweet_df.author

# Split training and testing data
X_train, X_test, y_train, y_test = train_test_split(tweet_df['status'], y,
test_size=0.33,
                 random_state=53)

# Initialize count vectorizer
count_vectorizer = CountVectorizer(stop_words='english',
                                   min_df=0.05, max_df=0.9)

# Create count train and test variables
count_train = count_vectorizer.fit_transform(X_train)
count_test = count_vectorizer.transform(X_test)

# Initialize tfidf vectorizer
tfidf_vectorizer = TfidfVectorizer(stop_words='english',
                                   min_df=0.05, max_df=0.9)

# Create tfidf train and test variables
tfidf_train = tfidf_vectorizer.fit_transform(X_train)
tfidf_test = tfidf_vectorizer.transform(X_test)

tfidf_nb = MultinomialNB()
tfidf_nb.fit(tfidf_train, y_train)
tfidf_nb_pred = tfidf_nb.predict(tfidf_test)
tfidf_nb_score = metrics.accuracy_score(y_test, tfidf_nb_pred)

count_nb = MultinomialNB()
count_nb.fit(count_train, y_train)
count_nb_pred = count_nb.predict(count_test)
count_nb_score = metrics.accuracy_score(y_test, count_nb_pred)

print('NaiveBayes Tfidf Score: ', tfidf_nb_score)
print('NaiveBayes Count Score: ', count_nb_score)

%matplotlib inline
from datasets.helper_functions import plot_confusion_matrix


tfidf_nb_cm = metrics.confusion_matrix(y_test, tfidf_nb_pred,
labels=['Donald J. Trump', 'Justin Trudeau'])
count_nb_cm = metrics.confusion_matrix(y_test, count_nb_pred,
labels=['Donald J. Trump', 'Justin Trudeau'])
```

```
plot_confusion_matrix(tfidf_nb_cm, classes=['Donald J. Trump', 'Justin
Trudeau'], title="TF-IDF NB Confusion Matrix")

plot_confusion_matrix(count_nb_cm, classes=['Donald J. Trump', 'Justin
Trudeau'], title="Count NB Confusion Matrix", figure=1)

tfidf_svc = LinearSVC()
tfidf_svc.fit(tfidf_train, y_train)
tfidf_svc_pred = tfidf_svc.predict(tfidf_test)
tfidf_svc_score = metrics.accuracy_score(y_test, tfidf_svc_pred)

print("LinearSVC Score:   %0.3f" % tfidf_svc_score)

svc_cm = metrics.confusion_matrix(y_test, tfidf_svc_pred, labels=['Donald
J. Trump', 'Justin Trudeau'])
plot_confusion_matrix(svc_cm, classes=['Donald J. Trump', 'Justin
Trudeau'], title="TF-IDF LinearSVC Confusion Matrix")

from datasets.helper_functions import plot_and_return_top_features

from pprint import pprint
top_features = plot_and_return_top_features(tfidf_svc, tfidf_vectorizer)
pprint(top_features)

trump_tweet = "America is great!"
trudeau_tweet = "Canada les"

trump_tweet_vectorized = tfidf_vectorizer.transform([trump_tweet])
trudeau_tweet_vectorized = tfidf_vectorizer.transform([trudeau_tweet])

trump_tweet_pred = tfidf_svc.predict(trump_tweet_vectorized)
trudeau_tweet_pred = tfidf_svc.predict(trudeau_tweet_vectorized)

print("Predicted Trump tweet", trump_tweet_pred)
print("Predicted Trudeau tweet", trudeau_tweet_pred)
```

Who Is Drunk and When in Ames, Iowa?

Task 1: Instructions

First, get the data into your workspace and summarize it by year. Do you notice a pattern over time?

- Import pandas aliased as `pd`.
- Read `datasets/breath_alcohol_ames.csv` into your workspace using the `read_csv()` function. Save it as `ba_data`.
- Look at the format of your dataset using the `head()` function.
- Count how many tests were administered in each year using the `value_counts()` function. Assign the results to `ba_year`.

Good to know

This Project lets you practice the skills from Data Manipulation with pandas, including reading, exploring, filtering, and grouping data. We recommend that you take this course before starting this Project.

Helpful links:

- pandas cheat sheet
- pandas `read_csv()` function documentation
- Reading a CSV exercise in our Data Manipulation with pandas course
- For even more, visit the `pandas` documentation.

If you're stuck, check the hint below. ↓

-

Solution

```
# import pandas
import pandas as pd

# read the data into your workspace
ba_data = pd.read_csv("datasets/breath_alcohol_ames.csv")

# quickly inspect the data
print(ba_data.head())

# obtain counts for each year
ba_year = ba_data['year'].value_counts()
ba_year

# Alternative method:
```

```
# ba_data.set_index(["year", "month", "day", "hour", "location",
"gender"]).count(level="year")
```

Task 2: Instructions

Count the data by location to see which department administers more breathalyzer tests.

- Group and summarize the data by location with the `vaue_counts` function. Save this data as `pds`.

-

Solution

```
# use value_counts to tally up the totals for each department
pds = ba_data['location'].value_counts()
pds

# Alternative method:
# ba_data.set_index(["year", "location", "gender"]).count(level="location")
```

Task 3: Instructions

Summarize the data and create a bar chart of the number of tests by hour of the day.

- Using `groupby()` and `size()`, summarize the data by hour. Save this as `hourly`.
- Create a bar chart of total breath alcohol tests by hour of the day using `plot.bar` on the `hourly` Series.

The `groupby()` function groups a data frame according to a variable. `size()` obtains the counts per group. Here's a helpful Stack Overflow answer describing how to use both together.

The `plot.bar` method can also be written as `plot(kind='bar')`. For more information about data visualization with pandas check out the documentation.

-

Solution

```
%matplotlib inline

# count by hour and arrange by descending frequency
# hourly = ba_data.groupby(['hour'])
['location'].count().sort_values(ascending=False)
hourly = ba_data.groupby(['hour']).size()
```

```
print(len(hourly))

hourly.plot.bar(x='hour') # TODO this plot is not in decending frequency
```

Task 4: Instructions

We'll look at the month variable to determine the most popular time of year for breathalyzer tests.

- Using `groupby()` and `size()`, summarize the data by month. Save this as `monthly`.
- Create a bar chart of total breath alcohol tests by month using `plot.bar` on the `monthly` Series.

The values of the month variable are (1, 2, …, 12) and correspond to the months (January, February, …, December).

-

Solution

```
# count by month and arrange by descending frequency
# monthly = ba_data.groupby(['month'])
['location'].count().sort_values(ascending=False)
monthly = ba_data.groupby(['month']).size()
print(len(monthly))

# use plot.bar to make the appropriate bar chart
monthly.plot.bar(x='month')
```

Task 5: Instructions

Compare test frequency and results for men vs. women.

- Count the number of tests by gender using `value_counts()` to see which gender took more tests.
- Remove the NA values in the gender variable with `dropna()` and save the results as `gen`.
- In `gen`, use the `assign()` method to create a new variable called `meanRes`, the mean of the two tests `Res1` and `Res2`. Assign the resulting DataFrame to `mean_bas`.
- Using `mean_bas`, create boxplots of mean results for men and women. Use `meanRes` on the y-axis and `gender` on the x-axis.

The function `dropna()` will drop the rows where at least one element is missing.

The box plots will need to have the variable of interest selected using `by='var'`.

-

Solution

```
# count by gender (may not be relevent in this work flow)
counts_gender = ba_data['gender'].value_counts()

# create a dataset with no NAs in gender
gen = ba_data.dropna(subset=['gender'])

# create a mean test result variable
mean_bas = gen.assign(meanRes=(gen.Res1+gen.Res2)/2)
# gen['meanRes'] = (gen.Res1+gen.Res2)/2

# create side-by-side boxplots to compare the mean blood alcohol levels of
men and women
mean_bas.boxplot(['meanRes'], by = 'gender')
```

Task 6: Instructions

Determine what percent of the breathalyzer tests in the data are above the legal limit.

- Filter the `ba_data` to include only tests where one or both of `Res1`, `Res2` are greater than 0.08. Call this filtered data `duis`.
- Create a variable, `p_dui`, for the proportion of all tests that would have resulted in a DUI.

To calculate `p_dui`, use the `shape()` attribute of the `duis` and `ba_data` DataFrames to obtain the number of rows of each.

-

Solution

```
# Filter the data
duis = ba_data[(ba_data.Res1 > 0.08) | (ba_data.Res2 > 0.08)]

# proportion of tests that would have resulted in a DUI
p_dui = duis.shape[0] / ba_data.shape[0]
p_dui
```

Task 7: Instructions

Create a date variable and determine the week of the year each test occurred.

- Create a new column in `ba_data` called `date` by using `to_datetime()` to combine the date variables in the order of year, month, day.
- Using the new `date` variable, create another new column in `ba_data` called `week` using `dt.week`.

To learn more about working with dates and times using pandas, see the `to_datetime()` documentation.

-

Solution

```
# Create date variable
# ba_data = ba_data.assign(date = pd.to_datetime(ba_data[['year', 'month',
'day']]))
ba_data['date'] = pd.to_datetime(ba_data[['year', 'month', 'day']])

# Create a week variable
# ba_data = ba_data.assign(week= ba_data['date'].dt.week)
ba_data['week'] = ba_data['date'].dt.week

# Check your work
ba_data.head()
```

Task 8: Instructions

Create a time series plot to compare weeks across years.

- Group `ba_data` by the `week` and `year` columns and `count()` the number arrests per week in each year. Assign the result to `timeline`.
- Turn index values of `timeline` into column names using `unstack()`, then create a line plot using `plot()`. `week` should be the x-axis and the count variable on the y-axis, colored by `year`.

We need to reorganize the data for it to plot well. `unstack()` is used here to move the innermost row index (`year`) to become the innermost column index.

The legend is set to `True` to display the year.

-

Solution

```
# create the weekly data set (most similar to original project)
# weekly1 = ba_data[['year', 'week', 'gender']]
# weekly1.groupby(['week','year']).count().unstack().plot()

# choose the variables of interest, count
timeline = ba_data.groupby(['week','year']).count()['Res1']

# unstack and plot
timeline.unstack().plot(title='VEISHEA DUIs', legend=True)
print(len(timeline))
```

Task 9: Instructions

In your opinion, `True` or `False`: canceling VEISHEA was the right decision based on the amount of breathalyzer tests alone. The last two VEISHEA weeks in Iowa State's history were the 16th week in 2013 and the 15th week in 2014.

- Inspect the plot produced in task 8.
- Assign `True` or `False` to `canceling_VEISHEA_was_right`.

Congratulations on making it to the end of the project! If you haven't checked your project yet, you can do so by clicking the yellow "Check Project" button.

Good luck! :)

-

Solution

```
# ## Run this code to create the plot
# ba_data.groupby(['week','year']).count()
['Res1'].unstack().plot(legend=True)

# veishea= ba_data.loc[(ba_data.week < 20) & (ba_data.week > 10)]

# veishea.groupby(['week','year']).count()
['Res1'].unstack().plot(title='VEISHEA Weeks',legend=True)

## Was it right to permanently cancel VEISHEA? TRUE or FALSE?
canceling_VEISHEA_was_right = False
```

Where Would You Open a Chipotle?

Task 1: Instructions

Load and explore Thinknum's Chipotle data.

- Load the `tidyverse`, `leaflet`, and `leaflet.extras` packages.
- Read `datasets/chipotle.csv` into a tibble named `chipotle` using `read_csv`.
- Print out the `chipotle` tibble using the `head` function.

Good to know

This notebook focuses on the application of the skills acquired in [Interactive Maps with leaflet in R](), including using different map tiles, creating color palettes, and conducting an exploratory data analysis using Leaflet. Skills from [Introduction to the Tidyverse]() are used as well. It is recommended that those Courses are completed before starting this Project.

Helpful links for the Project:

- Leaflet for R website
- The Leaflet package documentation

Helpful links for Task 1:

- The readr package documentation
- RStudio's data import cheatsheet

-

Solution

```
# Load tidyverse, leaflet, and leaflet.extras
library(tidyverse)
library(leaflet)
library(leaflet.extras)
library(sf)

# Read datasets/chipotle.csv into a tibble named chipotle using read_csv
chipotle <- read_csv("datasets/chipotle.csv")

# Print out the chipotle tibble using the head function
head(chipotle)
```

Task 2: Instructions

Create a leaflet map of all closed Chipotle restaurants.

- Filter the `chipotle` tibble to stores with a value of `TRUE` for `closed`.
- Use `addTiles` to add the default Open Street Map tile to the map.
- Plot the closed stores using `addCircles`.

Helpful links:

- The filter function's page on the tidyverse website
- The markers section on RStudio's Leaflet website
- The mapping California's colleges exercise in DataCamp's Interactive Maps with leaflet in R Course

-

Solution

```
# Create a leaflet map of all closed Chipotle stores
closed_chipotles <-
  chipotle %>%
  # Filter the chipotle tibble to stores with a value of t for closed
  filter(closed == TRUE) %>%
  leaflet() %>%
  # Use addTiles to plot the closed stores on the default Open Street Map
tile
  addTiles() %>%
  # Plot the closed stores using addCircles
  addCircles()

# Print map of closed chipotles
closed_chip
```

Task 3: Instructions

Find out how many Chipotles have closed (hint: not many!) and remove closed stores from the data.

- Use `count` from `dplyr` to count the values for the `closed` variable.
- Create a new tibble named `chipotle_open` that contains only open Chipotle stores.
- Drop the `closed` column from `chipotle_open` using `select`.

Helpful links:

- The tidyverse pages for select, count, and tibble functions

-

Solution

```
# Use count from dplyr to count the values for the closed variable
chipotle %>%
  count(closed)

# Create a new tibble named chipotle_open that contains only open chipotle
chipotle_open <-
  chipotle %>%
  filter(closed == FALSE) %>%
  # Drop the closed column from chipotle_open
  select(-closed)
```

Task 4: Instructions

Create a heatmap of all open Chipotles.

- Pipe `chipotle_open` into a chain of `leaflet` functions.
- Use `addProviderTiles` to add the `CartoDB` provider tile.
- Use `addHeatmap` with a `radius` of 8.

Helpful links:

- The map tiles section in DataCamp's Interactive Maps with leaflet in R Course
- Example Leaflet heatmaps
- The heatmap function page in the `leaflet.extras` GitHub repo

-

Solution

```
# Pipe chipotle_open into a chain of leaflet functions
chipotle_heatmap <-
chipotle_open %>%
  leaflet() %>%
  # Use addProviderTiles to add the CartoDB provider tile
  addProviderTiles("CartoDB") %>%
  # Use addHeatmap with a radius of 8
  addHeatmap(radius = 8)

# Print heatmap
chipotle_heatmap
```

Task 5: Instructions

Count the number of open Chipotle stores in each state.

- Create a new tibble called `chipotles_by_state` to store the results.
- Filter the data to only Chipotles in the United States.
- Count the number of stores in `chipotle_open` by the `st` variable.
- Arrange the number of stores by state in ascending order.

Helpful links:

- The tidyverse pages for tibble, count, and arrange functions
-

Solution

```
# Create a new tibble called chipotles_by_state to store the results
chipotles_by_state <-
  chipotle_open %>%
  # Filter the data to only Chipotles in the United States
  filter(ctry == "United States") %>%
  # Count the number of stores in chipotle_open by st
  count(st) %>%
  # Arrange the number of stores by state in ascending order
  arrange(n)

# Print the state counts
chipotles_by_state
```

Task 6: Instructions

Find the states without a Chipotle.

- Print the state.abb vector.
- Use the `%in%` operator to determine which states are in `chipotles_by_state`.
- Use the `%in%` and `!` operators to determine which states are not in `chipotles_by_state`.
- Print the states that are not in `chipotles_by_state`.

Helpful links:

- The `%in%` operator's documentation
- Testing if the values of one vector are `%in%` another returns a logical vector with TRUE and FALSE values. We can return the opposite of this vector to indicate the values that are not `%in%` the vector by using the not operator at the start of the statement(i.e., `!`).
-

Solution

```
# Print the state.abb vector
state.abb

# Use the %in% operator to determine which states are in chipotles_by_state
```

```
state.abb %in% chipotles_by_state$st

# Use the %in% and ! operators to determine which states are not in
chipotles_by_state
!state.abb %in% chipotles_by_state$st

# Create a states_wo_chipotles vector
states_wo_chipotles <- state.abb[!state.abb %in% chipotles_by_state$st]

# Print states with no Chipotles
states_wo_chipotles
```

Task 7: Instructions

Create county-level map of South Dakota population.

- Load `south_dakota_pop.rds` into an object called `south_dakota_pop`.
- Create color palette to color the map by county population `estimate`.
- Add county boundaries with `addPolygons` and color by population `estimate`.
- Add a legend using `addLegend`.

Helpful links:

- The colors section on RStudio's Leaflet website
- The mapping polygons section in DataCamp's Interactive Maps with leaflet in R Course
- The lines and shapes section on RStudio's Leaflet website

-

Solution

```
# Load south_dakota_pop.rds into an object called south_dakota_pop
south_dakota_pop <- readRDS("datasets/south_dakota_pop.rds")

# Create color palette to color map by county population estimate
pal <- colorNumeric(palette = "viridis", domain =
range(south_dakota_pop$estimate))

sd_pop_map <-
  south_dakota_pop %>%
  leaflet() %>%
  addProviderTiles("CartoDB") %>%
  # Add county boundaries with addPolygons and color by population estimate
  addPolygons(stroke = FALSE, fillOpacity = 0.7, color = ~ pal(estimate),
label = ~NAME) %>%
  # Add a legend using addLegend
  addLegend(pal = pal, values = ~estimate, title = "Population")

# Print map of South Dakota population by county
sd_pop_map
```

Task 8: Instructions

Create a data frame with proposed South Dakota store locations and Chipotles in surrounding states.

- Load the `chipotle_sd_locations.csv` file that contains proposed South Dakota locations.
- Filter the open Chipotle store data to include locations in states bordering South Dakota only.
- Bind the data on proposed South Dakota locations onto the open store data using `bind_rows`.

Helpful links:

- The tidyverse pages for readcsv, select, filter, mutate, and bindrows
-

Solution

```
# Load chipotle_sd_locations.csv that contains proposed South Dakota
locations
chipotle_sd_locations <- read_csv("datasets/chipotle_sd_locations.csv")

# limit chipotle store data to locations in states boardering South Dakota
chipotle_market_research <-
  chipotle_open %>%
  filter(st %in% c("IA", "MN", "MT", "ND", "NE", "WY")) %>%
  select(city, st, lat, lon) %>%
  mutate(status = "open") %>%
  # bind the data on proposed SD locations onto the open store data
  bind_rows(chipotle_sd_locations)

# print the market research data
chipotle_market_research
```

Task 9: Instructions

Determine how many existing stores are within 100 miles of the proposed locations.

- Create a blue and red color palette to distinguish between open and proposed stores using `colorFactor`.
- Map the open and proposed locations.
- Add the stamen toner provider tile.
- Apply the pal color palette.
- Draw a circle with a 100 mi radius around the proposed locations.

Helpful links:

- The colors and the shapes sections on RStudio's Leaflet website
- The Creating a Color Palette using colorFactor exercise in DataCamp's Interactive Maps with leaflet in R Course
- The `radius` argument of `addCircles` takes a numeric vector of radii for the circles in meters. There are ~1609.34 meters in a mile.

-

Solution

```
# Create a blue and red color palette to distinguish between open and
proposed stores
pal <- colorFactor(palette = c("Blue", "Red"), domain = c("open",
"proposed"))

# Map the open and proposed locations
sd_proposed_map <-
  chipotle_market_research %>%
  leaflet() %>%
  # Add the Stamen Toner provider tile
  addProviderTiles("Stamen.Toner") %>%
  # Apply the pal color palette
  addCircles(color = ~pal(status)) %>%
  # Draw a circle with a 100 mi radius around the proposed locations
  addCircles(data = chipotle_sd_locations, radius = 100 * 1609.34, color =
~pal(status), fill = FALSE)

# Print the map of proposed locations
sd_proposed_map
```

Task 10: Instructions

Plot Voronoi polygons to visualize the area covered by each existing and proposed Chipotle.

- Load the Voronoi polygon data.
- Use the `CartoDB` provider tile.
- Plot Voronoi polygons using `addPolygons`.
- Add proposed and open locations as another layer.

Helpful links:

- More about Voronoi or Thiessen polygons in this video
- How to create Voronoi polygons in R using the dismo package

-

Solution

```
# load the Voronoi polygon data
```

```
polys <- readRDS("datasets/voronoi_polygons.rds")

voronoi_map <-
  polys %>%
  leaflet() %>%
  # Use the CartoDB provider tile
  addProviderTiles("CartoDB") %>%
  # Plot Voronoi polygons using addPolygons
  addPolygons(fillColor = ~pal(status), weight = 0.5, color = "black") %>%
  # Add proposed and open locations as another layer
  addCircleMarkers(data = chipotle_market_research, label = ~city, color =
~pal(status))

# Print the Voronoi map
voronoi_map
```

Task 11: Instructions

Determine where the next Chipotle should be located.

- Create the `next_chipotle` tibble with one `FALSE` value and one `TRUE` value for the `open_new_store` variable.

You have reached the end of the Project. Congratulations! Remember to review the notebook with the "Check Project" button when you are satisfied with your code.

-

Solution

```
# Where should the next Chipotle store be?
next_chipotle <- tibble(location = c("Rapid City, SD", "Sioux Falls, SD"),
                        open_new_store = c(TRUE, FALSE))
```

Do Left-handed People Really Die Young?

Task 1: Instructions

Load the handedness data from the National Geographic survey and create a scatter plot.

- Import `pandas` as `pd` and `matplotlib.pyplot` as `plt`.
- Load the data into a pandas DataFrame named `lefthanded_data` using the provided `data_url_1`. Note that the file is a CSV file.
- Use the `.plot()` method to create a plot of the "Male" and "Female" columns vs. "Age".

Good to know

This project lets you apply the skills from Data Manipulation with pandas, Statistical Thinking in Python, Introduction to Data Visualization with Python, and concepts from Foundations of Probability in R and Fundamentals of Bayesian Data Analysis in R, including reading data, creating new columns, creating matplotlib visualizations, using numpy arrays, constructing probabilities from frequency data, and Bayes' theorem. We recommend that you take those courses before starting this project.

Helpful links:

- pandas cheat sheet
- pandas `read_csv()` function documentation
- numpy tutorial
- matplotlib cheat sheet
- short intro to Python functions
- Bayes' theorem on Wikipedia

-

Solution

```
# import libraries
import pandas as pd
import matplotlib.pyplot as plt

# load the data
data_url_1 =
"https://gist.githubusercontent.com/mbonsma/8da0990b71ba9a09f7de395574e54df
1/raw/aec88b30af87fad8d45da7e774223f91dad09e88/lh_data.csv"
lefthanded_data = pd.read_csv(data_url_1, sep = ',')

# plot male and female left-handedness rates vs. age
%matplotlib inline
fig, ax = plt.subplots() # create figure and axis objects
```

```
ax.plot('Age', 'Female', data = lefthanded_data, marker='o') # plot
"Female" vs. "Age"
ax.plot('Age', 'Male', data = lefthanded_data, marker = 'x') # plot "Male"
vs. "Age"
ax.legend() # add a legend
ax.set_xlabel("Age")
ax.set_ylabel("Percentage of people who are left-handed")
```

Task 2: Instructions

Add two new columns, one for birth year and one for mean left-handedness, then plot the mean as a function of birth year.

- Create a column in `lefthanded_data` called `Birth_year`, which is equal to 1986 - `Age` (since the study was done in 1986).
- Create a column in `lefthanded_data` called `Mean_lh` which is equal to the mean of the `Male` and `Female` columns.
- Use the `.plot()` method to plot `Mean_lh` vs. `Birth_year`.

Helpful links:

- Creating a column in `pandas`
- matplotlib cheat sheet

-

Solution

```
# create a new column for birth year of each age
lefthanded_data['Birth_year'] = 1986 - lefthanded_data['Age'] # the study
was done in 1986

# create a new column for the average of male and female
lefthanded_data['Mean_lh'] = lefthanded_data[['Female', 'Male']].mean(axis
= 1)

# create a plot of 'Mean_lh' vs. 'Birth_year'
fig, ax = plt.subplots()
ax.plot('Birth_year', 'Mean_lh', data = lefthanded_data)
ax.set_xlabel("Year of birth")
ax.set_ylabel("Percentage left-handed")
```

Task 3: Instructions

Create a function that will return P(LH | A) for particular ages of death in a given study year.

- Import the `numpy` package aliased as `np`.
- Use the last ten `Mean_lh` data points to get an average rate for the early 1900s. Name the resulting DataFrame `early_1900s_rate`.

- Use the first ten `Mean_lh` data points to get an average rate for the late 1900s. Name the resulting DataFrame `late_1900s_rate`.
- For the early 1900s ages, fill in `P_return` with the appropriate left-handedness rates for `ages_of_death`. That is, input `early_1900s_rate` as a fraction, i.e., divide by 100.
- For the late 1900s ages, fill in `P_return` with the appropriate left-handedness rates for `ages_of_death`. That is, input `late_1900s_rate` as a fraction, i.e., divide by 100.

When calculating `early_1900s_rate` and `late_1900s_rate`, remember that because the original data was from youngest age to oldest age, that means that the data is organized from latest birth year to earliest birth year. You will use the first ten `Mean_lh` data points to get an average rate for the late 1900s and the last ten for the early 1900s.

Helpful links:

- Python functions
- Conditional probability
- numpy array subsetting and indexing
- numpy `logical_and()` documentation

-

Solution

```
import numpy as np

# create a function for P(LH | A)
def P_lh_given_A(ages_of_death, study_year = 1990):
    """ P(Left-handed | age of death), calculated based on the reported
rates of left-handedness.
    Inputs: age of death, study_year
    Returns: probability of left-handedness given that a subject died in
`study_year` at age `age_of_death` """

    # Use the mean of the 10 neighbouring points for rates before and after
the start
    early_1900s_rate = lefthanded_data['Mean_lh'][-10:].mean()
    late_1900s_rate = lefthanded_data['Mean_lh'][:10].mean()
    middle_rates =
lefthanded_data.loc[lefthanded_data['Birth_year'].isin(study_year -
ages_of_death)]['Mean_lh']

    youngest_age = study_year - 1986 + 10 # the youngest age in the NatGeo
dataset is 10
    oldest_age = study_year - 1986 + 86 # the oldest age in the NatGeo
dataset is 86

    P_return = np.zeros(ages_of_death.shape)  # create an empty array to
store the results
    # extract rate of left-handedness for people of age age_of_death
    P_return[ages_of_death > oldest_age] = early_1900s_rate / 100
    P_return[ages_of_death < youngest_age] = late_1900s_rate / 100
```

```
    P_return[np.logical_and((ages_of_death <= oldest_age), (ages_of_death
>= youngest_age))] = middle_rates / 100

    return P_return
```

Task 4: Instructions

Load death distribution data for the United States and plot it.

- Load death distribution data in the provided `data_url_2` into `death_distribution_data`, setting `sep = '\t'` and `skiprows=[1]` to account for the dataset's format.
- Drop the NaN values from the `Both Sexes` column.
- Use the `.plot()` method to plot the number of people who died as a function of their age.

Helpful links:

- pandas `read_csv()` function documentation
- pandas `dropna()` documentation
- matplotlib cheat sheet

-

Solution

```
# Death distribution data for the United States in 1999
data_url_2 =
"https://gist.githubusercontent.com/mbonsma/2f4076aab6820ca1807f4e29f75f18e
c/raw/62f3ec07514c7e31f5979beeca86f19991540796/cdc_vs00199_table310.tsv"

# load death distribution data
death_distribution_data = pd.read_csv(data_url_2, sep = '\t', skiprows=[1])

# drop NaN values from the `Both Sexes` column
death_distribution_data = death_distribution_data.dropna(subset = ["Both
Sexes"]) # drop NaN from'Both Sexes' column

# plot number of people who died as a function of age
fig, ax = plt.subplots()
ax.plot('Age', 'Both Sexes', data = death_distribution_data, marker='o')
ax.set_xlabel("Age")
ax.set_ylabel("Number of people who died")
```

Task 5: Instructions

Create a function called `P_lh()` which calculates the overall probability of left-handedness in the population for a given study year.

- Create a series, `p_list`, by multiplying the number of dead people in the `Both Sexes` column with the probability of their being lefthanded using `P_lh_given_A()`.
- Set the variable `p` equal to the sum of that series.
- Divide `p` by the total number of dead people by summing `death_distribution_data` over the `Both Sexes` column. Return result from the function.

P(LH | A) was defined in Task 3. N(A) is the value of `Both Sexes` in the `death_distribution_data` DataFrame where the `Age` column is equal to A. The denominator is total number of dead people, which you can get by summing over the entire dataframe in the `Both Sexes` column.

Helpful links:

- Python functions
-

Solution

```
def P_lh(death_distribution_data, study_year = 1990): # sum over P_lh for
each age group
    """ Overall probability of being left-handed if you died in the study
year
    P_lh = P(LH | Age of death) P(Age of death) + P(LH | not A) P(not A) =
sum over ages
    Input: dataframe of death distribution data
    Output: P(LH), a single floating point number """
    p_list = death_distribution_data['Both
Sexes']*P_lh_given_A(death_distribution_data['Age'], study_year)
    p = np.sum(p_list)
    return p/np.sum(death_distribution_data['Both Sexes']) # normalize to
total number of people in distribution

print(P_lh(death_distribution_data))
```

Task 6: Instructions

Write a function to calculate `P_A_given_lh()`.

- Calculate `P_A`, the overall probability of dying at age A, which is given by `death_distribution_data` at age A divided by the total number of dead people (the sum of the `Both Sexes` column of `death_distribution_data`).
- Calculate the overall probability of left-handedness P(LH) using the function defined in Task 5.
- Calculate P(LH | A) using the function defined in Task 3.

Helpful links:

- Python functions
- Bayes' theorem

-

Solution

```
def P_A_given_lh(ages_of_death, death_distribution_data, study_year =
1990):
    """ The overall probability of being a particular `age_of_death` given
that you're left-handed """
    P_A = death_distribution_data['Both Sexes'][ages_of_death] /
np.sum(death_distribution_data['Both Sexes'])
    P_left = P_lh(death_distribution_data, study_year) # use P_lh function
to get probability of left-handedness overall
    P_lh_A = P_lh_given_A(ages_of_death, study_year) # use P_lh_given_A to
get probability of left-handedness for a certain age
    return P_lh_A*P_A/P_left
```

Task 7: Instructions

Write a function to calculate `P_A_given_rh()`.

- Calculate `P_A`, the overall probability of dying at age A, which is given by `death_distribution_data` at age A divided by the total number of dead people. (This value is the same as in task 6.)
- Calculate the overall probability of right-handedness P(RH), which is 1-P(LH).
- Calculate P(RH | A), which is 1 - P(LH | A).

Helpful links:

- Python functions
- Bayes' theorem

-

Solution

```
def P_A_given_rh(ages_of_death, death_distribution_data, study_year =
1990):
    """ The overall probability of being a particular `age_of_death` given
that you're right-handed """
    P_A = death_distribution_data['Both Sexes'][ages_of_death] /
np.sum(death_distribution_data['Both Sexes'])
    P_right = 1- P_lh(death_distribution_data, study_year) # either you're
left-handed or right-handed, so these sum to 1
    P_rh_A = 1-P_lh_given_A(ages_of_death, study_year) # these also sum to
1
```

```
    return P_rh_A*P_A/P_right
```

Task 8: Instructions

Plot the probability of being a certain age at death given that you're left- or right-handed for a range of ages.

- Calculate `P_A_given_lh` and `P_A_given_rh` using the functions defined in Task 6.
- Use the `.plot()` method to plot the results versus age.

-

Solution

```
ages = np.arange(6,115,1) # make a list of ages of death to plot

# for each age, calculate the probability of being left- or right-handed
left_handed_probability = P_A_given_lh(ages, death_distribution_data)
right_handed_probability = P_A_given_rh(ages, death_distribution_data)

fig, ax = plt.subplots() # create figure and axis objects
ax.plot(ages, left_handed_probability, label = "Left-handed")
ax.plot(ages, right_handed_probability, label = "Right-handed")
ax.legend()
ax.set_xlabel("Age at death")
ax.set_ylabel(r"Probability of being age A at death")
```

Task 9: Instructions

Find the mean age at death for left-handers and right-handers.

- Multiply the `ages` list by the left-handed probabilities of being those ages at death, then use `np.nansum` to calculate the sum. Assign the result to `average_lh_age`.
- Do the same with the right-handed probabilities to calculate `average_rh_age`.
- Print `average_lh_age` and `average_rh_age`.
- Calculate the difference between the two average ages and print it.

To make your printed output prettier, try using the `round()` function to round your results to two decimal places.

Helpful links:

- `np.nansum()` documentation
- `round()` documentation

-

Solution

```
# calculate average ages for left-handed and right-handed groups
# use np.array so that two arrays can be multiplied
average_lh_age =  np.nansum(ages*np.array(left_handed_probability))
average_rh_age =  np.nansum(ages*np.array(right_handed_probability))

# print the average ages for each group
print(round(average_lh_age,1))
print(round(average_rh_age,1))

# print the difference between the average ages
print("The difference in average ages is " + str(round(average_rh_age -
average_lh_age, 1)) + " years.")
```

Task 10: Instructions

Redo the calculation from Task 8, setting the `study_year` parameter to 2018.

- In the call to `P_A_given_lh`, set `age_of_death` to `ages`, `death_distribution_data` to `death_distribution_data`, and `study_year` to 2018.
- Do the same for `P_A_given_rh`.

You can read more about this interpretation of the study in this [BBC article](#).

-

Solution

```
# loop through ages, calculating the probability of being left- or right-
handed
left_handed_probability_2018 = P_A_given_lh(ages, death_distribution_data,
study_year = 2018)
right_handed_probability_2018 = P_A_given_rh(ages, death_distribution_data,
study_year = 2018)

# calculate average ages for left-handed and right-handed groups
average_lh_age_2018 =
np.nansum(ages*np.array(left_handed_probability_2018))
average_rh_age_2018 =
np.nansum(ages*np.array(right_handed_probability_2018))

print("The difference in average ages is " +
      str(round(average_rh_age_2018 - average_lh_age_2018, 1)) + " years.")
```

Drunken Datetimes in Ames, Iowa

Task 1: Instructions

Begin working with this data by creating a bar chart of number of tests by day of the week.

- Load the `tidyverse` and `lubridate` packages.
- Read `"datasets/breath_alcohol_datetimes.csv"` into the workspace using `read_csv`, and save it as `ba_dates`. Change the timezone of the `DateTime` to `"America/Chicago"`.
- Using `wday()`, create a column in `ba_dates` called `wkday`, the weekday of the test. Set `label = TRUE` to extract Sun, Mon, etc. labels.
- Create a bar chart for the `ba_dates` data using `geom_bar()` with `wkday` on the x-axis.

Good to know

Taking Charlotte Wickham's DataCamp course Working with Dates and Times in R is a recommended prerequisite for this project. You should also know how to do some basic data manipulation (e.g., `mutate`, `filter`, `group_by`, `summarize`) using `tidyverse` packages. The Introduction to the Tidyverse course is the recommended resource.

Helpful links:

- Tidyverse cheat sheet
- `lubridate` and `ggplot2` documentation
- `wday()` function documentation
- If you want work with this data more, complete the project, Who is Drunk and When in Ames, Iowa?
- The `force_tz` function returns the same clock time in a different zone.

-

Solution

```
# Your solution code. This won't be shown to the student.

# load packages
library(tidyverse)
library(lubridate)

# read the data into your workspace
ba_dates <- read_csv("datasets/breath_alcohol_datetimes.csv")

# change DateTime column to America/Chicago with force_tz
ba_dates <- ba_dates %>% mutate(DateTime = force_tz(DateTime,
"America/Chicago"))
```

```
# group data by year and obtain counts for each year
ba_dates <- ba_dates %>% mutate(wkday = wday(DateTime, label = T))

# bar chart by day of week
ggplot(data = ba_dates, aes(x = wkday)) +
    geom_bar()
```

Task 2: Instructions

Look at the hour of the day for weekend days only and look at side-by-side bar charts for these three days by hour of the day.

- Create a column in `ba_dates` called `hr`, the hour the `DateTime` variable. Use `hour()`.
- Make a new data frame, `weekend`, by filtering `ba_dates` to include only tests from Friday, Saturday, and Sunday.
- Using the `weekend` data, create side-by-side bar charts (for each weekend day) of number of tests by `hr` of the day using `facet_grid()`.

The `facet_grid(x~y)` function will create rows of plots for each level of the `x` variable and columns of plots for each level of the `y` variable. To create only one row of plots, use `facet_grid(.~ var)`.

Helpful links:

- dplyr's `filter()` function documentation
- ggplot2's `facet_grid()` function documentation

-

Solution

```
# create hour variable
ba_dates <- ba_dates %>%
    mutate(hr = hour(DateTime))

# create weekend data
weekend <- ba_dates %>% filter(wkday %in% c("Fri", "Sat", "Sun"))

# plot side-by side bar charts of the distribution of hour of the day of
tests in each weekend day.
ggplot(data = weekend ) +
    geom_bar(aes(x = hr)) +
    facet_grid(.~wkday) +
    scale_x_continuous(breaks = 1:12*2-1)
```

Task 3: Instructions

After rounding the time of the test to the nearest date, summarize by date and plot some time series.

- Use the `round_date` function to round the `DateTime` column to the nearest day, creating a `date` column in `ba_dates`.
- Use the `count` function to count up the number of tests per date and save the result as `ba_summary`.
- Pipe (`%>%`) the `ba_summary` data into a `ggplot` command that uses `geom_line()` to make a time series plot.
- Use `scale_x_date()` to create labels on the x-axis every `"6 months"`.

To create a time series plot, put the time variable on the x-axis and the number of breathalyzer tests per day, `n`, on the y-axis.

Helpful links:

- lubridates's `round_date()` function [documentation]. Make sure the unit is `day`.
- ggplot2's `geom_line()` and `scale_*_date()` functions

-

Solution

```
# create a date column rounded to the nearest day. as.Date() is for the
plot later
ba_dates <- ba_dates %>%
  mutate(date = as.Date(round_date(DateTime, unit = "day")))

# count number of tests per date
ba_summary <- ba_dates %>% count(date)

# then group by and make a time series plot
ba_summary %>%
  ggplot() +
  geom_line(aes(x = date, y = n), alpha = .7) +  # change alpha for
readability
  scale_x_date(date_breaks = "6 months") +
  theme(axis.text.x = element_text(angle = 30))
```

Task 4: Instructions

Read in the football game data, and subset the `ba_summary` data to contain only the dates when Iowa State's football team played.

- Read in `"datasets/isu_football.csv"` using `read_csv()`. Save it as `isu_fb`.
- Make the `Date` column a date object with `parse_date()`. Supply the correct date format to `format`.
- Filter the `ba_summary` data to include only the dates in `isu_fb`. Save it as `ba_fb`.
- Arrange `ba_fb` so that the date with the most breathalyzer tests is shown first. Print the first six rows.

The dates in the football data are formatted as "Mo. Day, Year".

Helpful links:

- readr's `parse_date()` function documentation
- For the `format` argument, see documentation for the base function strptime()
- Check out the `%in%` operator
- dplyr's `arrange()` function documentation

-

Solution

```
# read in the football data
isu_fb <- read_csv("datasets/isu_football.csv")

# make Date a date variable
isu_fb <- isu_fb %>% mutate(Date = parse_date(Date, format = "%b %d, %Y"))

# filter ba_summary
ba_fb <- ba_summary %>% filter(date %in% isu_fb$Date)

# arrange ba_fb by number of tests from high to low and print first six
rows
ba_fb %>% arrange(desc(n)) %>% head()
```

Task 5: Instructions

Join the `ba_summary` data to the `isu_fb` data and create a visualization showing the difference in breathalyzer test counts between wins, losses, home, and away games.

- Do a `left_join()` to join `ba_summary` to `isu_fb`, creating the object `isu_fb2`.
- There are several game dates with no breathalyzer test data. Use `mutate` and `ifelse` to change the NAs to 0s.
- Create a bar chart with `n` on the x-axis. Fill by `Home` and facet by `Res` (the game result) to see the conditions with the most breathalyzer tests.

A "left join" keeps all rows in the first (or "left") dataset while taking only rows from the second (or "right") dataset that correspond to rows in the left. Joining `ba_summary` to `isu_fb` should keep all rows in `isu_fb` while taking only the matching dates in `ba_summary`. Use the `by` argument.

Helpful links:

- dplyr's `left_join()` function documentation
- `ifelse()` documentation

- Use `facet_grid(.~....)` (documentation) again
- `geom_bar()` documentation

-

Solution

```
# join ba_summary to isu_fb
isu_fb2 <- isu_fb %>% left_join(ba_summary, by = c("Date" = "date"))

# change nas to 0s
isu_fb2 <- isu_fb2 %>% mutate(n = ifelse(is.na(n), 0, n))

# plot
isu_fb2 %>%
  ggplot() +
  geom_bar(aes(x = n, fill = Home)) +
  facet_grid(.~Res)
```

Task 6: Instructions

Extract the month of each date in the `ba_dates` data and create visualizations of the number of tests by month and year.

- In `ba_dates`, create a column `mo` using the `month()` function and a column `yr` using the `year` function.
- Create a bar chart of number of tests per month.
- Create the same bar chart again but color the bars according to year. Use `as.factor(yr)` to color the bars as a categorical variable, not a numeric variable.

Helpful links:

- dplyr's `mutate()` function documentation
- lubridate's `month()` function documentation
- lubridate's `year()` function documentation
- `as.factor()` function documentation

-

Solution

```
# create a mo and a yr column in ba_dates
ba_dates <- ba_dates %>% mutate(mo = month(date, label = T), yr =
year(date))

# make bar chart by mo.
ggplot(data = ba_dates) +
    geom_bar(aes(x = mo))

# color by year
```

```
ggplot(data = ba_dates) +
    geom_bar(aes(x = mo, fill = as.factor(yr)))
```

Task 7: Instructions

Create time intervals for the weeks during VEISHEA and for similar weeks in other years, then see which year has the most tests given.

- Use the `make_date()` and `interval()` functions in `lubridate` to create time intervals for the VEISHEA weeks in 2013 and 2014. In 2013, VEISHEA was held from April 15-21. In 2014, it was held from April 7-13. Don't forget about the timezone!
- Create a data frame `veishea` by using the `%within%` and `filter()` to select only the `ba_dates` in a VEISHEA week.
- Using `count()`, count the number of breathalyzer tests during each VEISHEA week.

Helpful links:

- Documentation for `make_date()` and `interval()`
- Using `%within%`
- dplyr's `count()` and `filter()` functions

-

Solution

```
# In 2013, VEISHEA was held from April 15-21. In 2014, it was held from
April 7-13
v13 <- interval(make_date(2013, 4, 15) , make_date(2013, 4, 21), tzone =
"America/Chicago")
v14 <- interval(make_date(2014, 4, 7) , make_date(2014, 4, 13), tzone =
"America/Chicago")
# Other comparable VEISHEA weeks in 2015-2017
v15 <- interval(make_date(2015, 4, 13) , make_date(2015, 4, 19), tzone =
"America/Chicago")
v16 <- interval(make_date(2016, 4, 11) , make_date(2016, 4, 17), tzone =
"America/Chicago")
v17 <- interval(make_date(2017, 4, 10) , make_date(2017, 4, 16), tzone =
"America/Chicago")

# filter ba_dates for only the 5 VEISHEA intervals
veishea <- ba_dates %>%
  filter(date %within% v13 | date %within% v14 | date %within% v15 | date
%within% v16 | date %within% v17)

# count up years
veishea %>% count(yr)
```

Task 8: Instructions

Compute the mean BAC result and draw a ridgeline plot for the result by hour of the day.

- Using `mutate()`, create a column in ba_dates called `res` that is the mean of `Res1` and `Res2`.
- Load the `ggridges` package.
- Draw the ridgeline plot so that `res` is on the x-axis and one density ridge is drawn for each hour.

Helpful links:

- dplyr's `mutate()` function documentation
- `geom_density_ridges()` documentation
- Intro to ggridges vignette

-

Solution

```
# take a mean of res1, res2
ba_dates <- ba_dates %>% mutate(res = (Res1+Res2)/2)

# library pkg
library(ggridges)

# make ridgeline plot
ggplot(data = ba_dates, aes(x = res, y = hr, group = hr)) +
  geom_density_ridges(alpha = 0.7, fill = "steelblue", bandwidth = .01,
rel_min_height = 0.0001) +
  scale_y_continuous(breaks = 0:23)
```

Task 9: Instructions

Count the number of tests with breath alcohol reading of zero, then create another ridgeline plot that excludes zeroes.

- Create an indicator variable in `ba_dates` called `zero` that is `TRUE` when `res` is 0 and is `FALSE` otherwise.
- Tabulate the `zero` variable with the `count` function.
- Recreate the previous ridgeline plot with data that has been filtered to exclude the zero values.

Helpful links:

- dplyr's `count()` function documentation
- dplyr's `filter()` function documentation
- `geom_density_ridges()` documentation
- Intro to ggridges vignette

-

Solution

```
# create a zero indicator variable
ba_dates <- ba_dates %>% mutate(zero = res == 0)

# tabulate the data by the zero column
count(ba_dates, zero)

# redo ridge with no 0s
ba_dates %>% filter(res > 0) %>%
ggplot(aes(x = res, y = hr, group = hr)) +
  geom_density_ridges(alpha = 0.7, fill = "steelblue", bandwidth = .01,
rel_min_height = 0.005) +
  scale_y_continuous(breaks = 0:23)
# other solution
# ba_dates %>% filter(!zero) %>%
# ggplot(aes(x = res, y = hour, group = hour)) +
#   geom_density_ridges(alpha = 0.7, fill = "steelblue", bandwidth = .01,
rel_min_height = 0.005) +
#   scale_y_continuous(breaks = 0:23)
```

Task 10: Instructions

Subset the data to contain tests with results greater than 0.31, and examine the time of day they occurred.

- Create a new data frame, `danger`, that contains `res` of at least 0.31.
- Print `danger` and examine the dates and times of these tests.

-

Solution

```
# filter the ba_dates data to contain only those with the most dangerous
result
danger <- ba_dates %>% filter(res >= 0.31)

# print danger
danger
```

Which Debts Are Worth the Bank's Effort?

Task 1: Instructions

Load the data.

- Import `pandas` under the alias `pd` and `numpy` under the alias `np`.
- Read in `"datasets/bank_data.csv"` using `read_csv()` and save it as `df`.

This project lets you apply the skills from [Data Manipulation with pandas](), including reading, exploring, filtering, and grouping data. We recommend that you take those courses before starting this project.

This project also uses basic statistics. You may wish to take an introduction to statistics course like [Statistical Thinking in Python]() or review this [tutorial]().

In this task, the `head()` command is used to print the first few rows of the DataFrame.

-

Solution

```
# Importing modules
import pandas as pd
import numpy as np

# Read in dataset
df = pd.read_csv('datasets/bank_data.csv')

# Print the first few rows of the DataFrame
df.head()
```

Task 2: Instructions

Create a scatter plot of Age vs. Expected Recovery Amount.

- From `matplotlib`, import the `pyplot` module under the alias `plt`.
- Plot `expected_recovery_amount` on the x-axis and `age` on the y-axis.
- Set the label on the x-axis to "Expected Recovery Amount" and the label on the y-axis to "Age".
- Remember to finish the `plt` statement with `plt.show()`.

Style the plot to your preference if you'd like! The automatic grader does not check for plot "correctness," so please consult this [image of one accepted plot]() to verify your solution.

Helpful links:

- Scatter plot demo
-

Solution

```
# Scatter plot of Age vs. Expected Recovery Amount
from matplotlib import pyplot as plt
%matplotlib inline
plt.scatter(x=df['expected_recovery_amount'], y=df['age'], c="g", s=2)
plt.xlim(0, 2000)
plt.ylim(0, 60)
plt.xlabel("Expected Recovery Amount")
plt.ylabel("Age")
plt.legend(loc=2)
plt.show()
```

Task 3: Instructions

Compute the average age just above and just below the threshold then test if these average ages are different.

- To compute `era_900_1100`, filter for the values of `expected_recovery_amount` that are greater than or equal to $900 and less than $1100.
- To compute `Level_0_Age`, filter for the values of `era_900_1100` that correspond to "Level 0 Recovery", then select the `age` column.
- Do the same for `Level_1_Age` (filtering for "Level 1 Recovery").
- Compute the Kruskal-Wallis test to see if they are statistically significantly different using `stats.kruskal(Level_0_age,Level_1_age)`.

---

t-tests can be used to compare the average value of two populations. This statistical test makes specific assumptions about the distributions of the variables that can be understood more by reviewing the documentation.

The Kruskal-Wallis test is a non-parametric test meaning that it doesn't make any assumptions about the distributions. You can learn more about the Kruskal-Wallis test by reading the documentation.

-

Solution

```
# Import stats module
from scipy import stats

# Compute average age just below and above the threshold
era_900_1100 = df.loc[(df['expected_recovery_amount']<1100) &
                      (df['expected_recovery_amount']>=900)]
```

```
by_recovery_strategy = era_900_1100.groupby(['recovery_strategy'])
by_recovery_strategy['age'].describe().unstack()

# Perform Kruskal-Wallis test
Level_0_age = era_900_1100.loc[df['recovery_strategy']=="Level 0 Recovery"]
['age']
Level_1_age = era_900_1100.loc[df['recovery_strategy']=="Level 1 Recovery"]
['age']
stats.kruskal(Level_0_age,Level_1_age)
```

Task 4: Instructions

Compute the chi-square test for Sex versus Recovery Strategy to see if the sex distribution differs across Recovery Strategy.

- To compute the crosstab of `sex` and `recovery_strategy`, filter `df` for values of `expected_recovery_amount` >= $900 and < $1100. Do not include any `dropna` statement.
- Print `crosstab`.
- Run the chi-square test by inputting the `crosstab` variable into the `stats.chi2_contingency()` function.
- Print the p-value using the command `p_val`.

The chi-square test is often used to see if two categorical variables are independent or dependent. If they are independent, the p-value is not likely to be statistically significant while if they are dependent, the p-value is more likely to be significant (for example, less than 0.01).

Helpful links:

- Chi-square documentation
-

Solution

```
# Number of customers in each category
crosstab = pd.crosstab(df.loc[(df['expected_recovery_amount']<1100) &
                              (df['expected_recovery_amount']>=900)]
['recovery_strategy'],
                       df['sex'])
print(crosstab)

# Chi-square test
chi2_stat, p_val, dof, ex = stats.chi2_contingency(crosstab)
p_val
```

Task 5: Instructions

Create a scatter plot of Actual Recovery Amount versus Expected Recovery Amount.

- Set `x` to `expected_recovery_amount` and `y` to `actual_recovery_amount`.
- Set the label on the x-axis to "Expected Recovery Amount" and the label on the y-axis to "Actual Recovery Amount".
- Remember to finish the `plt` statement with `plt.show()`.

Style the plot to your preference if you'd like! The automatic grader does not check for plot "correctness," so please consult this [image of one accepted plot]() to verify your solution.

Helpful links:

- Scatter plot demo
-

Solution

```
# Scatter plot of Actual Recovery Amount vs. Expected Recovery Amount
plt.scatter(x=df['expected_recovery_amount'],
y=df['actual_recovery_amount'], c="g", s=2)
plt.xlim(900, 1100)
plt.ylim(0, 2000)
plt.xlabel("Expected Recovery Amount")
plt.ylabel("Actual Recovery Amount")
plt.legend(loc=2)
plt.show()
```

Task 6: Instructions

Test if the average actual recovery amounts just above and just below the threshold are different.

- Create variables called `Level_0_actual` and `Level_1_actual`. These are the actual recovery amounts of the customers with expected recovery amounts between $900 and $1100 belonging to Level 0 and Level 1 respectively.
- Compute the Kruskal-Wallis test to see if they are statistically significantly different using the `stats.kruskal()` function.
- Redefine `Level_0_actual` and `Level_1_actual` as $950 to $1050, then perform the Kruskal-Wallis test again.

Helpful links:

- Many people use the t-test to compare the average values of two populations. This statistical test makes specific assumptions about the distributions of the variables that can be understood more by reviewing the documentation.

- The Kruskal-Wallis test is a non-parametric test meaning that it doesn't make any assumptions about the distributions. You can learn more about the Kruskal-Wallis test by reading the documentation.

-

Solution

```
# Compute average actual recovery amount just below and above the threshold
by_recovery_strategy['actual_recovery_amount'].describe().unstack()

# Perform Kruskal-Wallis test
Level_0_actual = era_900_1100.loc[df['recovery_strategy']=='Level 0
Recovery']['actual_recovery_amount']
Level_1_actual = era_900_1100.loc[df['recovery_strategy']=='Level 1
Recovery']['actual_recovery_amount']
stats.kruskal(Level_0_actual,Level_1_actual)

# Repeat for a smaller range of $950 to $1050
era_950_1050 = df.loc[(df['expected_recovery_amount']<1050) &
                      (df['expected_recovery_amount']>=950)]
Level_0_actual = era_950_1050.loc[df['recovery_strategy']=="Level 0
Recovery"]['actual_recovery_amount']
Level_1_actual = era_950_1050.loc[df['recovery_strategy']=="Level 1
Recovery"]['actual_recovery_amount']
stats.kruskal(Level_0_actual,Level_1_actual)
```

Task 7: Instructions

Compute a regression model to predict the actual recovery amount (y) using expected recovery amount (x).

- Select the `expected_recovery_amount` column of `era_900_1100` and assign it to `x`.
- Select the `actual_recovery_amount` column of `era_900_1100` and assign it to `y`.
- Print out the model summary using `model.summary()`.

Regression modeling is a basic method of building predictive models that can be understood more by reviewing the documentation.

-

Solution

```
# Import statsmodels
import statsmodels.api as sm

# Define X and y
X = era_900_1100['expected_recovery_amount']
y = era_900_1100['actual_recovery_amount']
X = sm.add_constant(X)
```

```
# Build linear regression model
model = sm.OLS(y, X).fit()
predictions = model.predict(X)

# Print out the model summary statistics
model.summary()
```

Task 8: Instructions

Compute a regression model to predict the actual recovery amount using expected recovery amount and the recovery strategy as the indicator.

- To create the new `indicator_1000` column, filter `df` for expected recovery amounts less than 1000.
- Select the `expected_recovery_amount` and `indicator_1000` columns from `era_900_1100` and assign it to `X`.
- Select the `actual_recovery_amount` column from `era_900_1100` and assign it to `y`.
- Print out the model summary using `model.summary()`.

Regression modeling is a basic method of building predictive models that can be understood more by reviewing the documentation.

-

Solution

```
# Create indicator (0 or 1) for expected recovery amount >= $1000
df['indicator_1000'] = np.where(df['expected_recovery_amount']<1000, 0, 1)
era_900_1100 = df.loc[(df['expected_recovery_amount']<1100) &
                      (df['expected_recovery_amount']>=900)]

# Define X and y
X = era_900_1100[['expected_recovery_amount','indicator_1000']]
y = era_900_1100['actual_recovery_amount']
X = sm.add_constant(X)

# Build linear regression model
model = sm.OLS(y,X).fit()

# Print the model summary
model.summary()
```

Task 9: Instructions

Perform the same analysis in task 8, but for the range of $950 to $1050.

- Redefine `era_950_1050` so the indicator variable created in task 8 is included.

- Define x as the `expected_recovery_amount` and the `indicator_1000` when the expected recovery amount was >= $950 and < $1050.
- Define y as the `actual_recovery_amount` when the expected recovery amount was >= $950 and < $1050.

Congratulations on reaching the end of the project!

-

Solution

```
# Redefine era_950_1050 so the indicator variable is included
era_950_1050 = df.loc[(df['expected_recovery_amount']<1050) &
                      (df['expected_recovery_amount']>=950)]

# Define X and y
X = era_950_1050[['expected_recovery_amount','indicator_1000']]
y = era_950_1050['actual_recovery_amount']
X = sm.add_constant(X)

# Build linear regression model
model = sm.OLS(y,X).fit()

# Print the model summary
model.summary()
```

ASL Recognition with Deep Learning

Task 1: Instructions

First, load the dataset using a special helper file (`sign_language.py`).

- Run the code cell as-is, without modification.

Good to know

This project lets you practice the skills from [Introduction to Deep Learning in Python] and [Image Processing with Keras in Python], including building convolutional neural networks to classify images. We recommend that you take those courses before starting this project.

-

Solution

```
# Import packages and set numpy random seed
import numpy as np
np.random.seed(5)
import tensorflow as tf
tf.set_random_seed(2)
from datasets import sign_language
import matplotlib.pyplot as plt
%matplotlib inline

# Load pre-shuffled training and test datasets
(x_train, y_train), (x_test, y_test) = sign_language.load_data()
```

Task 2: Instructions

Use `matplotlib` to visualize some of the images in the training dataset.

- Assign `labels` to a Python list with three items: `'A'`, `'B'`, and `'C'`, corresponding to the signed letters that appear in the images.

-

Solution

```
# Store labels of dataset
labels = ['A', 'B', 'C']
```

```
# Print the first several training images, along with the labels
fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
    ax.set_title("{}".format(labels[y_train[i]]))
plt.show()
```

Task 3: Instructions

Count the number of occurrences of each letter in the train and test datasets.

- Assign the variable `num_B_train` to an integer counting the number of times that `1` appears in `y_train`.
- Assign the variable `num_C_train` to an integer counting the number of times that `2` appears in `y_train`.
- Assign the variable `num_B_test` to an integer counting the number of times that `1` appears in `y_test`.
- Assign the variable `num_C_test` to an integer counting the number of times that `2` appears in `y_test`.

-

Solution

```
# Number of A's in the training dataset
num_A_train = sum(y_train==0)
# Number of B's in the training dataset
num_B_train = sum(y_train==1)
# Number of C's in the training dataset
num_C_train = sum(y_train==2)

# Number of A's in the test dataset
num_A_test = sum(y_test==0)
# Number of B's in the test dataset
num_B_test = sum(y_test==1)
# Number of C's in the test dataset
num_C_test = sum(y_test==2)

# Print statistics about the dataset
print("Training set:")
print("\tA: {}, B: {}, C: {}".format(num_A_train, num_B_train,
num_C_train))
print("Test set:")
print("\tA: {}, B: {}, C: {}".format(num_A_test, num_B_test, num_C_test))
```

Task 4: Instructions

Use the built-in Keras function `to_categorical` to one-hot encode the data.

- Use the class vector `y_train` to assign the variable `y_train_OH` to the one-hot training labels.
- Use the class vector `y_test` to assign the variable `y_test_OH` to the one-hot test labels.

Helpful links:

- Keras `to_categorical` function [documentation]

-

Solution

```
from keras.utils import np_utils

# One-hot encode the training labels
y_train_OH = np_utils.to_categorical(y_train, 3)

# One-hot encode the test labels
y_test_OH = np_utils.to_categorical(y_test, 3)
```

Task 5: Instructions

Specify a convolutional neural network in Keras.

- The first convolutional layer in the network has already been provided in the code. Add a max pooling layer (pooling over windows of size 4x4).
- Add another convolutional layer (15 filters, kernel size of 5, `same` padding, `relu` activation).
- Add another max pooling layer (pooling over windows of size 4x4).

Helpful links:

- Keras `Conv2D` [documentation]
- Keras pooling layers [exercise] in the Convolutional Neural Networks for Image Processing course

-

Solution

```
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Flatten, Dense
from keras.models import Sequential

model = Sequential()
# First convolutional layer accepts image input
```

```
model.add(Conv2D(filters=5, kernel_size=5, padding='same',
activation='relu',
                        input_shape=(50, 50, 3)))
# Add a max pooling layer
model.add(MaxPooling2D(pool_size=4))
# Add a convolutional layer
model.add(Conv2D(filters=15, kernel_size=5, padding='same',
activation='relu'))
# Add another max pooling layer
model.add(MaxPooling2D(pool_size=4))
# Flatten and feed to output layer
model.add(Flatten())
model.add(Dense(3, activation='softmax'))

# Summarize the model
model.summary()
```

Task 6: Instructions

Specify the optimizer and loss function, along with a metric that will be tracked during training.

- Compile the model with the `'rmsprop'` optimizer, `'categorical_crossentropy'` as the loss function, and `'accuracy'` as a metric.

Helpful links:

- Compile a neural network exercise in the Convolutional Neural Networks for Image Processing course
-

Solution

```
# Compile the model
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Task 7: Instructions

Use the model's `.fit()` method to train the model.

- Use the training data in `x_train` and the label data in `y_train_OH` to train the model for two epochs. You must attain a final validation accuracy of at least 80%. (Note that you should get good results by setting aside 20% of the training data to be used as validation data, and by using a batch size of 32.)

Helpful links:

- Fitting a neural network to clothing data exercise in the Convolutional Neural Networks for Image Processing course
-

Solution

```
# Train the model
hist = model.fit(x_train, y_train_OH,
                 validation_split=0.2,
                 epochs=2,
                 batch_size=32)
```

Task 8: Instructions

Use the model's `.evaluate()` method to evaluate the model.

- Assign `x` to the test data in `x_test`, and assign `y` to the test labels in `y_test_OH`.

Helpful links:

- Keras `.evaluate()` method documentation
-

Solution

```
# Obtain accuracy on test set
score = model.evaluate(x=x_test,
                       y=y_test_OH,
                       verbose=0)
print('Test accuracy:', score[1])
```

Task 9: Instructions

Visualize images that were incorrectly classified by the model.

- Use the model's `.predict()` method to assign `y_probs` to a numpy array with shape `(600, 3)` containing the model's predicted probabilities that each image belongs to each image class.
- Assign `y_preds` to the model's predicted labels for each image in the testing dataset. Note that `y_preds` should be a numpy array with shape

`(600,)`, where each entry is one of `0`, `1`, or `2`, corresponding to `'A'`, `'B'`, and `'C'`, respectively.
- Use the ground truth labels for the testing dataset (`y_test`) and the model's predicted labels (`y_preds`) to determine which images were misclassified. Assign the variable `bad_test_idxs` to a one-dimensional numpy array containing all indices corresponding to images that were incorrectly classified by the model.

Helpful links:

- Keras `.predict()` method [documentation]

-

Solution

```
# Get predicted probabilities for test dataset
y_probs = model.predict(x_test)

# Get predicted labels for test dataset
y_preds = np.argmax(y_probs, axis=1)

# Indices corresponding to test images which were mislabeled
bad_test_idxs = np.where(y_preds!=y_test)[0]

# Print mislabeled examples
fig = plt.figure(figsize=(25,4))
for i, idx in enumerate(bad_test_idxs):
    ax = fig.add_subplot(2, np.ceil(len(bad_test_idxs)/2), i + 1,
xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    ax.set_title("{} (pred: {})".format(labels[y_test[idx]],
labels[y_preds[idx]]))
```

A Text Analysis of Trump's Tweets

Task 1: Instructions

Load the libraries and data, and filter for the election period.

- Load the `dplyr`, `readr`, `tidyr` and `lubridate` libraries.
- Read in the data, `datasets/trump_tweets.csv`, with `read_csv()` and filter for the election period between June 1, 2015, and November 8, 2016. Dates are in the `created_at` column.
- Inspect the first six rows using `head()`. Pay attention to the column names and how the data are formatted.

This project was updated on December 21, 2019. If you started the project before that date, please click the circular arrow in the bottom-right corner of the screen to reset the project. To save your code, download your project before resetting it.

Good to know

This project lets you apply the skills from Introduction to the Tidyverse, Intermediate Data Visualization with ggplot2, String Manipulation in R with stringr, and Sentiment Analysis in R. We recommend that you take those course before starting this project.

Helpful links:

- Tidyverse cheat sheet
- Work with Strings cheat Sheet (Scroll down to "Work with Strings Cheat Sheet")
- Introduction to tidytext
-

Solution

```
# Load the libraries
library(dplyr)
library(readr)
library(tidyr)
library(lubridate)

# Read in the data
tweets <- read_csv("datasets/trump_tweets.csv", guess_max = 36000) %>%
  filter(created_at >= "2015-06-01", created_at <= "2016-11-08")

# Inspect the first six rows
head(tweets)
```

Task 2: Instructions

Count the number of tweets by the device, and then filter for tweets from iPhone and Android.

- Use `count()` to determine the number of tweets by each source.
- Select `id_str`, `source`, `text`, and `created_at`, and filter for tweets from iPhone and Android.
- Use `extract()` to remove the leading characters, "Twitter for ", in `source`.
- Inspect the first six rows.

The output of `count()` is a column, `n`. `extract()` turns groups into new columns using a regular expression.

Helpful links:

- count() documentation
- extract() documentation
- Interactive regex website

-

Solution

```
# Count the nubmer of tweets by source
tweets %>% count(source, sort = TRUE)

# Clean the tweets
cleaned_tweets <- tweets %>%
  select(id_str, source, text, created_at) %>%
  filter(source %in% c("Twitter for iPhone", "Twitter for Android")) %>%
  extract(source, "source", "(\\w+)$")

# Inspect the first six rows
head(cleaned_tweets)
```

Task 3: Instructions

Plot the percentage of tweets by hour of the day for each device.

- Use `count()` to count the number of tweets from each device by hour (in "EST").
- Add a new column, `percent`, that is the percent of tweets by each device within each hour.
- Plot the percent of tweets by hour, colored by `source` as a line graph.
- Add labels to the plot using `labs(x = "Hour of day (EST)", y = "% of tweets")`, and an empty string for `color` so that no legend is displayed.

You can create variables to group by within the call to `count()`.

Calling `label_percent()` from `scales` in the call to `scale_y_continuous()` changes the y-axis format.

Don't forget to add `%>%` and `+` where needed.

Helpful links:

- count() documentatation
-

Solution

```
# Load the packages
library(ggplot2)

# Plot the percentage of tweets by hour of the day for each device
cleaned_tweets %>%
  count(source, hour = hour(with_tz(created_at, "EST"))) %>%
  mutate(percent = n / sum(n)) %>%
  ggplot(aes(hour, percent, color = source)) +
  geom_line() +
  scale_y_continuous(labels = scales::label_percent()) +
  labs(x = "Hour of day (EST)",
       y = "% of tweets",
       color = "")
```

Task 4: Instructions

Create a bar plot of the number of tweets that are quoted and not quoted from each device.

- Use `count()` to determine the number of tweets quoted or not quoted by each device.
  - Within `ifelse()`, set the true outcome to `"Quoted"`, and the false outcome to `"Not quoted"`.
- Add `source` as the x-variable and the number of quoted and not quoted tweets as the y-variable. Set `fill=` to whether the tweet was quoted or not.
- Set the correct parameter for `position=` within `geom_bar()` to display the bars for each device side-by-side.

`str_detect(text, '^"')` is a handy bit of code to find the quote tweets, where `text` is the column in the data frame that contains the full character string of each tweet (one tweet per row).

Helpful links:

- geom_bar()

- count() documentation
- stringr documentation

-

Solution

```
# Load stringr
library(stringr)

# Plot the number of tweets with and without quotes by device
cleaned_tweets %>%
  count(source,
        quoted = ifelse(str_detect(text, '^"'), "Quoted", "Not quoted")) %>
%
  ggplot(aes(source, n, fill = quoted)) +
  geom_bar(stat = "identity", position = "dodge") +
  labs(x = "", y = "Number of tweets", fill = "") +
  ggtitle('Whether tweets start with a quotation mark (")')
```

Task 5: Instructions

Create a bar plot of the number of tweets that do and do not have a picture/link from each device.

- After filtering out all the quote tweets, count the number of tweets with a picture/link by each device.
  - Add the correct `stringr` function within `ifelse()` to detect the pattern, `t.co`, in the `text` column.
- Using the new data frame, add source as the x-variable and the number of tweets as the y-variable. Set `fill=` to whether or not the tweet had a picture/link.
- Set the correct parameter for `position=` within geom_bar() to display the bars for each device side-by-side.

Now we're filtering out the quote tweets with `filter(!str_detect(text, '^"'))`.

Helpful links:

- Logical operators
- geom_bar()
- count() documentation
- Introduction to stringr

-

Solution

```
# Count the number of tweets with and without picture/links by device
tweet_picture_counts <- cleaned_tweets %>%
```

```
  filter(!str_detect(text, '^"')) %>%
  count(source,
        picture = ifelse(str_detect(text, "t.co"),
                         "Picture/link", "No picture/link"))

# Make a bar plot
ggplot(tweet_picture_counts, aes(source, n, fill = picture)) +
  geom_bar(stat = "identity", position = "dodge") +
  labs(x = "", y = "Number of tweets", fill = "")
```

Task 6: Instructions

Create a new data frame of words from all the tweets.

- Add the correct regex pattern to remove quote tweets. Look back at previous tasks for help.
- Use `unnest_tokens()` to transform the lines of text into words (in a new `word` column) using the regex pattern in `reg`.
- Remove any stopwords using the correct column from the `stop_words` data frame.

The data frame `stop_words` is loaded with `tidytext`. The column `word` contains the "stopwords" in the data frame `stop_words`.

Helpful links:

- Introduction to tidytext
- Regular Expression

-

Solution

```
# Load the tidytext package
library(tidytext)

# Create a regex pattern
reg <- "([^A-Za-z\\d#@']|'(?![A-Za-z\\d#@]))"

# Unnest the text strings into a data frame of words
tweet_words <- cleaned_tweets %>%
  filter(!str_detect(text, '^"')) %>%
  mutate(text = str_replace_all(text, "https://t.co/[A-Za-z\\d]+|&amp;",
"")) %>%
  unnest_tokens(word, text, token = "regex", pattern = reg) %>%
  filter(!word %in% stop_words$word,
         str_detect(word, "[a-z]"))

# Inspect the first six rows of tweet_words
head(tweet_words)
```

Task 7: Instructions

Plot the most common words.

- Use `count()` to count the most common words and sort them.
- Take the first 20 words and reorder them according to their number of occurrences.
- Plot `word` on the x-axis, the number of occurrences on the y-axis, and flip the coordinates.

Remember the output of `count()` is a variable, `n`.

Helpful links:

- geom_bar()
- count() documentation
- Coordinate systems in ggplot2

-

Solution

```
# Plot the most common words from @realDonaldTrump tweets
tweet_words %>%
  count(word, sort = TRUE) %>%
  head(20) %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(word, n)) +
  geom_bar(stat = "identity") +
  ylab("Occurrences") +
  coord_flip()
```

Task 8: Instructions

Create the log odds ratio for each word.

- After counting the occurrences of each word by its source, group by the word, and filter for words that occur at least five times in both platforms.
- Use `spread()` to turn the row values of `source` into their columns.
- Divide Android by iPhone and take the binary log of the ratio to create `logratio`.
- Arrange the data in descending order of the `logratio`.

The type of device, Android or iPhone, is in `source`.

Helpful links:

- Odds ratio
- `spread()` documentation
- summarize and mutate multiple columns

-

Solution

```
# Create the log odds ratio of each word
android_iphone_ratios <- tweet_words %>%
  count(word, source) %>%
  group_by(word)  %>%
  filter(sum(n) >= 5) %>%
  spread(source, n, fill = 0) %>%
  ungroup()  %>%
  mutate_if(is.numeric, ~((. + 1) / sum(. + 1))) %>%
  mutate(logratio = log2(Android / iPhone)) %>%
  arrange(desc(logratio))

# Inspect the first six rows
head(android_iphone_ratios)
```

Task 9: Instructions

Plot the log odds ratios by device.

- Group the data by positive and negative values.
- Get the top 15 values of each group.
- In the call to `ggplot()`, put the words on the x-axis and their log ratio on the y-axis.
- Use the correct parameter in `geom_bar()` to map the height of the bar to the data value in y.

If you want to take the top positive and negative values from grouped variables, you might want to think about absolute values.

Helpful links:
Absolute values in R
`top_n()` documentation

-

Solution

```
# Plot the log odds ratio for each word by device
android_iphone_ratios %>%
  group_by(logratio > 0) %>%
  top_n(15, abs(logratio)) %>%
  ungroup() %>%
  mutate(word = reorder(word, logratio)) %>%
  ggplot(aes(word, logratio, fill = logratio < 0)) +
```

```
  geom_bar(stat = "identity") +
  coord_flip() +
  ylab("Android / iPhone log ratio") +
  scale_fill_manual(name = "", labels = c("Android", "iPhone"),
                    values = c("red", "lightblue"))
```

Task 10: Instructions

Add the NRC sentiment lexicon to the log odds ratio data frame.

- Read in the NRC sentiment lexicon with `read_rds()`.
- Use a join that will keep all the variables from `android_iphone_ratios`, and if there are multiple matches between `android_iphone_ratios` and `nrc`, all combinations of the matches will be returned.
- Reorder the values of `sentiment` and `word`.
- Take the top 10 values of each sentiment group.

The data frame `sentiments` was loaded with `tidytext`.

Helpful links:

- dplyr joins
- Introduction to tidytext

-

Solution

```
# Create a sentiment data frame from the NRC lexicon
nrc <- read_rds("datasets/nrc.rds")

# Join the NRC lexicon to log odds ratio data frame
android_iphone_sentiment <- android_iphone_ratios %>%
  inner_join(nrc, by = "word") %>%
  filter(!sentiment %in% c("positive", "negative")) %>%
  mutate(sentiment = reorder(sentiment, -logratio),
         word = reorder(word, -logratio)) %>%
  group_by(sentiment) %>%
  top_n(10, abs(logratio)) %>%
  ungroup()

# Inspect the first six rows
head(android_iphone_sentiment)
```

Task 11: Instructions

Plot the log odds ratio of each word from both devices by sentiment.

- Add the data frame and x and y aesthetics to plot the log odds ratio of each word.

- Facet the data by `sentiment` and create two rows.
- Use the correct parameter in `geom_bar()` to map the height of the bar to the data value in y.

Helpful links:

- facets in ggplot2
-

Solution

```
# Plot the log odds ratio of words by device in groups sentiments
ggplot(android_iphone_sentiment, aes(word, logratio, fill = logratio < 0))
+
  facet_wrap(~ sentiment, scales = "free", nrow = 2) +
  geom_bar(stat = "identity") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  labs(x = "", y = "Android / iPhone log ratio") +
  scale_fill_manual(name = "", labels = c("Android", "iPhone"),
                    values = c("red", "lightblue"))
```

Task 12: Instructions

What do you think?

- Pick one: "True Believer" or "Cog"

Congratulations on reaching the end of the project!

This project was adapted with permission, from David Robinson's blog post, *Text analysis of Trump's tweets confirms he writes only the (angrier) Android half*, published on August 9, 2016. An update to the post, *Trump's Android and iPhone tweets, one-year later*, was published on August 9, 2017.

-

Solution

```
anonymous_iPhone_tweeter <- ""
```

Functions for Food Price Forecasts

Task 1: Instructions

Import the potato data and view its structure.

- Load the `readr` and `dplyr` packages.
- Read the CSV potatoes dataset from `"datasets/Potatoes (Irish).csv"`, assigning to `potato_prices`.
- Take a look at the structure of the data by using `glimpse()`.

Good to know

This project uses concepts found in the following courses.

- Manipulating data frames with `dplyr` and drawing line plots with `ggplot2`, as covered in Introduction to the Tidyverse.
- Reading data from CSV files with `readr`, covered in Chapter 2 of Introduction to Importing Data in R.
- Specifying and extracting parts of dates with `lubridate`, covered in Chapter 2 of Working with Times and Dates in R.
- Simple time series forecasting with `forecast`, as covered in Chapter 1 of Forecasting in R
- Writing functions, as covered in Introduction to Writing Functions in R.

-

Solution

```
# Load the readr and dplyr packages
library(readr)
library(dplyr)

# Import the potatoes dataset
potato_prices <- read_csv("datasets/Potatoes (Irish).csv")

# Take a glimpse at the contents
glimpse(potato_prices)
```

Task 2: Instructions

Read the potato data again, this time limiting the columns to be read, and changing the column names to be more readable.

- Read the potato data from `"datasets/Potatoes (Irish).csv"` again, specifying `col_types` to keep only these columns: `adm1_name`, `mkt_name`, `cm_name`, `mp_month`, `mp_year`, and `mp_price`.
- Rename those columns to `region`, `market`, `commodity_kg`, `month`, `year`, and `price_rwf`.
- Check the new structure of the new dataset, `potato_prices_renamed`.

Helpful links:

- readr's `read_csv()` function documentation.
- readr's `cols_only()` function documentation.
- dplyr's `rename()` function documentation.
- dplyr's `glimpse()` function documentation.
- Introduction to Importing Data in R, Chapter 2, Exercise 7 provides an alternate way of specifying the column types.

-

Solution

```
# Import again, only reading specific columns
potato_prices <- read_csv(
  "datasets/Potatoes (Irish).csv",
  col_types = cols_only(
    adm1_name = col_character(),
    mkt_name = col_character(),
    cm_name = col_character(),
    mp_month = col_integer(),
    mp_year = col_integer(),
    mp_price = col_double()
  )
)

# Rename the columns to be more informative
potato_prices_renamed <- potato_prices %>%
  rename(
    region = adm1_name,
    market = mkt_name,
    commodity_kg = cm_name,
    month = mp_month,
    year = mp_year,
    price_rwf = mp_price
  )

# Check the result
glimpse(potato_prices_renamed)
```

Task 3: Instructions

Convert the years and months in the potato dataset to dates.

- Load the `lubridate` package.

- Add a new column, `date`, to `potato_prices_renamed`. This should be a `Date`, constructed from the `year`, the `month`, and assuming that the day is the first of the month, `01`.
- Drop the `year` and `month` columns.
- Take a look at the structure of the new dataset, `potato_prices_cleaned`.

-

Solution

```
# Load lubridate
library(lubridate)

# Convert year and month to Date
potato_prices_cleaned <- potato_prices_renamed %>%
  mutate(
    date = ymd(paste(year, month, "01"))
  ) %>%
  select(-month, -year)

# See the result
glimpse(potato_prices_cleaned)
```

Task 4: Instructions

Wrap the importing and cleaning code into a function that can be used with other commodities.

- Write a function, `read_price_data`, with one input, `commodity`, which is the name of a commodity.
  - The function should convert the commodity into a filename of the form `datasets/commodity.csv`.
  - Then the function should perform the importing, renaming, and cleaning steps you used on the potato data.
  - Finally, the function should return the cleaned dataset.
- Test the function by calling it with `"Peas (fresh)"`.

Helpful links:

- This video from Introduction to Function Writing in R shows how to convert a snippet of code into a function.

-

Solution

```
# Wrap this code into a function
read_price_data <- function(commodity) {
  data_file <- paste0("datasets/", commodity, ".csv")
  prices <- read_csv(
```

```
    data_file,
    col_types = cols_only(
      adm1_name = col_character(),
      mkt_name = col_character(),
      cm_name = col_character(),
      mp_month = col_integer(),
      mp_year = col_integer(),
      mp_price = col_double()
    )
  )

  prices_renamed <- prices %>%
    rename(
      region = adm1_name,
      market = mkt_name,
      commodity_kg = cm_name,
      month = mp_month,
      year = mp_year,
      price_rwf = mp_price
    )

  prices_renamed %>%
    mutate(
      date = ymd(paste(year, month, "01"))
    ) %>%
    select(-month, -year)
}

# Test it
pea_prices <- read_price_data("Peas (fresh)")
glimpse(pea_prices)
```

Task 5: Instructions

Plot the potato price over time for each market.

- Load the `ggplot2` package.
- Using `potato_prices_cleaned`, plot `price_rwf` vs. `date`, grouped by `market`.
- Make it a line plot. Since there are many lines, set the line transparency (`alpha`) to `0.2`.
- Set the plot title to `"Potato price over time"`.

-

Solution

```
# Load ggplot2
library(ggplot2)

# Draw a line plot of price vs. date grouped by market
potato_prices_cleaned %>%
  ggplot(aes(date, price_rwf, group = market)) +
  geom_line(alpha = 0.2) +
  ggtitle("Potato price over time")
```

Task 6: Instructions

Wrap the plotting code into a function.

- Create a function, `plot_price_vs_time`, with inputs `prices`, a data frame of food price data, and `commodity`, a string naming the commodity.
    - Amend the potato price plotting code to use `prices`.
    - Make the plot title display the correct commodity.
- Test the function with `pea_prices` and the commodity name `"Pea"`.

-

Solution

```
# Wrap this code into a function
plot_price_vs_time <- function(prices, commodity) {
  prices %>%
    ggplot(aes(date, price_rwf, group = market)) +
    geom_line(alpha = 0.2) +
    ggtitle(paste(commodity, "price over time"))
}

# Try the function on the pea data
plot_price_vs_time(pea_prices, "Pea")
```

Task 7: Instructions

Calculate the median potato price at each time point, assigning to `potato_prices_summarized`.

- Using `potato_prices_cleaned`, group by each `date`.
- Create a summary column, `median_price_rwf`, equal to the median of `price_rwf`.

-

Solution

```
# Group by date, and calculate the median price
potato_prices_summarized <- potato_prices_cleaned %>%
  group_by(date) %>%
  summarize(median_price_rwf = median(price_rwf))

# See the result
potato_prices_summarized
```

Task 8: Instructions

Convert the potato data frame into a time series.

- Load the `magrittr` package.
- Create a time series, `potato_time_series`, from `potato_prices_summarized`'s `median_price_rwf` column.
    - The `start` value is a vector of length 2: the year then the month of the first (minimum) date.
    - The `end` value is the year then the month of the last date.
    - The `frequency` is the number of months in a year.
- Look at the resulting time series.

-

Solution

```
# Load magrittr
library(magrittr)

# Extract a time series
potato_time_series <- potato_prices_summarized %$%
  ts(
    median_price_rwf,
    start = c(year(min(date)), month(min(date))),
    end   = c(year(max(date)), month(max(date))),
    frequency = 12
  )

# See the result
potato_time_series
```

Task 9: Instructions

Wrap the time series preparation code into a function.

- Write a function, `create_price_time_series`, with an input, `prices`, that is a data frame of food prices.
    - Amend the potato price preparation code to work with `prices`.
    - Return the `ts` time series.
- Test the function on `pea_prices`.

-

Solution

```
# Wrap this code into a function
create_price_time_series <- function(prices) {
  prices_summarized <- prices %>%
    group_by(date) %>%
    summarize(median_price_rwf = median(price_rwf))

  prices_summarized %$%
    ts(
      median_price_rwf,
      start = c(year(min(date)), month(min(date))),
      end   = c(year(max(date)), month(max(date))),
```

```
      frequency = 12
    )
}

# Try the function on the pea data
pea_time_series <- create_price_time_series(pea_prices)
pea_time_series
```

Task 10: Instructions

Run and visualize a forecast of the potato prices.

- Load the `forecast` package.
- Call `forecast()` without arguments on `potato_time_series`.
- Visualize the forecast with the automatic plotter, `autoplot()`.
    - Set the `main` title to `"Potato price forecast"`.

-

Solution

```
# Load forecast
library(forecast)

# Forecast the potato time series
potato_price_forecast <- forecast(potato_time_series)

# View it
potato_price_forecast

# Plot the forecast
autoplot(potato_price_forecast, main = "Potato price forecast")
```

Task 11: Instructions

Wrap the forecasting code into a function.

- Write a function, `plot_price_forecast`, with two inputs: `time_series` is the time series of prices created by your data preparation, and `commodity` is a string naming the type of commodity.
    - Amend the potato forecasting code to use `time_series`.
    - Set the title to use the commodity you pass into the function.
- Test the function on `pea_time_series`, with commodity name `"Pea"`.

-

Solution

```
# Wrap the code into a function
plot_price_forecast <- function(time_series, commodity) {
  price_forecast <- forecast(time_series)
```

```
  autoplot(price_forecast, main = paste(commodity, "price forecast"))
}

# Try the function on the pea data
plot_price_forecast(pea_time_series, "Pea")
```

Task 12: Instructions

Rerun the whole analysis with `"Beans (dry)"`.

- Read and clean the data with `read_price_data()`.
- Plot price vs. time with `plot_price_vs_time()`.
- Create a price time series using `create_price_time_series()`.
- Plot the price forecast using `plot_price_forecast()`.

-

Solution

```
# Choose dry beans as the commodity
commodity <- "Beans (dry)"

# Read the price data
bean_prices <- read_price_data(commodity)

# Plot price vs. time
plot_price_vs_time(bean_prices, commodity)

# Create a price time series
bean_time_series <- create_price_time_series(bean_prices)

# Plot the price forecast
plot_price_forecast(bean_time_series, commodity)
```

Up and Down With the Kardashians

Task 1: Instructions

Load and inspect the data.

- Import `pandas` aliased as `pd`.
- Read the CSV file, `datasets/trends_kj_sisters.csv`, into a pandas DataFrame using the `read_csv()` function. Name the DataFrame `trends`.
- Inspect the data using the `head()` method.

Good to know

This Project provides the opportunity to apply the skills covered in DataCamp's Data Manipulation with pandas and Manipulating Time Series Data in Python and their prerequisite courses.

"Present day" for this Project was March 21st, 2019 so the monthly data spans from 2007-01-01 to 2019-03-01.

Helpful links specific to this task:

- pandas `read_csv()` function documentation
- Reading a flat file exercise in the pandas Foundations course

-

Solution

```
# Load pandas
import pandas as pd

# Read in dataset
trends = pd.read_csv('datasets/trends_kj_sisters.csv')

# Inspect data
trends.head()
```

Task 2: Instructions

Improve the column names.

- Set the `trends.columns` attribute to a six-item list with the following strings: "month", "kim", "khloe", "kourtney", "kendall", "kylie".
- Inspect the data using the `head()` method.

It is nice to have column names without spaces so columns can be accessed using the period convention, e.g., `trends.kim` instead of `trends['Kim Kardashian: (Worldwide)']`. There's a caveat with that, though. More on that in this Stack Overflow answer.

Helpful links:

- More information on the `columns` attribute of pandas DataFrames
- The lists chapter of the Intro to Python for Data Science course

-

Solution

```
# Make column names easier to work with
trends.columns = ['month', 'kim', 'khloe', 'kourtney', 'kendall', 'kylie']

# Inspect data
trends.head()
```

Task 3: Instructions

Inspect the data types.

- Print information about the `trends` DataFrame using the `info()` method. Look at the data types of the columns.

The `info()` method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

Helpful links:

- `info()` method documentation

-

Solution

```
# Inspect data types
trends.info()
```

Task 4: Instructions

Remove the "<" characters and cast the columns to integer.

- Use a `for` loop to loop through each column in `trends.columns`.

- Use an `if` statement to control the flow of the `for` loop to only proceed if "<" exists in the column's values. The `to_string()` method in conjunction with the `in` membership operator is handy.
- Remove the "<" character by replacing it with the empty string (`""`) using the `str.replace()` method.
- Cast the columns that previously had "<" characters to integer type using the pandas `to_numeric()` function.

Don't forget to overwrite the columns in the `trends` DataFrame when you're completing this task!

Helpful links:

- `for` loop exercises in the Intermediate Python for Data Science course
- `if` exercises in the Intermediate Python for Data Science course
- The basic dtypes of pandas
- `to_string()` method documentation
- `str.replace()` method documentation
- `to_numeric()` function documentation

-

Solution

```
# Loop through columns
for column in trends.columns:
    # Only modify columns that have the "<" sign
    if "<" in trends[column].to_string():
        # Remove "<" and convert dtype to integer
        trends[column] = trends[column].str.replace('<', '')
        trends[column] = pd.to_numeric(trends[column])

# Inspect data types and data
trends.info()
trends.head()
```

Task 5: Instructions

Cast the `month` column to the datetime.

- Cast the `month` column to type `datetime64[ns]` using the pandas `.to_datetime()` function.
- Inspect the data types of the columns in `trends` using the `info()` method.
- Inspect the data using the `head()` method.

This DataCamp tutorial gives a great background on why converting strings to dates as `datetime` objects is standard practice for working data scientists today.

Helpful links:

- The basic dtypes of pandas
- `to_datetime()` function documentation

-

Solution

```
# Convert month to type datetime
trends.month = pd.to_datetime(trends.month)

# Inspect data types and data
trends.info()
trends.head()
```

Task 6: Instructions

Set the DataFrame index to month.

- Set the index of `trends` to the `month` column using the `set_index()` method.

Helpful links:

- `set_index()` method documentation

-

Solution

```
# Set month as DataFrame index
trends = trends.set_index('month')

# Inspect the data
trends.head()
```

Task 7: Instructions

Plot search interest vs. month.

- Type `%matplotlib inline` to make plots show in the notebook.
- Call the `plot()` method on the DataFrame `trends` to plot search interest vs. month.

`%matplotlib` is a magic function. `%inline` is a backend provided by IPython. Together, they allow plots to appear in the notebook below code cells. More details on both of those in the links provided.

Helpful links:

- `plot()` method documentation
-

Solution

```
# Plot search interest vs. month
%matplotlib inline
trends.plot()
```

Task 8: Instructions

Plot search interest vs. month from January 2014 onward.

- Subset `trends` to include month data from January 2014 to March 2019 (the end of the dataset), then call the `plot()` method on the resulting DataFrame to plot search interest vs. those specific months.

This filtering pandas DataFrames on dates Stack Overflow answer is excellent.

Helpful links:

- `.loc` documentation
-

Solution

```
# Zoom in from January 2014
trends.loc['2014-01-01':].plot()
```

Task 9: Instructions

Plot a twelve-month rolling mean for search interest vs. month.

- Call the `rolling` method on `trends`, set the `window` parameter to 12 months, then call `mean()` on the resulting DataFrame. Then call the `plot()` method to plot.

Rolling means are also called moving averages. The linked Wikipedia article contains a detailed explanation.

Helpful links:

- `rolling()` method documentation
- `mean()` method documentation
- Rolling mean exercise in the pandas Foundations course

-

Solution

```
# Smooth the data with rolling means
trends.rolling(window=12).mean().plot()
```

Task 10: Instructions

Create columns for each family line then plot search interest vs. month for each.

- Add a column named `kardashian` to `trends` that contains the total search interest by month for Kim, Khloé, and Kourtney divided by three.
- Add a column named `jenner` to `trends` that contains the total search interest by month for Kendall and Kylie divided by two.
- Subset `trends` to include only the newly created `kardashian` and `jenner` columns, then plot search interest vs. month.

Helpful links:

- Adding new column to existing DataFrame in Python pandas Stack Overflow answer
- Selecting multiple columns in a pandas dataframe Stack Overflow answer

-

Solution

```
# Average search interest for each family line
trends['kardashian'] = (trends.kim + trends.khloe + trends.kourtney) / 3
trends['jenner'] = (trends.kendall + trends.kylie) / 2

# Plot average family line search interest vs. month
trends[['kardashian', 'jenner']].plot()
```

Where Are the Fishes?

Task 1: Instructions

Load the libraries and depth data.

- Load the following packages: `dplyr`, `readr`, `lubridate`, `ggplot2`, and `patchwork`.
- Read in the depth data, `datasets/bottom_line.csv`, using `read_csv()` and assign it to `bottom`. Use the correct format for `ping_date`, which is in month/day/4-digit year, and put all variables in lowercase.
- Use `glimpse()` to look at the structure and first few observations of the raw data.

This project was updated on December 16, 2019. If you started the project before that date, please click the circular arrow in the bottom-right corner of the screen to reset the project. If you would like to save your code, download your project before resetting it.

Good to know

In this project, you'll practice skills taught in Working with Data in the Tidyverse and Working with Dates and Times in R. You will join datasets, compute distances from latitude and longitude using a function from the `geosphere` package, and make simple visualizations with `ggplot2` and `patchwork`.

- tidyverse cheat sheet
- lubridate cheat sheet
- read_csv() documentation
- Date-Time Conversion Functions To And From Character

The plural of fish is fish. **Fishes** refers to multiple species of fish.

-

Solution

```
# Load the libraries
library(dplyr)
library(readr)
library(lubridate)
library(ggplot2)
library(patchwork)

# Read in the depth data
bottom <- read_csv("datasets/bottom_line.csv",
                   col_types = cols(Ping_date = col_datetime(format =
"%m/%d/%Y"))) %>%
          rename_all(tolower)
```

```
# Glimpse the data
glimpse(bottom)
```

Task 2: Instructions

Clean the bottom data.

- Filter the depth data to keep points where `position_status` equals 1, and select the columns: `ping_date`, `ping_time`, `latitude`, `longitude`, and `depth`.
- In the same pipe chain, create a new datetime column, `date_time`, by adding `ping_date` and `ping_time`.
- Use `glimpse()` to look at the structure and first few observations of the cleaned data.

`distance_between` is the distance (meters) between each GPS location (longitude and latitude). If you are curious about what is going on in `distHaversine()`, take a look at this Stack Overflow answer.

Helpful links:

- distHaversine()
- cumsum()

-

Solution

```
# Clean the bottom data
bottom_clean <- bottom %>%
  filter(position_status == 1)  %>%
  select(ping_date, ping_time, latitude, longitude, depth) %>%
  mutate(date_time = ping_date + ping_time,
        distance_between = c(0,
                             geosphere::distHaversine(cbind(longitude[-
n()], latitude[-n()]),
                                                      cbind(longitude[ -1],
latitude[ -1]))),
        distance_along = cumsum(distance_between))

# Inspect the data
glimpse(bottom_clean)
```

Task 3: Instructions

Let's plot the cleaned bottom data!

- Change the size of the plots for easier viewing. Code is given.

- Make a point plot of longitude (x) and latitude (y) and set the size of the points to 0.5.
- Make a point plot of distance along the track line (x) and depth (y) and set the size of the points to 0.5. Reverse the y-axis with `scale_y_reverse()`.
- With `patchwork`, you can arrange the plots side-by-side with `+`. Code is given.

By convention, the sea surface is at 0 meters. Because these depth data are positive, the y-axis needs to be reversed.

Helpful links:

- ggplot2: Position Scales for Continuous Data (x&y)
- patchwork blog

-

Solution

```
# Set the size of the plots
options(repr.plot.width = 7, repr.plot.height = 5)

# Plot the ship's track
p_ship_track <- ggplot(bottom_clean, aes(longitude, latitude)) +
               geom_point(size = 0.5) +
               labs(x = "Longitude", y = "Latitude")

# Plot the depth of the sea floor along the ship's track
p_bathymetry <- ggplot(bottom_clean, aes(distance_along, depth)) +
               geom_point(size = 0.5) +
               scale_y_reverse() +
               labs(x = "Distance along trackline (m)", y = "Depth (m)")

# Arrange the plots side by side
p_ship_track + p_bathymetry
```

Task 4: Instructions

Load and clean the acoustic data.

- Read in the acoustic data, `datasets/acoustic.csv`, using `read_csv()` and pipe it to `filter()` to remove any bad positional data (i.e. `Lon_M` not equal to 999.0). The format of Date_M has been set for you.
- Glimpse the new data to look at the structure and some values.

The acoustic data have three Longitude/Latitude positions per grid cell. `Lon_S`/`Lat_S` for the start of the grid cell. `Lon_M`/`Lat_M` for the mid-point of the grid cell. `Lon_E`/`Lat_E` for the endpoint of the grid cell.

Helpful links:

- Logical operators
- Date-Time Conversion Functions To And From Character

-

Solution

```
# Read in the acoustic data
acoustic <- read_csv("datasets/acoustic.csv",
                     col_types = cols(Date_M = col_datetime(format = "%Y%m
%d")))  %>%
  filter(Lon_M != 999.0)

# Glimpse the data
glimpse(acoustic)
```

Task 5: Instructions

Clean the acoustic data.

- Create a list of variables to keep called `variables_keep`.
- Use `select()` and its helper to keep the variables in `variables_keep`. Change the column names `Interval` to `Spatial_interval` and `Date_M` to `Date` with `rename()`. Filter to keep data from the first depth layer. Create the start (`Datetime_start`) and end (`Datetime_end`) timestamps by adding the date column and the correct time column.
- Glimpse the cleaned data.

`Date_M` is the date column, which you rename to `Date`. `Time_S` is the start time. `Time_E` is the time.

Helpful links:

- `select()` helper functions
- `rename()`
- lubridate

-

Solution

```
# Create a list of variables to keep
variables_keep <- c("Interval", "Layer", "Sv_mean", "Frequency",
                "Date_M", "Time_S", "Time_E", "Lat_M", "Lon_M")

# Select and filter the data
Sv_layer1 <- acoustic %>%
    select(one_of(variables_keep)) %>%
```

```
    rename(Spatial_interval = Interval, Date = Date_M) %>%
    filter(Layer == "1")  %>%
    mutate(Datetime_start = Date + Time_S,
           Datetime_end = Date + Time_E)  %>%
    arrange(Datetime_start)

# Glimpse the cleaned acoustic data
glimpse(Sv_layer1)
```

Task 6: Instructions

Clean the acoustic data.

- Create `Distance_between` and `Distance_along` using the appropriate functions. Look back at Task 2 if you need help.
- Replace -999.0s with `NA`, and create a time interval called `Time_interval` using the start and stop timestamps.
- Glimpse the cleaned data.

Unlike the bottom data, which is point data, the acoustic data are integrated into grid cells with latitudes and longitudes for the start, middle, and end of the grid cell. You're using the mid-points (`Lon_M`, `Lat_M`) to calculate the distance between each grid cell and assigning it to `Distance_betweeen`.

Helpful links:

- na_if()
- interval()

-

Solution

```
# Data prep for temporal interval join
Sv <- Sv_layer1 %>%
    mutate(Distance_between = c(0,
                                geosphere::distHaversine(cbind(Lon_M[-n()],
Lat_M[-n()]),
                                                         cbind(Lon_M[  -1],
Lat_M[  -1]))),
           Distance_along = cumsum(Distance_between)) %>%
    na_if(-999) %>%
    mutate(Time_interval = interval(Datetime_start, Datetime_end))

# Glimpse the data
glimpse(Sv)
```

Task 7: Instructions

Assign the correct spatial interval from `Sv` to each point in the clean bottom data.

- Examine `get_Interval_by_time()`.
- Create `trackline_interval` in `bottom_clean` using `map_dbl()`.
- Inspect the first 15 rows. Pay attention to the column `trackline_interval`.

`get_Interval_by_time()` is a function that assigns values of `Spatial_interval` from the acoustic data to points in the bottom data that fall within acoustic temporal intervals, `Time_interval`.

Helpful links:

- lubridate's %within%
- map_*
- DataCamp's Foundation fo Functional Programming with purrr

-

Solution

```
# Name the function
get_Interval_by_time <- function(bottom_data){
  res <- Sv$Spatial_interval[bottom_data %within% Sv$Time_interval]
  if(length(res)==0) return(NA)         # dealing with NAs
  return(res)
}

# Map the track line interval value to the bottom_clean data
bottom_spatial_interval_segments <- bottom_clean  %>%
    mutate(trackline_interval = purrr::map_dbl(date_time,
get_Interval_by_time))

# Inspect the first 15 rows
head(bottom_spatial_interval_segments, 15)
```

Task 8: Instructions

Summarize the mean depth for each track line interval.

- Group `bottom_spatial_interval_segments` by `trackline_interval` and summarize the mean depth for each interval.
- Join `bottom_intervals` to the acoustic data and create a new variable, `depth_plot`, that replaces each depth value greater than 250 meters with 250.
- Glimpse the new data set.

Helpful links:

- When to `ungroup()` Helpful links:
- ifelse()

-

Solution

```
# Group bottom_clean and calculate the mean depth
bottom_intervals <- bottom_spatial_interval_segments %>%
    group_by(trackline_interval) %>%
    summarize(depth_mean = mean(depth)) %>%
    ungroup()

# Join the bottom intervals data to the acoustic data
Sv_and_depth <- Sv %>%
  left_join(bottom_intervals, by = c("Spatial_interval" =
"trackline_interval")) %>%
  mutate(depth_plot = ifelse(depth_mean >= 250, 250, depth_mean))

# Glimpse the data
glimpse(Sv_and_depth)
```

Task 9: Instructions

Final plot using `Sv_and_depth`!

- In the top panel, use the line geometry from `ggplot2` to plot `Sv_mean` (y-axis) and distance along the track line (x-axis). Assign it to `Sv_mean_plot`.
- In the bottom panel, use the line geometry from `ggplot2` to plot `depth_plot` (y-axis) and distance along the track line (x-axis). Reverse the y-axis and assign the plot to `bathymetry`.
- With patchwork you can arrange the plot one over the other with `/`. Code is given.

All the data you need are in `Sv_and_depth`.

Helpful links:

- ggplot2: Position Scales for Continuous Data (x&y)
- Subscripts in axis labels
- patchwork blog

-

Solution

```
# Top panel
Sv_mean_plot <- ggplot(Sv_and_depth, aes(Distance_along, Sv_mean)) +
  geom_line() +
  labs(y=expression(mean~volume~backscatter~S[v]~(dB))) +
  theme(axis.title.x=element_blank())

# Bottom panel
bathymetry <- ggplot(Sv_and_depth, aes(Distance_along, depth_plot)) +
```

```
  geom_line(size = 0.5) +
  scale_y_reverse() +
  labs(x = "Distance along trackline (m)", y = "Depth (m)")

# Display the two panels in one figure
Sv_mean_plot / bathymetry
```

Task 10: Instructions

What section of the track line has the most fish per 200 m x 250 m grid cell?

- Using a character string, indicate which section of the track line has the most fish.
  - Options: Shelf, Shelf Break, Offshore

Congratulations on finishing the project!

-

Solution

```
# Where do you think the fish are along this track line?

# Options: Shelf, Shelf Break, Offshore

(where_are_the_fishes <- "Shelf")
```

Clustering Heart Disease Patient Data

Task 1: Instructions

Start by loading and exploring the data.

- Read in the dataset located in `"datasets/heart_disease_patients.csv"` into a variable called `heart_disease`.
- Print out the first ten rows of the data using `head()`, and check that all variables are a form of numeric data (integer or double).

Good to know

This project works with K-means and hierarchical clustering algorithms. We recommend that you take the following courses before starting this project: Introduction to Data Visualization with ggplot2 and Unsupervised Learning in R.

Helpful links throughout the project:

- `kmeans()` documentation
- `hclust()` function documentation
- ggplot2 package documentation

You can reset the project by clicking the circular arrow in the bottom-right corner of the screen if you experience odd behavior. Resetting the project will also discard all the code you have written so be sure to save it offline first.

-

Solution

```
# Load the data
heart_disease <- read.csv("datasets/heart_disease_patients.csv")

# Print the first ten rows of the dataset
head(heart_disease, n = 10)
```

Task 2: Instructions

Check if the data should be scaled before clustering.

- Look at the distributions of the variables in `heart_disease` using `summary()`.
- Remove the `id` variable from the data set.
- Scale the data and save it in a new data frame, `scaled`.
- Look at the distributions of the variables in `scaled` using `summary()`.

Helpful links:

- `summary()` function documentation
- `scale()` function documentation

-

Solution

```
# Evidence that the data should be scaled?
summary(heart_disease)

# Remove id
heart_disease <- heart_disease[ , !(names(heart_disease) %in% c("id"))]

# Scaling data and saving as a data frame
scaled <- scale(heart_disease)

# What do the data look like now?
summary(scaled)
```

Task 3: Instructions

Run an iteration of the k-means algorithm.

- Set the seed to the number 10 using `set.seed()`.
- Define the number of clusters to be five and save as `k`.
- Run the k-means algorithm on the `scaled` data where `centers` is the number of clusters and `nstart` is one. Name the returned k-means object `first_clust`.
- Find the number of patients in each cluster using the `size` attribute of the object returned by the `kmeans()` function.

Helpful links:

- `kmeans()` function documentation

-

Solution

```
# Set the seed so that results are reproducible
seed_val <- 10
set.seed(seed_val)

# Select a number of clusters
k <- 5

# Run the k-means algorithm
```

```
first_clust <- kmeans(scaled, centers = k, nstart = 1)

# How many patients are in each cluster?
first_clust$size
```

Task 4: Instructions

Run another k-means algorithm.

- Set the random seed to 38.
- Run another k-means algorithm using a random initialization of five clusters.
- Find the number of patients in each cluster.

Helpful links:

- `kmeans()` function [documentation]

-

Solution

```
# Set the seed
seed_val <- 38
set.seed(seed_val)

# Select a number of clusters and run the k-means algorithm
k <- 5
second_clust <- kmeans(scaled, centers = k, nstart = 1)

# How many patients are in each cluster?
second_clust$size
```

Task 5: Instructions

Create visualizations to evaluate the stability of the k-means algorithm on the patient data.

- Add columns to the `heart_disease` data containing the cluster assignments for each iteration. Name these columns `"first_clust"` and `"second_clust"`.
- Load the package `ggplot2` using `library()`.
- Create a scatter plot of `x = age`, `y = chol` for the first iteration of the clustering algorithm, and color code the points by cluster assignment.
- Create a scatter plot of `x = age`, `y = chol` for the second iteration of the clustering algorithm, and color code the points by cluster assignment.

To color code the points by cluster assignment, convert the cluster assignments to factors with `as.factor()`.

Helpful links:

- ggplot2 `geom_point()` function documentation
-

Solution

```
# Add cluster assignments to the data
heart_disease["first_clust"] <- first_clust$cluster
heart_disease["second_clust"] <- second_clust$cluster

# Load ggplot2
library(ggplot2)

# Create and print the plot of age and chol for the first clustering
algorithm
plot_one <- ggplot(heart_disease, aes(x=age, y=chol,
color=as.factor(first_clust))) +
  geom_point()
plot_one

# Create and print the plot of age and chol for the second clustering
algorithm
plot_two <- ggplot(heart_disease, aes(x=age, y=chol,
color=as.factor(second_clust))) +
  geom_point()
plot_two
```

Task 6: Instructions

Write the code for hierarchical clustering with a complete linkage function.

- Run the hierarchical clustering algorithm on `scaled` with `method = "complete"` to use the complete linkage function. You will need to calculate the distance matrix for the data using the `dist()` function.
- Plot the dendrogram of the algorithm run using `plot()`.
- Get the cluster assignments for five clusters and save as `hc_1_assign` using the function `cutree()`.

Helpful links:

- The `dist()` function calculates the distance matrix between rows in a data matrix. For more information see the documentation.
- `hclust()` function documentation
- `cutree()` function documentation
-

Solution

```
# Execute hierarchical clustering with complete linkage
hier_clust_1 <- hclust(dist(scaled), method = "complete")

# Print the dendrogram
plot(hier_clust_1)

# Get cluster assignments based on number of selected clusters
hc_1_assign <- cutree(hier_clust_1, 5)
```

Task 7: Instructions

Execute another iteration of the hierarchical clustering algorithm.

- Run the hierarchical clustering algorithm with `method = "single"`. You will again need to calculate the distance matrix for the data using the `dist()` function.
- Plot the dendrogram of the algorithm run using `plot()`.
- Use `cutree()` to get the cluster assignments for five clusters and save the output as `hc_2_assign`.

Helpful links:

- `hclust()` function [documentation]

-

Solution

```
# Execute hierarchical clustering with single linkage
hier_clust_2 <- hclust(dist(scaled), method = "single")

# Print the dendrogram
plot(hier_clust_2)

# Get cluster assignments based on number of selected clusters
hc_2_assign <- cutree(hier_clust_2, 5)
```

Task 8: Instructions

Write the code to examine the results of the hierarchical clustering algorithm.

- Look at the dendrogarams for both the single and complete linkages. Recalling that the doctors want groups with multiple patients, choose either `hc_1_assign` or `hc_2_assign` and add it to `heart_disease` in a column called `hc_clust`.
- Remove the categorical variables in `heart_disease` to create `hd_simple`.
- Calculate the mean and standard deviation of each variable for each cluster assignment by aggregating on `hc_clust`. Use the `do.call` function together with the `mean` and `sd` functions for these calculations.

Helpful links:

- `do.call()` documentation
- `aggregate()` documentation

-

Solution

```
# Add assignment of chosen hierarchical linkage
heart_disease["hc_clust"] <- hc_1_assign

# Remove the sex, first_clust, and second_clust variables
hd_simple <- heart_disease[, !(names(heart_disease) %in% c("sex",
"first_clust", "second_clust"))]

# Get the mean and standard deviation summary statistics
clust_summary <- do.call(data.frame, aggregate(. ~hc_clust, data =
hd_simple, function(x) c(avg = mean(x), sd = sd(x))))
clust_summary
```

Task 9: Instructions

Using the `heart_disease` data, create visualizations from the hierarchical clustering algorithm.

- Create a scatter plot where `x = age` and `y = chol`. Color code the points by cluster assignment.
- Create a scatter plot where `x = oldpeak` and `y = trestbps`. Color code the points by cluster assignment.

To color code the points by cluster assignment, convert the cluster assignments to factors with `as.factor()`.

Helpful links:

- ggplot2 package documentation

-

Solution

```
# Plot age and chol
plot_one <- ggplot(heart_disease, aes(x = age, y = chol,
                                      color = as.factor(hc_clust))) +
  geom_point()
plot_one
```

```
# Plot oldpeak and trestbps
plot_two <- ggplot(heart_disease, aes(x = oldpeak, y = trestbps,
                                      color = as.factor(hc_clust))) +
  geom_point()
plot_two
```

Task 10: Instructions

Determine if any of the algorithms show promise for grouping patients.

- For each of the algorithms (k-means, hierarchical with complete linkage, hierarchical with single linkage), determine if you think you should spend more time exploring the patient clustering from that algorithm by assigning `TRUE` or `FALSE` to each object.

Congratulations on completing the project and practicing your clustering skills!

-

Solution

```
# Add TRUE if the algorithm shows promise, add FALSE if it does not
explore_kmeans <- FALSE
explore_hierarch_complete <- TRUE
explore_hierarch_single <- FALSE
```

Naïve Bees: Deep Learning with Images

Task 1: Instructions

Import the Python libraries with which you will work.

- Import `keras`, the deep learning library you'll be using.
- Import the function `Sequential` from the `models` module of `keras`. This is the model type you'll use.
- Import the functions `Dense`, `Dropout`, `Flatten`, `Conv2D`, `MaxPooling2D` from the `layers` module of `keras`. These will form the different layers of your convolutional neural network.

Good to know

Welcome to the third project in a series on working with image data. You will be working through a Driven Data Competition to identify Honey Bees and Bumble Bees given an image of these insects! To learn more about the background, you can explore the competition page.

For this project, the documentation for keras, scikit-learn, scikit-image, and numpy will be helpful resources. For more information about bees, see the BeeSpotter project or the DrivenData competition.

The recommended prerequisites for this project are Advanced Deep Learning with Keras in Python, Introduction to Data Visualization with Python, Naïve Bees: Image Loading and Processing, and Naïve Bees: Predict Species from Images.

-

Solution

```
import pickle
from pathlib import Path
from skimage import io

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# import keras library
import keras
```

```
# import Sequential from the keras models module
from keras.models import Sequential

# import Dense, Dropout, Flatten, Conv2D, MaxPooling2D from the keras
layers module
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
```

Task 2: Instructions

Load the DataFrame of labels and image names, explore the dataset, then assign image labels to `y`.

- Using `read_csv` function from pandas, load the `labels.csv` file which lives in the `datasets` folder. Be sure to set `index_col=0` so that the images names are loaded as the index.
- Print the value counts of `genus` in the `labels` DataFrame to show that the dataset is perfectly balanced between the two classes.
- Assign the `genus` column values (the array of image labels) to `y`.

-

Solution

```
# load labels.csv from datasets folder using pandas
labels = pd.read_csv('datasets/labels.csv', index_col=0)

# print value counts for genus
print(labels.genus.value_counts())

# assign the genus label values to y
y = labels.genus.values
```

Task 3: Instructions

Load the first image from your DataFrame and explore its shape and RGB values.

- Load the first image from the labels DataFrame index using `io.imread`, the image loading function from scikit-image and assign it to `example_image`.
- Display `example_image` using `plt.imshow()`.
- Print the shape of `example_image` to see that it is 50 by 50 pixels and has 3 channels.
- Print the R, G, and B values for the top left pixel of the `example_image`. Recall that the image has shape (X, Y, Z) and that the that the X and Y coordinates for this pixel are (0, 0).

You can explore different value combinations with this RGB calculator.

-

Solution

```
# load an image and explore
example_image = io.imread('datasets/{}.jpg'.format(labels.index[0]))

# show image
plt.imshow(example_image)

# print shape
print('Image has shape:', example_image.shape)

# print color channel values for top left pixel
print('RGB values for the top left pixel are:', example_image[0, 0, :])
```

Task 4: Instructions

Normalize each feature (i.e. channel) of each image by iterating over the channels of each image in a for loop. Then stack the resulting arrays into a single matrix and assign it to X.

- Assign the `StandardScaler()` object to `ss`.
- Within the for loop that iterates over each channel of an image, call `ss.fit_transform` on each channel.
- Use `np.array()` to stack the `image_list` (a list of normalized image arrays) into an array that contains all of the images in the dataset.
- Print the shape of `X`.

-

Solution

```
# initialize standard scaler
ss = StandardScaler()

image_list = []
for i in labels.index:
    # load image
    img = io.imread('datasets/{}.jpg'.format(i)).astype(np.float64)

    # for each channel, apply standard scaler's fit_transform method
    for channel in range(img.shape[2]):
        img[:, :, channel] = ss.fit_transform(img[:, :, channel])

    # append to list of all images
    image_list.append(img)

# convert image list to single array
X = np.array(image_list)

# print shape of X
print(X.shape)
```

Task 5: Instructions

Split the data into train, test, and evaluation sets.

- Use `train_test_split` to split out 20% of the `X` and `y` data into validation sets by setting `test_size` equal to 0.2.
- Split the remaining data (`x_interim` and `y_interim`) into `x_train`, `x_test`, `y_train`, `y_test` so that 40% of the data goes into the test set. Be sure to set `random_state=52` to ensure consistent results.
- Print the shape of `x_train`.
- Print the number of samples in `x_train`.

-

Solution

```
# split out evaluation sets (x_eval and y_eval)
x_interim, x_eval, y_interim, y_eval = train_test_split(X, y,
test_size=0.2, random_state=52)

# split remaining data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x_interim, y_interim,
test_size=0.4, random_state=52)

# examine number of samples in train, test, and validation sets
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_eval.shape[0], 'eval samples')
```

Task 6: Instructions

Specify the number of classes in the model and then build the first two layers of the neural network.

- Set `num_classes` equal to `1`. Since your model is trying to predict whether an image is of a bumble bee or a honey bee (i.e. not a bumble bee), you can frame this as a binary classification problem.
- Define `model` as `Sequential()` to initialize the model.
- There is already one 2D convolutional layer in the code with 32 filters. Add another 2D convolutional layer (`Conv2D`), this time with 64 filters. It should have the same `kernel_size` and `activation` specifications as the first 2D convolutional layer.

The input shape in the first layer refers to the dimensions of the images that get passed in. The input shape does not need to specified after the first layer as the following layers can do automatic shape inference.

-

Solution

```
# set model constants
num_classes = 1

# define model as Sequential
model = Sequential()

# first convolutional layer with 32 filters
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=(50, 50, 3)))

# add a second 2D convolutional layer with 64 filters
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

Task 7: Instructions

Finish building the CNN and then print the model summary.

- Add a `MaxPooling2D` layer with `pool_size=(2, 2)`.
- Add a second `Dropout` layer with a rate of `0.5`.
- For the final `Dense` layer which generates predictions, pass in your previously specified `num_classes` as the first parameter. Set `sigmoid` as the activation function since this is a binary classification problem.
- Show the model summary using `model.summary()`.

Convolutional layers share weights across pixels, unlike fully connected layers.

-

Solution

```
# reduce dimensionality through max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))

# third convolutional layer with 64 filters
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
# add dropout to prevent over fitting
model.add(Dropout(0.25))
# necessary flatten step preceeding dense layer
model.add(Flatten())
# fully connected layer
model.add(Dense(128, activation='relu'))

# add additional dropout to prevent overfitting
model.add(Dropout(0.5))
```

```
# prediction layers
model.add(Dense(num_classes, activation='sigmoid', name='preds'))

# show model summary
model.summary()
```

Task 8: Instructions

Compile the model and mock train it on a subset of the data for five epochs.

- In the compile function, set `keras.losses.binary_crossentropy` as the loss and set `accuracy` as the metric.
- In the fit function which trains the model, set epochs equal to `5`.

-

Solution

```
model.compile(
    # set the loss as binary_crossentropy
    loss=keras.losses.binary_crossentropy,
    # set the optimizer as stochastic gradient descent
    optimizer=keras.optimizers.SGD(lr=0.001),
    # set the metric as accuracy
    metrics=['accuracy']
)

# mock-train the model using the first ten observations of the train and
test sets
model.fit(
    x_train[:10, :, :, :],
    y_train[:10],
    epochs=5,
    verbose=1,
    validation_data=(x_test[:10, :, :, :], y_test[:10])
)
```

Task 9: Instructions

Load the pre-trained model and calculate the loss and accuracy for the holdout set.

- Load the pretrained model using `load_model` from the `keras.models` module and assign it to `pretrained_cnn`.
- Evaluate the pretrained model on the holdout set (`x_eval` and `y_eval`) using the `.evaluate` method. Assign the resulting tuple to `eval_score`.
- Print the loss for the holdout set.
- Print the accuracy for the holdout set.

You can see that the model generalizes quite well as the accuracy is similar for the test set and holdout set: 0.66 for data the model has seen (the test set) and 0.65 for data the model has not seen (the holdout set).

Note: the kernel for this notebook may die if you load the pretrained model too many times. In that case, save the notebook and reload the project. If you still run into issues, you can always click the reset project arrow next to the "Check Project" button (warning: you will lose all of your code after clicking this button).

-

Solution

```
# load pre-trained model
pretrained_cnn = keras.models.load_model('datasets/pretrained_model.h5')

# evaluate model on test set
score = pretrained_cnn.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

print("")

# evaluate model on holdout set
eval_score = pretrained_cnn.evaluate(x_eval, y_eval, verbose=0)
# print loss score
print('Eval loss:', eval_score[0])
# print accuracy score
print('Eval accuracy:', eval_score[1])
```

Task 10: Instructions

Load the model history and plot the validation accuracy and loss over the training period.

- Print the keys for the `pretrained_cnn_history` dictionary. This shows that you can access loss and accuracy scores for the test set and holdout set.
- Plot the validation accuracy with `pretrained_cnn_history['val_acc']`.
- Plot the validation loss.

-

Solution

```
# load history
with open('datasets/model_history.pkl', 'rb') as f:
    pretrained_cnn_history = pickle.load(f)

# print keys for pretrained_cnn_history dict
print(pretrained_cnn_history.keys())

fig = plt.figure(1)
```

```
plt.subplot(211)
# plot the validation accuracy
plt.plot(pretrained_cnn_history['val_acc'])
plt.title('Validation accuracy and loss')
plt.ylabel('Accuracy')
plt.subplot(212)
# plot the validation loss
plt.plot(pretrained_cnn_history['val_loss'], 'r')
plt.xlabel('Epoch')
plt.ylabel('Loss value');
```

Task 11: Instructions

Finally, you'll get the predicted probabilities and classes for each image in the validation set.

- Use the `predict` method from our `pretrained_cnn` model to get the probabilities for `x_eval` and assign them to `y_proba`.
- Print the first five probabilities in the `y_proba` list you just created.
- Use the `predict_classes` method from your `pretrained_cnn` model to get the predicted classes for `x_eval` and assign them to `y_pred`. Note that those with a probability greater than 0.5 were assigned class 1, a bumble bee.

Transfer learning will be explored in a future Naïve Bees DataCamp project and will be linked here when launched.

-

Solution

```
# predicted probabilities for x_eval
y_proba = pretrained_cnn.predict(x_eval)

print("First five probabilities:")
print(y_proba[:5])
print("")

# predicted classes for x_eval
y_pred = pretrained_cnn.predict_classes(x_eval)

print("First five class predictions:")
print(y_pred[:5])
print("")
```

Predicting Credit Card Approvals

Task 1: Instructions

Load and look at the dataset.

- Import the pandas library under the alias `pd`.
- Load the dataset, `"datasets/cc_approvals.data"`, into a pandas DataFrame called `cc_apps`. Set the `header` argument to `None`.
- Print the first 5 rows of `cc_apps` using the `head()` method.

Good to know

For this project, it is recommended that you know basic Python programming, the `pandas` and `numpy` packages, some data preprocessing, and a little bit of machine learning. Here are some resources that may be helpful throughout the project:

- For a quick introduction to Python:
  - DataCamp's Intro to Python for Data Science course
- For learning the basics of the `pandas` and `numpy` packages:
  - Data Manipulation with pandas
  - pandas Cheatsheet
  - NumPy Cheat Sheet
- For data preprocessing:
  - Preprocessing in Data Science (Part 1)
  - Preprocessing in Data Science (Part 2)
  - Preprocessing in Data Science (Part 3)
- For machine learning:
  - Google's Machine Learning Crash Course
  - Supervised Learning with scikit-learn

Apart from the above, we encourage you to use your preferred search engine to find other useful resources.

-

Solution

```
# Import pandas
import pandas as pd

# Load dataset
cc_apps = pd.read_csv("datasets/cc_approvals.data", header=None)

# Inspect data
cc_apps.head()
```

Task 2: Instructions

Inspect the structure, numerical summary, and specific rows of the dataset.

- Extract the summary statistics of the data using the `describe()` method of `cc_apps`.
- Use the `info()` method of `cc_apps` to get more information about the DataFrame.
- Print the last 17 rows of `cc_apps` using the `tail()` method to display missing values.

Helpful links:

- pandas `tail()` method documentation
-

Solution

```
# Print summary statistics
cc_apps_description = cc_apps.describe()
print(cc_apps_description)

print("\n")

# Print DataFrame information
cc_apps_info = cc_apps.info()
print(cc_apps_info)

print("\n")

# Inspect missing values in the dataset
cc_apps.tail(17)
```

Task 3: Instructions

Inspect the missing values in the dataset and replace the question marks with NaN.

- Import the numpy library under the alias `np`.
- Print the last 17 rows of the dataset.
- Replace the '?'s with NaNs using the `replace()` method.
- Print the last 17 rows of `cc_apps` using the `tail()` method to confirm that the `replace()` method performed as expected.

Helpful links:

- pandas `replace()` method documentation

- NumPy data types for special values
-

Solution

```
# Import numpy
import numpy as np

# Inspect missing values in the dataset
print(cc_apps.tail(17))

# Replace the '?'s with NaN
cc_apps = cc_apps.replace('?', np.nan)

# Inspect the missing values again
cc_apps.tail(17)
```

Task 4: Instructions

Impute the NaN values with the mean imputation approach.

- For the numeric columns, impute the missing values (NaNs) with pandas method `fillna()`.
- Verify if the `fillna()` method performed as expected by printing the total number of NaNs in each column.

Remember that you have already marked all the question marks as NaNs. pandas provides `fillna()` to help you impute missing values with different strategies, mean imputation being one of them. pandas also has a `mean()` method to calculate the mean of a DataFrame. As your dataset contains both numeric and non-numeric data, for this task you will only impute the missing values (NaNs) present in the columns having numeric data-types (columns 2, 7, 10 and 14).

Helpful links:

- mean imputation tutorial
- pandas `fillna()` method documentation
- pandas `mean()` method documentation
- pandas `isnull()` method documentation
-

Solution

```
# Impute the missing values with mean imputation
cc_apps.fillna(cc_apps.mean(), inplace=True)

# Count the number of NaNs in the dataset and print the counts to verify
print(cc_apps.isnull().sum())
```

Task 5: Instructions

Impute the missing values in the non-numeric columns.

- Iterate over each column of `cc_apps` using a `for` loop.
- Check if the data-type of the column is of `object` type by using the `dtypes` keyword.
- Using the `fillna()` method, impute the column's missing values with the most frequent value of that column with the `value_counts()` method and index attribute and assign it to `cc_apps`.
- Finally, verify if there are any more missing values in the dataset that are left to be imputed by printing the total number of NaNs in each column.

The column names of a pandas DataFrame can be accessed using `columns` attribute. The `dtypes` attribute provides the data type. In this part, `object` is the data type that you should be concerned about. The `value_counts()` method returns the frequency distribution of each value in the column, and the `index` attribute can then be used to get the most frequent value.

Helpful links:

- pandas `value_counts()` method documentation
- Accessing the `index` attribute in a tutorial
- Method chaining with pandas tutorial

-

Solution

```
# Iterate over each column of cc_apps
for col in cc_apps.columns:
    # Check if the column is of object type
    if cc_apps[col].dtypes == 'object':
        # Impute with the most frequent value
        cc_apps = cc_apps.fillna(cc_apps[col].value_counts().index[0])

# Count the number of NaNs in the dataset and print the counts to verify
print(cc_apps.isnull().sum())
```

Task 6: Instructions

Convert the non-numeric values to numeric.

- Import the `LabelEncoder` class from `sklearn.preprocessing` module.
- Instantiate `LabelEncoder()` into a variable `le`.
- Iterate over all the **values** of each column `cc_apps` and check their data types using a `for` loop.
- If the data type is found to be of `object` type, label encode it to transform into numeric (such as `int64`) type.

The values of each column a pandas DataFrame can be accessed using `columns` and `values` attributes consecutively. The `dtypes` attribute provides the data type. In this part, `object` is the data type that you should be concerned about.

Helpful links:

- Checking data types of the columns in a DataFrame Stack Overflow answer
- sklearn `LabelEncoder` class documentation
-

Solution

```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Instantiate LabelEncoder
le=LabelEncoder()

# Iterate over all the values of each column and extract their dtypes
for col in cc_apps.columns.values:
    # Compare if the dtype is object
    if cc_apps[col].dtypes=='object':
    # Use LabelEncoder to do the numeric transformation
        cc_apps[col]=le.fit_transform(cc_apps[col])
```

Task 7: Instructions

Split the preprocessed dataset into train and test sets.

- Import `train_test_split` from the `sklearn.model_selection` module.
- Drop features 11 and 13 using the `drop()` method and convert the DataFrame to a NumPy array using `.values`.
- Segregate the features and labels into `X` and `y` (the column with index 13 is the label column).
- Using the `train_test_split()` method, split the data into train and test sets with a split ratio of 33% (`test_size` argument) and set the `random_state` argument to 42.

A NumPy array can be segregated using array slicing. Before slicing, take note of the total number of columns that should be present in the array after dropping features 11 and 13.

Setting `random_state` ensures the dataset is split with same sets of instances every time the code is run.

Helpful links:

- pandas `drop()` method documentation
- NumPy indexing and slicing tutorial
- sklearn `train_test_split()` method documentation

-

Solution

```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Drop the features 11 and 13 and convert the DataFrame to a NumPy array
cc_apps = cc_apps.drop([11, 13], axis=1)
cc_apps = cc_apps.values

# Segregate features and labels into separate variables
X,y = cc_apps[:,0:13] , cc_apps[:,13]

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X
                                                    ,
                                                    y,
                                                    test_size=0.33,
                                                    random_state=42)
```

Task 8: Instructions

Drop `DriversLicense` and `ZipCode` features and rescale the data.

- Import the `MinMaxScaler` class from the `sklearn.preprocessing` module.
- Instantiate `MinMaxScaler` class in a variable called `scaler` with the `feature_range` parameter set to `(0,1)`.
- Fit the `scaler` to `X_train` and transform the data, assigning the result to `rescaledX_train`.
- Use the `scaler` to transform `X_test`, assigning the result to `rescaledX_test`.

When a dataset has varying ranges as in this credit card approvals dataset, one a small change in a particular feature may not have a significant effect on the other feature, which can cause a lot of problems when predictive modeling.

Helpful links:

- sklearn's `MinMaxScaler` class documentation

-

Solution

```
# Import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
```

```
# Instantiate MinMaxScaler and use it to rescale X_train and X_test
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX_train = scaler.fit_transform(X_train)
rescaledX_test = scaler.transform(X_test)
```

Task 9: Instructions

Fit a `LogisticRegression` classifier with `rescaledX_train` and `y_train`.

- Import `LogisticRegression` from the `sklearn.linear_model` module.
- Instantiate `LogisticRegression` into a variable named `logreg` with default values.
- Fit `rescaledX_train` and `y_train` to `logreg` using the `fit()` method.

If a quick refresher on logistic regression's working mechanism is needed, check out this tutorial.

Helpful links:

- sklearn Logistic Regression documentation
-

Solution

```
# Import LogisticRegression
from sklearn.linear_model import LogisticRegression

# Instantiate a LogisticRegression classifier with default parameter values
logreg = LogisticRegression()

# Fit logreg to the train set
logreg.fit(rescaledX_train,y_train)
```

Task 10: Instructions

Make predictions and evaluate performance.

- Import `confusion_matrix()` from `sklearn.metrics` module.
- Use `predict()` on `rescaledX_test` (which contains instances of the dataset that `logreg` has not seen until now) and store the predictions in a variable named `y_pred`.
- Print the accuracy score of `logreg` using the `score()`. Don't forget to pass `rescaledX_test` and `y_test` to the `score()` method.
- Call `confusion_matrix()` with `y_test` and `y_pred` to print the confusion matrix.

Helpful links:

- sklearn confusion matrix documentation
-

Solution

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

# Use logreg to predict instances from the test set and store it
y_pred = logreg.predict(rescaledX_test)

# Get the accuracy score of logreg model and print it
print("Accuracy of logistic regression classifier: ",
logreg.score(rescaledX_test,y_test))

# Print the confusion matrix of the logreg model
confusion_matrix(y_test,y_pred)
```

Task 11: Instructions

Define the grid of parameter values for which grid searching is to be performed.

- Import `GridSearchCV` from the `sklearn.model_selection` module.
- Define the grid of values for `tol` and `max_iter` parameters into `tol` and `max_iter` lists respectively.
- For `tol`, define the list with values 0.01, 0.001 and 0.0001. For `max_iter`, define the list with values 100, 150 and 200.
- Using the `dict()` method, create a dictionary where `tol` and `max_iter` are keys, and the lists of their values are the corresponding values. Name this dictionary as `param_grid`.

Grid search can be very exhaustive if the model is very complex and the dataset is extremely large. Luckily, that is not the case for this project.

-

Solution

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Define the grid of values for tol and max_iter
tol = [0.01, 0.001 ,0.0001]
max_iter = [100, 150, 200]

# Create a dictionary where tol and max_iter are keys and the lists of
their values are the corresponding values
param_grid = dict(tol=tol, max_iter=max_iter)
```

Task 12: Instructions

Find the best score and best parameters for the model using grid search.

- Instantiate `GridSearchCV()` with the attributes set as `estimator = logreg`, `param_grid = param_grid` and `cv = 5` and store this instance in `grid_model` variable.
- Use `scaler` (which you created in Task-8) rescale `X` and assign it to `rescaledX`.
- Fit `rescaledX` and `y` to `grid_model` and store the results in `grid_model_result`.
- Call the `best_score_` and `best_params_` attributes on the `grid_model_result` variable, then print both.

Grid searching is a process of finding an optimal set of values for the parameters of a certain machine learning model. This is often known as hyperparameter optimization which is an active area of research. Note that, here we have used the word parameters and hyperparameters interchangeably, but they are not exactly the same.

Helpful links:

- Hyperparameter Optimization in Machine Learning Models tutorial
-

Solution

```
# Instantiate GridSearchCV with the required parameters
grid_model = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=5)

# Use scaler to rescale X and assign it to rescaledX
rescaledX = scaler.fit_transform(X)

# Fit grid_model to the data
grid_model_result = grid_model.fit(rescaledX, y)

# Summarize results
best_score, best_params = grid_model_result.best_score_,
grid_model_result.best_params_
print("Best: %f using %s" % (best_score, best_params))
```

Going Down to South Park: A Text Analysis

Task 1: Instructions

Warning: the dataset used in this project contains explicit language.

Load the datasets and take a look at the first few observations.

- Load `sp_lines.csv` and `sp_ratings.csv` datasets using `read_csv()` from the `datasets` directory.
- Examine the last few observations of `sp_lines` and `sp_ratings`.

Good to know

This project lets you apply the skills from Introduction to the Tidyverse, including filtering, grouping and summarizing data. You will also apply some skills from Sentiment Analysis in R. Lastly, there will also be some visualization in the project, so familiarity with ggplot2 will also be useful, such as the skills taught in Intermediate Data Visualization with ggplot2. Completing these courses will be helpful throughout the project.

Helpful links:

- tidyverse cheat sheet
- ggplot2 cheat sheet
- tidytext book
- sweary, an R package with a database of swear words from different languages

-

Solution

```
# Load libraries
library(dplyr)
library(readr)
library(tidytext)
library(sweary)

# Load datasets
sp_lines <- read_csv("datasets/sp_lines.csv")
sp_ratings <- read_csv("datasets/sp_ratings.csv")

# Take a look at the last six observations
tail(sp_lines)
tail(sp_ratings)
```

Task 2: Instructions

Create the `sp_words` data frame with useful columns.

- Join `sp_lines` and `sp_ratings` into `sp`.
- Create `sp_words` by unnesting lines to words. Use `unnest_tokens()` from `tidytext` to break up the lines in `text` and create the column, `word`. Leave out all stop words using the appropriate `_join()` function, and create `word_stem` and `swear_word` columns.
- View the last six observations.

`en_swear_words` is a data frame of English swear words. `swear_word` (in `sp_words`) is `TRUE` when `word` is a swear word **OR** if `word_stem` is the stem of a swear word.

To stem a line, use the `wordStem()` from the `SnowballC` package.

The AFINN lexicon was downloaded from the `tidytext` package and stored in the `datasets` folder.

Helpful links:

- dplyr joins cheat sheet
- `unnest_tokens()` detailed info
- `wordStem()` usage
- `%in%` operator in R

-

Solution

```
# Load english swear words
en_swear_words <- sweary::get_swearwords("en") %>%
    mutate(stem = SnowballC::wordStem(word))

# Load the AFINN lexicon
afinn  <- read_rds("datasets/afinn.rds")

# Join lines with episode ratings
sp <- inner_join(sp_lines, sp_ratings)

# Unnest lines to words, leave out stop words and add a
# swear_word logical column
sp_words <- sp %>%
    unnest_tokens(word, text) %>%
    anti_join(stop_words) %>%
    left_join(afinn) %>%
    mutate(word_stem = SnowballC::wordStem(word),
        swear_word = word %in% en_swear_words$word | word_stem %in%
en_swear_words$stem)

# View the last six observations
tail(sp_words)
```

Task 3: Instructions

Create a `by_episode` data frame.

- Group `sp_words` by `episode_name`, `rating`, and `episode_order`, and summarize the groups to create `swear_word_ratio` and `avg_sentiment_score`.
- Examine the last few six of `by_episode`.
- Print information about the episode with the highest `swear_word_ratio`.

Calling `sum()` on a logical vector returns the number of `TRUE` occurrences while `n()` returns the number of observations in a group.

The sentiment score of an episode is the *mean* sentiment score of all the words in the episode. `value` contains a lot of `NA` values because not every word is in the sentiment lexicon.

-

Solution

```
# Group by and summarize data by episode
by_episode <- sp_words %>%
    group_by(episode_name, rating, episode_order) %>%
    summarize(
        swear_word_ratio = sum(swear_word) / n(),
        sentiment_score = mean(value, na.rm = TRUE)
    ) %>%
    arrange(episode_order)

# Examine the last few rows of by_episode
tail(by_episode)

# What is the naughtiest episode?
( naughtiest <- by_episode[which.max(by_episode$swear_word_ratio), ] )
```

Task 4: Instructions

Create a column chart of mean episode sentiment scores.

- Set the minimal theme that will be used for this and all future plots.
- Plot `sentiment_score` for each episode. Use `geom_col()` to display episode sentiments. Use `geom_smooth()` to see the trend.

Helpful links:

- More about `ggplot2`'s basic themes

-

Solution

```
# Load the ggplot2
library(ggplot2)

# Set a minimal theme for all future plots
theme_set(theme_minimal())

# Plot sentiment score for each episode
ggplot(by_episode, aes(episode_order, sentiment_score)) +
    geom_col() +
    geom_smooth()
```

Task 5: Instructions

Create an episode popularity plot.

- Plot the episode `rating` for each episode. Use `geom_point()` to display the ratings and `geom_smooth()` to see the trend. Add a *red*, *dashed*, *vertical* line for episode *100*.

The parameters **col**, **lty**, and **xintercept** might be useful. For `lty` and `col`, please use words and not the numerical equivalent.

-

Solution

```
# Plot episode ratings
ggplot(by_episode, aes(episode_order, rating)) +
    geom_point() +
    geom_smooth() +
    geom_vline(xintercept = 100, col = "red", lty = "dashed")
```

Task 6: Instructions

Plot a relationship between episode swear word ratio and popularity.

- Plot episode `swear_word_ratio` against episode `rating`. Use `geom_point()` with `alpha` transparency set to **0.6** and add `geom_smooth()` to add a smooth trend line. Use `percent` from the `scales` package to improve y-axis readability.

You can call a single function from a package that isn't attache by using the form, `package::function`.

-

Solution

```
# Plot swear word ratio over episode rating
ggplot(by_episode, aes(rating, swear_word_ratio)) +
    geom_point(alpha = 0.6, size = 3) +
    geom_smooth() +
    scale_y_continuous(labels = scales::percent) +
    scale_x_continuous(breaks = seq(6, 10, 0.5)) +
    labs(
        x = "IMDB rating",
        y = "Episode swear word ratio"
)
```

Task 7: Instructions

Create a function that compares profanity of two characters.

- Create `char_2` by filtering `words` for the second characater.
- Create `char_2_summary` by summarizing the number of swear words and a number of non-swear words.
- Convert `char_both_summary` to a `matrix` and run a `prop.test()` on it.
- Use `tidy` from `broom` to convert the statistical test result to a tidy data frame.

Once you're done, try playing a bit with the `compare_profanity()` function. Try this example: `compare_profanity("butters", "cartman", sp_words)`. Replace *butters* with any other character to compare it with *cartman*.

Helpful links:

- `broom` package vignette
- `dplyr` filter documentation
- more info about prop.test

-

Solution

```
# Create a function that compares profanity of two characters
compare_profanity <- function(char1, char2, words) {
    char_1 <- filter(words, character == char1)
    char_2 <- filter(words, character == char2)
    char_1_summary <- summarise(char_1, swear = sum(swear_word), total =
n() - sum(swear_word))
    char_2_summary <- summarise(char_2, swear = sum(swear_word), total =
n() - sum(swear_word))
    char_both_summary <- bind_rows(char_1_summary, char_2_summary)
    result <- prop.test(as.matrix(char_both_summary), correct = FALSE)
    return(broom::tidy(result) %>% bind_cols(character = char1))
```

```
}
```

Task 8: Instructions

Plot the comparison of profanity between Eric Cartman and others.

- Apply `compare_profanity()` to the `characters` vector to create a consistent data frame with all statistical results in one place.
- Plot `estimate1-estimate2` against reordered characters, descending by `estimate1`.
- Color all geoms based on `p.value` < 0.05.
- Add an errorbar geom to show estimate confidence intervals.

`ggplot2` geom parameters can have an expression that returns a logical vector for example.

Helpful links:

- `purrr::map()` documentation
- more info about p-values

-

Solution

```
# Vector of most speaking characters in the show
characters <- c("butters", "cartman", "kenny", "kyle", "randy", "stan",
"gerald", "mr. garrison",
                "mr. mackey", "wendy", "chef", "jimbo", "jimmy", "sharon",
"sheila", "stephen")

# Map compare_profanity to all characters against Cartman
prop_result <- purrr::map_df(characters, compare_profanity, "cartman",
sp_words)

# Plot estimate1-estimate2 confidence intervals of all characters and color
it by a p.value threshold
ggplot(prop_result, aes(x = reorder(character, -estimate1), estimate1-
estimate2, color = p.value < 0.05)) +
    geom_point() +
    geom_errorbar(aes(ymin = conf.low, ymax = conf.high), show.legend =
FALSE) +
    geom_hline(yintercept = 0, col = "red", linetype = "dashed") +
    theme(axis.text.x = element_text(angle = 60, hjust = 1))
```

Task 9: Instructions

Answer a few questions based on the completed analysis.

- Are naughty episodes more popular? `TRUE/FALSE`
- Is Eric Cartman the naughtiest character? `TRUE/FALSE`

- If he is, assign an empty string, otherwise, write its name.

-

Solution

```
# Are naughty episodes more popular? TRUE/FALSE
naughty_episodes_more_popular <- FALSE

# Is Eric Cartman the naughtiest character? TRUE/FALSE
eric_cartman_naughtiest <- FALSE

# If he is, assign an empty string, otherwise write his name
who_is_naughtiest <- "kenny"
```

Gender Bias in Graduate Admissions

Task 1: Instructions

Read in and tidy the `UCBAdmissions` dataset.

- Read in the `UCBAdmissions` dataset using the `data()` function.
- Load the `broom` package.
- Use the `tidy` function to convert the dataset to tidy format.

Good to know

This project uses the `tidyverse` suite of packages, particularly `dplyr` and `ggplot2`, so it would be useful to have some familiarity with those packages beforehand. If you need to brush up, work through DataCamp's Introduction to the Tidyverse course before you begin. Students should also have a knowledge of common data structures in R, as taught through DataCamp's Introduction to R course, as well as some understanding of logistic regression, as taught through Multiple and Logistic Regression.

The first exercise uses the wonderfully convenient tidy() function from the `broom` package, which converts a model object (a three-dimensional array in our case) into a tidy tibble, where each variable is a column and each observation is a row. For more information on the principles of tidy data, see Hadley Wickham's research paper on the topic.

-

Solution

```
# Load UCBAdmissions dataset
data("UCBAdmissions")

# Print dataset to console
print(UCBAdmissions)

# Load broom package
library(broom)

# Convert UCBAdmissions to tidy format
ucb_tidy <- tidy(UCBAdmissions)

# Print tidy dataset to console
print(ucb_tidy)
```

Task 2: Instructions

Calculate the overall acceptance rate for men and women.

- Use the `group_by()` function from `dplyr` to group the dataset by `Admit` and `Gender`.
- Calculate the *total* number of acceptances/rejections for men and women – regardless of department – using `sum(n)`.
- Create a new variable, `prop`, that equals the *proportion* of acceptances/rejections for each gender, using `n / sum(n)`.
- Filter the `Admit` variable to include `"Admitted"` students only.

For more information on `dplyr`, check out the RStudio cheatsheet.

-

Solution

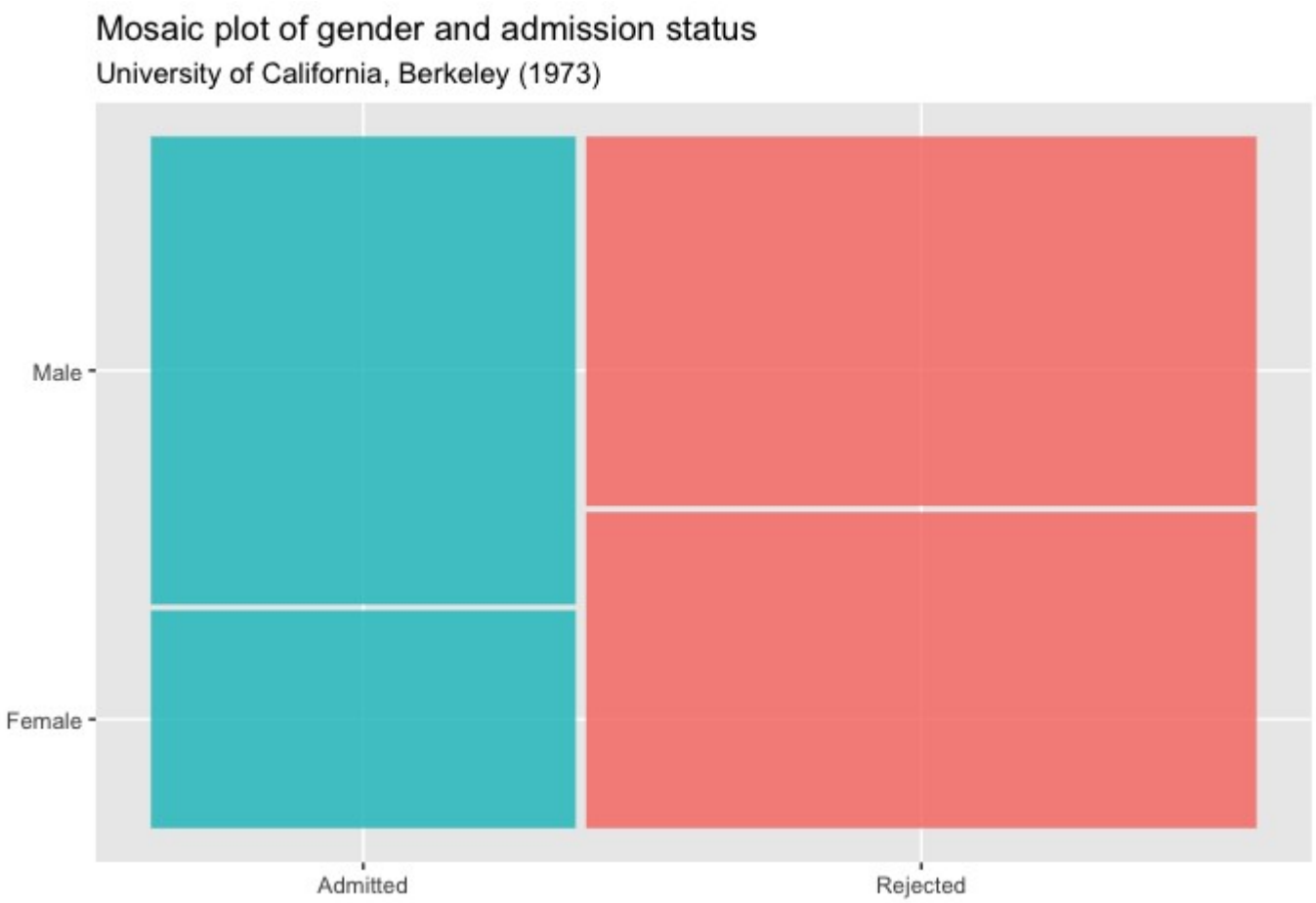
```
# Load the dplyr library
library(dplyr)

# Aggregate over department
ucb_tidy_aggregated <- ucb_tidy %>%
  group_by(Admit, Gender) %>%
  summarize(n = sum(n)) %>%
  ungroup() %>%
  group_by(Gender) %>%
  mutate(prop = n / sum(n)) %>%
  filter(Admit == "Admitted")

# Print aggregated dataset
print(ucb_tidy_aggregated)
```

Task 3: Instructions

Produce a bar chart using `ggplot2` that communicates the discrepancy in acceptance rate.

- Pipe the data into `ggplot()` with `Gender` on the x-axis, `prop` on the y-axis, and `Gender` as the fill variable.
- Use `geom_text()` to add a label for the percent of each gender accepted (`percent(prop)`), adjusting the height of the label by `-1`.
- Use `scale_y_continuous()` to format the y-axis labels as percentages and set its limits to `0` and `0.5`.

An alternative way to visualize categorical data is through a **mosaic plot**:



However, there is no native support for mosaic plots in `ggplot2` and bar charts are arguably clearer at communicating the discrepancy. Nevertheless, if you are interested in making your own mosaic plots in R, check out the built-in `mosaicplot()` function or the `ggmosaic` package.

For more information on `ggplot2`, check out the RStudio cheatsheet.

-

Solution

```
# Load the ggplot2 and scales packages
library(ggplot2)
library(scales)

# Prepare the bar plot
gg_bar <- ucb_tidy_aggregated %>%
    ggplot(aes(x = Gender, y = prop, fill = Gender)) +
    geom_col() +
    geom_text(aes(label = percent(prop)), vjust = -1) +
    labs(title = "Acceptance rate of male and female applicants",
         subtitle = "University of California, Berkeley (1973)",
         y = "Acceptance rate") +
    scale_y_continuous(labels = percent, limits = c(0, 0.5)) +
    guides(fill = FALSE)
```

```
# Print the bar plot
print(gg_bar)
```

Task 4: Instructions

Visualize acceptance rates separately for each department.

- After grouping by `Gender` and `Dept`, calculate the proportion of acceptances/rejections and filter for `Admitted` students.
- Use `facet_wrap()` to separate the graphic out by department.
- Remove the legend by setting the `fill` argument in `guides()` to `FALSE`.

Faceting is an efficient way to visualize data when it can be split by one or more categorical variables. There are two ways to facet data in `ggplot2`: `facet_wrap()` and `facet_grid()`. These two functions are similar and can produce identical output. As a rule of thumb, however, `facet_wrap()` is more convenient when you want to facet by a single variable, while `facet_grid()` is better for when you need to facet by two.

-

Solution

```
# Calculate acceptance/rejection rate
ucb_by_dept <- ucb_tidy %>%
    group_by(Gender, Dept) %>%
    mutate(prop = n / sum(n)) %>%
    filter(Admit == "Admitted")

# Print the dataset
print(ucb_by_dept)

# Prepare the bar plot for each department
gg_bar_faceted <- ucb_by_dept %>%
  ggplot(aes(Gender, prop, fill = Gender)) +
  geom_col() +
  geom_text(aes(label = percent(prop)), vjust = -1) +
  labs(title = "Acceptance rate of male and female applicants",
       subtitle = "University of California, Berkeley (1973)",
       y = "Acceptance rate") +
  scale_y_continuous(labels = scales::percent, limits = c(0, 1)) +
  facet_wrap(~Dept) +
  guides(fill = FALSE)

# Print the bar plot for each department
print(gg_bar_faceted)
```

Task 5: Instructions

De-aggregate the dataset so that each row represents one student.

- Define a function, `multiply_rows()`, that has two arguments: `column` and `n`
- Modify the body of the function so that it repeats each `column` `n` number of times.
- Create a new data frame, `ucb_full`, by applying the `multiply_rows()` function to the `Admit`, `Gender` and `Dept` columns.
- Use `nrow()` to check that the number of rows in `ucb_full` is equal to the number of students (4,526).

If you want to learn more about writing your own functions, we recommend DataCamp course, [Introduction to Function Writing in R](). It will help you to make your code more readable and automate repetitive tasks.

-

Solution

```
# Define function that repeats each row in each column n times
multiply_rows <- function(column, n) {
  rep(column, n)
}

# Create new de-aggregated data frame using the multiply_rows function
ucb_full <- data.frame(Admit = multiply_rows(ucb_tidy$Admit, ucb_tidy$n),
                       Gender = multiply_rows(ucb_tidy$Gender, ucb_tidy$n),
                       Dept = multiply_rows(ucb_tidy$Dept, ucb_tidy$n))

# Check the number of rows equals the number of students
nrow(ucb_full) == 4526
```

Task 6: Instructions

Run a binary logistic model that predicts admission as a function of gender alone.

- Use `fct_relevel()` to change the coding of the `Admit` variable, so that `Rejected` is level 1 and `Admitted` is level 2.
- Run the model with `Gender` as the only explanatory variable and `family` set to `binomial`.
- Use `summmary()` to summarize the results of the model.

If you need a refresher on logistic regression, check out Chapter 4 of Ben Baumer's DataCamp course, [Multiple and Logistic Regression]().

-

Solution

```
# Load the forcats library
library(forcats)
```

```
# Reverse the coding of the Admit variable
ucb_full$Admit <- fct_relevel(ucb_full$Admit, "Rejected", "Admitted")

# Run the regression
glm_gender <- glm(Admit ~ Gender, data = ucb_full, family = "binomial")

# Summarize the results
summary(glm_gender)
```

Task 7: Instructions

Run a binary logistic model that predicts admission as a function of both gender and department.

- Run the model again, but with `Dept` as an added explanatory variable. Remember to set `family` equal to `binomial`.
- Summarize the results.
-

Solution

```
# Run the regression, including Dept as an explanatory variable
glm_genderdept <- glm(Admit ~ Gender + Dept, data = ucb_full, family =
"binomial")

# Summarize the results
summary(glm_genderdept)
```

Task 8: Instructions

Run a binary logistic model that predicts admission as a function of gender, for Department A only.

- Filter `ucb_full` to include Department A only and save the result to `dept_a`.
- Run the model again, but with `Gender` as the only explanatory variable and `dept_a` as the dataset.
- Summarize the results.
-

Solution

```
# Filter for Department A
dept_a <- ucb_full %>%
    filter(Dept == "A")

# Run the regression
glm_gender_depta <- glm(Admit ~ Gender, data = dept_a, family = "binomial")

# Summarize the results
```

```
summary(glm_gender_depta)
```

Task 9: Instructions

Think about the relationship between bias and disrimination.

- Define `bias` based on the Bickel, Hammel & O'Connell (1975) paper (p. 398, middle column).
- Define `discrimination` based on the same paper.
- Test if `bias` is equal to `discrimination`.

If you're interested in further exploring the logic behind Simpson's paradox, we recommend the following (openly accessible) research papers:

- The Interpretation of Interaction in Contingency Tables (Simpson, 1951): The 'discovery' of the concept by British statistician Edward H. Simpson
- Simpson's paradox in psychological science (Kievit et al., 2013): The incidence of Simpson's paradox across cognitive neuroscience, clinical psychology and other areas of psychological science
- Understanding Simpson's Paradox (Pearl, 2013): A contemporary discussion of Simpson's paradox with an emphasis on causal inference

-

Solution

```
# Define bias
bias <- "a pattern of association between a particular decision and a
particular sex of applicant, of sufficient strength to make us confident
that it is unlikely to be the result of chance alone"

# Define discrimination
discrimination <- "the exercise of decision influenced by the sex of the
applicant when that is immaterial to the qualifications for entry"

# Does bias equal discrimination?
bias == discrimination
```

Degrees That Pay You Back

Task 1: Instructions

Load the required libraries and the `degrees-that-pay-back.csv` dataset.

- Load the `tidyr`,`dplyr`,`readr`, `ggplot2`, `cluster`, and `factoextra` libraries.
- Use `read_csv` (not `read.csv`) to read in `datasets/degrees-that-pay-back.csv` and use the `col_names` parameter to change the column names to the following: `College.Major`, `Starting.Median.Salary`, `Mid.Career.Median.Salary`, `Career.Percent.Growth`, `Percentile.10`, `Percentile.25`, `Percentile.75`, and `Percentile.90`. Assign the dataframe to `degrees`.
- Display the first few rows of `degrees`.
- Explore a `summary` of `degrees`.

Make sure to use `read_csv` (with an underscore) to read in the data. The `read.csv` function, which is built into R, has a number of problems which the new `read_csv` function avoids. The sample code already includes this, but the `skip` parameter is useful if you are using `col_names` to *replace* the column names. If `skip` is not set to 1, the original column names will default to row 1 of your imported data. For more information, check out the documentation here.

Good to know

This project assumes familiarity with standard tidyverse tools for R like the `dplyr`, `ggplot2` and the pipe operator (`%>%`). While not required, it would also be helpful to have a prior understanding of clustering unsupervised data with k-means. Before taking on this project we recommend that you have completed the following courses:

- Introduction to the Tidyverse
- Cluster Analysis in R

RStudio has created some very helpful cheat sheets for working in the tidyverse, including two that will be helpful for this project:

- Data Wrangling cheat sheet
- ggplot2 cheat sheet

-

Solution

```
# Load relevant packages
library(tidyr)
library(dplyr)
```

```
library(readr)
library(ggplot2)
library(cluster)
library(factoextra)

# Read in the dataset
degrees <- read_csv('datasets/degrees-that-pay-back.csv',
col_names=c('College.Major',

'Starting.Median.Salary','Mid.Career.Median.Salary','Career.Percent.Growth'
,

'Percentile.10','Percentile.25','Percentile.75','Percentile.90'), skip=1)

# Display the first few rows and a summary of the data frame
head(degrees)
summary(degrees)
```

Task 2: Instructions

Clean up the data types.

- Use `mutate_at` to modify all columns *except* `College.Major`, using the `gsub` function to strip the dollar signs. The code is already there, but you'll be converting the result to numeric with `as.numeric` at the same time.
- Use `mutate` to divide the `Career.Percent.Growth` values by 100.

You can use the helper function `vars` to apply `mutate_at` on a selection of columns. For more information, check out the documentation here.

The `gsub()` function looks for instances of the "regular expression" in the first parameter, and replaces it with the second parameter. For more tips on using `gsub`, check out this article on removing currency formating in R. You don't need to know everything about regular expressions to complete this task, but if you're curious for more practice or reference, check out this website.

For more tips on how to apply `gsub` across multiple columns and convert the results to numeric, check out this stack overflow answer.

Helpful links:

- dplyr's `mutate()` function documentation
- Mutate multiple functions with dplyr's `mutate_at()` function documentation
- `gsub` documentation

-

Solution

```
# Clean up the data
degrees_clean <- degrees %>%
```

```
    mutate_at(vars(Starting.Median.Salary:Percentile.90),
              function(x) as.numeric(gsub("[\\$,]","",x))) %>%
    mutate(Career.Percent.Growth = Career.Percent.Growth/100)
```

Task 3: Instructions

Select the relevant data for the k-means algorithm and apply the Elbow Method.

- Select `Starting.Median.Salary`, `Mid.Career.Median.Salary`, `Percentile.10`, and `Percentile.90` columns from `degrees_clean` and assign to `k_means_data`.
- Scale `k_means_data` using the `scale()` function.
- Use `fviz_nbclust` on the `k_means_data` with `FUNcluster` set to `kmeans` and `method` set to `"wss"`. Assign the results to `elbow_method`.

It's a good idea to normalize your data in preparation of applying the k-means algorithm. Check out the `scale` documentation here. For more information on choosing the optimal number of clusters, check out this article: Determining The Optimal Number of Clusters: 3 Must Know Methods. For more on how `fviz_nbclust` works, check out the documentation.

-

Solution

```
# Select and scale the relevant features and store as k_means_data
k_means_data <- degrees_clean %>%
    select(Starting.Median.Salary, Mid.Career.Median.Salary,
           Percentile.10, Percentile.90) %>%
    scale()

# Run the fviz_nbclust function with our selected data and method "wss"
elbow_method <- fviz_nbclust(k_means_data, kmeans, method = "wss")

# View the plot
elbow_method
```

Task 4: Instructions

Apply the Silhouette Method.

- Run `fviz_nbclust` again but with `"silhouette"` as the `method` this time.

Helpful links:

- `fviz_nbclust` documentation
- Determining The Optimal Number Of Clusters: 3 Must Know Methods

-

Solution

```
# Run the fviz_nbclust function with the method "silhouette"
silhouette_method <- fviz_nbclust(k_means_data, kmeans,
                                  method = "silhouette")

# View the plot
silhouette_method
```

Task 5: Instructions

Apply the Gap Statistic Method.

- Apply the `clusGap()` function on the `k_means_data` with `FUN` set to `kmeans`, `nstart` set to `25`, `K.max` set to `10`, and `B` set to `50`. Store the result as `gap_stat`.
- Visualize the results by applying the `fviz_gap_stat()` function on `gap_stat`.

Helpful links:

- `clusGap` documentation
- `fviz_nbclust` documentation
- Determining The Optimal Number Of Clusters: 3 Must Know Methods

-

Solution

```
# Use the clusGap function to apply the Gap Statistic Method
gap_stat <- clusGap(k_means_data, FUN = kmeans, nstart = 25,
                    K.max = 10, B = 50)

# Use the fviz_gap_stat function to vizualize the results
gap_stat_method <- fviz_gap_stat(gap_stat)

# View the plot
gap_stat_method
```

Task 6: Instructions

Run the k-means algorithm and label the clusters of the `degrees` data.

- Set the seed to 111 using `set.seed()` to make sure you get the same results every time.
- Set `num_clusters` equal to the recommended number of clusters, `3`.

- Run the `kmeans()` function on the `k_means_data` with `centers` set to `num_clusters`, `iter.max` set to `15`, and `nstart` set to `25`. Store the result as `k_means`.
- Create a new data frame `degrees_labeled` by adding a new column `clusters` to `degrees_clean` equal to the `cluster` values in the object stored as `k_means`.

Check out this article if you're curious about reasons for using `set.seed` here. For more information on `kmeans` function and resulting output, check out the documentation.

-

Solution

```
# Set a random seed
set.seed(111)

# Set k equal to the optimal number of clusters
num_clusters <- 3

# Run the k-means algorithm
k_means <- kmeans(k_means_data, num_clusters, iter.max = 15, nstart = 25)

# Add back the cluster labels to degrees
degrees_labeled <- degrees_clean %>%
    mutate(clusters = k_means$cluster)
```

Task 7: Instructions

Plot each major by Starting Median Salary vs. Mid Career Median Salary.

- Use `ggplot` to scatter plot `degrees_labeled` with `Mid.Career.Median.Salary` as a function of`Starting.Median.Salary` and `color` set to the `clusters` column as a factor (don't forget `geom_point()`!).
- Make your plot more readable by adding labels using `xlab`, `ylab`, and `ggtitle`.
- Use both `scale_x_continuous` and `scale_y_continuous` with `labels` set to `scales::dollar` to format the axes as currency values.
- Customize the transparency, shape, and colors of the scatter points, if you'd like.

Need some inspiration in customizing your visualization? Within `geom_point` set the `alpha` to 4/5 and `size` to 7 to change the transparency and size of the points. Use `scale_color_manual()` to set the `name` of the color legend to "Clusters" and assign the following color codes to `values`: #EC2C73, #29AEC7, #FFDD30. Want to know more about adding color to your graphs? Check out this article.

`ggplot` by default abbreviates the axis tick text of large continuous values. For more information on customizing the axes with `scale_*_continuous`, check out this documentation.

-

Solution

```
# Graph the clusters by Starting and Mid Career Median Salaries
career_growth <- ggplot(degrees_labeled,
aes(x=Starting.Median.Salary,y=Mid.Career.Median.Salary,
    color=factor(clusters))) +
    geom_point(alpha=4/5,size=6) +
    scale_x_continuous(labels = scales::dollar) +
    scale_y_continuous(labels = scales::dollar) +
    xlab('Starting Median Salary') +
    ylab('Mid Career Median Salary') +
    scale_color_manual(name="Clusters",values=c("#EC2C73","#29AEC7",
                    "#FFDD30")) +
    ggtitle('Clusters by Starting vs. Mid Career Median Salaries')

# View the plot
career_growth
```

Task 8: Instructions

Reshape the data to prepare for the upcoming visualizations.

- Select the `College.Major`, `Percentile.10`, `Percentile.25`, `Mid.Career.Median.Salary`, `Percentile.75`, `Percentile.90`, and `clusters` columns.
- Apply the `gather()` function on all columns *except* `College.Major` and `clusters`, with percentile as the `key` and salary as the `value`, which will collapse the percentile columns into rows of one new column, `percentile`, with the respective salary values in the new column `salary`.
- Use `mutate` to reorder the factor levels of the new `percentile` column from lowest to highest (such that `Mid.Career.Median.Salary` falls after `Percentile.25` and before `Percentile.75`).

The `gather()` function allows us "gather" columns into rows with key-value pairs, reshaping data from a wide to long format. This is a helpful trick for visualizations, when you'd like to compare several columns with related values on the same x-axis, rather than across multiple graphs. For the purposes of this task, you want to gather the *mid career percentile* columns, grouped by `College.Major` and `clusters`. Make sure to exclude those two columns in the selection for the `gather` function using `-c(....)`. For more information on `gather` check out the documentation.

-

Solution

```
# Use the gather function to reshape degrees and
# use mutate() to reorder the new percentile column
degrees_perc <- degrees_labeled %>%
    select(College.Major, Percentile.10, Percentile.25,
           Mid.Career.Median.Salary, Percentile.75,
           Percentile.90, clusters) %>%
    gather(key=percentile, value=salary, -c(College.Major, clusters)) %>%

mutate(percentile=factor(percentile,levels=c('Percentile.10','Percentile.25
',
            'Mid.Career.Median.Salary','Percentile.75','Percentile.90')))
```

Task 9: Instructions

Plot the majors of Cluster 1 by percentile.

- Use `ggplot()` to plot `degrees_perc` (filtered such that `clusters` equals 1) with `salary` as a function of `percentile` as a scatter plot (`geom_point()`), and set `group` and `color` to `College.Major`.
- Add lines to connect the points using `geom_line()`.
- Set the title to "Cluster 1: The Liberal Arts".
- Use the `theme()` and `element_text()` functions to set `axis.text.x` with a `size` of 7 and `angle` of 25 so the tick labels will be more readable.

For more on how you can use `theme` to customize the axis labels, check out the documentation.

-

Solution

```
# Graph the majors of Cluster 1 by percentile
cluster_1 <-  ggplot(degrees_perc[degrees_perc$clusters==1,],
                     aes(x=percentile,y=salary,
                     group=College.Major, color=College.Major,
order=salary)) +
                     geom_point() +
                     geom_line() +
                     ggtitle('Cluster 1:  The Liberal Arts') +
                     theme(axis.text.x = element_text(size=7, angle=25))

# View the plot
cluster_1
```

Task 10: Instructions

Plot the majors of Cluster 2 by percentile.

- Copy the code from Task 9, but where `clusters` is equal to 2.
- Change the title to "Cluster 2: The Goldilocks".

-

Solution

```
# Modify the previous plot to display Cluster 2
cluster_2 <- ggplot(degrees_perc[degrees_perc$clusters==2,],
    aes(x=percentile,y=salary,
    group=College.Major, color=College.Major)) +
    geom_point() +
    geom_line() +
    ggtitle('Cluster 2: The Goldilocks') +
    theme(axis.text.x = element_text(size=7, angle=25))

# View the plot
cluster_2
```

Task 11: Instructions

Plot the majors of Cluster 3 by percentile.

- Copy the code from Task 10, but where `clusters` is equal to 3.
- Change the title to "Cluster 3: The Over Achievers".

-

Solution

```
# Modify the previous plot to display Cluster 3
cluster_3 <- ggplot(degrees_perc[degrees_perc$clusters==3,],
    aes(x=percentile,y=salary,
    group=College.Major, color=College.Major)) +
    geom_point() +
    geom_line() +
    ggtitle('Cluster 3:  The Over Achievers') +
    theme(axis.text.x = element_text(size=7, angle=25))

# View the plot
cluster_3
```

Task 12: Instructions

Identify the two majors with the highest career percent growth.

- Use `arrange()` to sort the `degrees_labeled` data frame by `Career.Percent.Growth` in descending order (`desc`).

- Assign the top two majors (they're tied!) to the list `highest_career_growth`.

Great job! Now that you've finished, feel free to go back and tinker with some of the recommended inputs to the `kmeans()` function. What happens when you select different columns for the data in the algorithm? Or change the number of clusters?

-

Solution

```
# Sort degrees by Career.Percent.Growth
arrange(degrees_labeled,desc(Career.Percent.Growth))

# Identify the two majors tied for highest career growth potential
highest_career_growth <- c('Philosophy','Math')
```

Book Recommendations from Charles Darwin

Task 1: Instructions

List Darwin's bibliography.

- Retrieve a list of all text files (`*.txt`) present in the `datasets/` folder using `glob.glob()`. Store them in `files`.
- Sort `files` alphabetically using the `sort()` method.

Good to know

In order to be well prepared for this project, it is recommended you have finished the following DataCamp courses:

- Data Manipulation with pandas
- Introduction to Natural Language Processing in Python

The following links will be helpful to complete task 1:

- How to list files from a folder using glob
- Python's sort() method to sort a list

The following links may be helpful throughout the project:

- pandas cheat sheet
- gensim tutorials
- This DataCamp article on stemming
- Charles Darwin's bibliography

-

Solution

```
# Import library
import glob

# The books files are contained in this folder
folder = "datasets/"

# List all the .txt files and sort them alphabetically
files = glob.glob(folder + "*.txt")
files.sort()
files
```

Task 2: Instructions

Load Darwin's bibliography into Python.

- Open the file for each book with its `encoding` set to `utf-8-sig`.
- Remove all non-alphanumeric characters using `re.sub()`.
- Store the texts and titles of the books in two separate lists called `txts` and `titles`. Use the `os.path.basename()` and `replace()` functions to remove the folder name and .txt extension from the file name.

Helpful links:

- An example of how to remove non-alphanumeric character using the re library
- Python's os.path.basename() function to remove the folder name from a file name
- Python's replace() method to replace a substring from a string
- Python's append() function to add elements to a list

-

Solution

```
# Import libraries
import re, os

# Initialize the object that will contain the texts and titles
txts = []
titles = []

for n in files:
    # Open each file
    f = open(n, encoding='utf-8-sig')
    # Remove all non-alpha-numeric characters
    data = re.sub('[\W_]+', ' ', f.read())
    # Store the texts and titles of the books in two separate lists
    txts.append(data)
    titles.append(os.path.basename(n).replace(".txt", ""))

# Print the length, in characters, of each book
[len(t) for t in txts]
```

Task 3: Instructions

Find the index of the "On the Origin of Species" book.

- Browse the `titles` list and store the index of the title named 'OriginofSpecies' in a variable call `ori`.
- Print the content of the `ori` variable.

-

Solution

```
# Browse the list containing all the titles
for i in range(len(titles)):
    # Store the index if the title is "OriginofSpecies"
    if(titles[i]=="OriginofSpecies"):
        ori = i

# Print the stored index
print(str(ori))
```

Task 4: Instructions

Tokenize the corpus.

- Convert the contents of each book in `txts` to lower case using the `lower()` method.
- Transform the lower case text into tokens (individual words) using the `split()` method and store it in a variable called `txts_split`.
- Remove tokens which are part of the list of stop words in `stoplist` and store the resulting list in a variable called `texts`.
- Print the first 20 tokens for the "On the Origin of Species" book.

Using a list comprehension is recommended for the first three bullets of this task.

Helpful links:

- Python's split() function
- Python's lower() function

-

Solution

```
# Define a list of stop words
stoplist = set('for a of the and to in to be which some is at that we i who
whom show via may my our might as well'.split())

# Convert the text to lower case
txts_lower_case = [txt.lower() for txt in txts]

# Transform the text into tokens
txts_split = [txt.split() for txt in txts_lower_case]

# Remove tokens which are part of the list of stop words
texts = [[word for word in txt if word not in stoplist] for txt in
txts_split]
```

```
# Print the first 20 tokens for the "On the Origin of Species" book
texts[ori][0:20]
```

Task 5: Instructions

Get corresponding stems for each token.

- Open the pregenerated pickle file (`datasets/texts_stem.p`) with the mode set to `'rb'` then load the stemmed tokens list into `texts_stem` using the `pickle.load()` function.
- Print the 20 first stemmed tokens from the "On the Origin of Species" book.

Helpful links:

- Using the pickle library

For reference, the object contained in the pickle file was generated the following way:

```
# Load the Porter stemming function from the nltk package
from nltk.stem import PorterStemmer

# Create an instance of a PorterStemmer object
porter = PorterStemmer()

# For each token of each text, we generated its stem
texts_stem = [[porter.stem(token) for token in text] for text in texts]

# Save to pickle file
pickle.dump( texts_stem, open( "datasets/texts_stem.p", "wb" ) )
```

-

Solution

```
import pickle

# Load the stemmed tokens list from the pregenerated pickle file
texts_stem = pickle.load( open( "datasets/texts_stem.p", "rb" ) )

# Print the 20 first stemmed tokens from the "On the Origin of Species"
book
texts_stem[ori][0:20]
```

Task 6: Instructions

Build a bag-of-words model from the stemmed tokens.

- Create a dictionary from the stemmed tokens using the `corpora.Dictionary()` function.
- Create a bag-of-words model for each book, using the previously generated dictionary and the `dictionary.doc2bow()` function.
- Print the first five elements of the On the Origin of species' BoW model to understand its structure.

Using a list comprehension is recommended for the second bullet in this task.

Helpful links:

- This tutorial on creating dictionaries and bag-of-words models
-

Solution

```
# Load the functions allowing to create and use dictionaries
from gensim import corpora

# Create a dictionary from the stemmed tokens
dictionary = corpora.Dictionary(texts_stem)

# Create a bag-of-words model for each book, using the previously generated
dictionary
bows = [dictionary.doc2bow(text) for text in texts_stem]


# Print the first five elements of the On the Origin of species' BoW model
bows[ori][0:5]
```

Task 7: Instructions

Visualize the most common words of "On the Origin of Species".

- Convert the BoW model for "On the Origin of Species" into a DataFrame using pandas' `DataFrame()` function.
- Add the column names to the DataFrame (names should be `index` and `occurrences`).
- Add a column named `token` containing the stemmed token corresponding to the dictionary index.
- Sort the DataFrame by descending number of occurrences and print the first 10 values.

Helpful links:

- Create a pandas DataFrame from a list or dictionary
- pandas cheat sheet

-

Solution

```
# Import pandas to create and manipulate DataFrames
import pandas as pd

# Convert the BoW model for "On the Origin of Species" into a DataFrame
df_bow_origin = pd.DataFrame(bows[ori])

# Add the column names to the DataFrame
df_bow_origin.columns = ["index", "occurrences"]

# Add a column containing the token corresponding to the dictionary index
df_bow_origin["token"] = [dictionary[index] for index in
df_bow_origin["index"]]

# Sort the DataFrame by descending number of occurrences and print the
first 10 values
df_bow_origin.sort_values(by="occurrences", ascending=False).head(10)
```

Task 8: Instructions

Build a tf-idf model.

- Generate the tf-idf model from the bag-of-words model using gensim's `TfidfModel()` function.
- Print the model for "On the Origin of Species".

Helpful links:

- Documentation on tf-idf models in gensim

-

Solution

```
# Load the gensim functions that will allow us to generate tf-idf models
from gensim.models import TfidfModel

# Generate the tf-idf model
model = TfidfModel(bows)

# Print the model for "On the Origin of Species"
model[bows[ori]]
```

Task 9: Instructions

Visualize the tf-idf model for "On the Origin of Species".

- Convert the list of token indices and scores into a DataFrame for the "On the Origin of Species" book using pandas' `DataFrame()` function.
- Name the columns of the DataFrame `id` and `score`.
- Add the tokens corresponding to the numerical indices in a column named `token` for better readability.
- Sort the DataFrame by descending tf-idf score and print the first 10 rows.

Helpful links:

- Create a pandas DataFrame from a list or dictionary
- pandas cheat sheet

-

Solution

```
# Convert the tf-idf model for "On the Origin of Species" into a DataFrame
df_tfidf = pd.DataFrame(model[bows[ori]])

# Name the columns of the DataFrame id and score
df_tfidf.columns=["id", "score"]

# Add the tokens corresponding to the numerical indices for better
readability
df_tfidf['token'] = [dictionary[i] for i in list(df_tfidf["id"])]

# Sort the DataFrame by descending tf-idf score and print the first 10
rows.
df_tfidf.sort_values(by="score", ascending=False).head(10)
```

Task 10: Instructions

Compute the pairwise distance between each text of the corpus.

- Compute the similarity matrix (pairwise distance between all tf-idf models) using gensim's `similarities.MatrixSimilarity()` function.
- Convert the resulting object as a list and then transform it into a DataFrame.
- Add the titles of the books as columns and index of the DataFrame.
- Print the resulting matrix.

Helpful links:

- Similarities in gensim (see Example 4).
- Create a pandas dataframe from a list or dictionary
- pandas cheat sheet
- Change a DataFrame's column names in pandas
- Change a DataFrame's index names in pandas

-

Solution

```
# Load the library allowing similarity computations
from gensim import similarities

# Compute the similarity matrix (pairwise distance between all texts)
sims = similarities.MatrixSimilarity(model[bows])

# Transform the resulting list into a DataFrame
sim_df = pd.DataFrame(list(sims))

# Add the titles of the books as columns and index of the DataFrame
sim_df.columns = titles
sim_df.index = titles

# Print the resulting matrix
sim_df
```

Task 11: Instructions

Display the book most similar to "*On the Origin of Species*".

- Select the column corresponding to "On the Origin of Species" (the title is `OriginofSpecies`).
- Sort the resulting series by ascending scores.
- Plot this series as a horizontal bar plot using matplotlib's `barh()` method.
- Modify the axes labels and plot title for a better readability (e.g., using functions such as `xlabel()`, `ylabel()` or `title()`).

Helpful links:

- [Horizontal bar plot in pandas/matplotlib]

-

Solution

```
# This is needed to display plots in a notebook
%matplotlib inline

# Import the needed functions from matplotlib
import matplotlib.pyplot as plt

# Select the column corresponding to "On the Origin of Species" and
v = sim_df["OriginofSpecies"]

# Sort by ascending scores
v_sorted = v.sort_values(ascending=True)

# Plot this data has a horizontal bar plot
```

```
v_sorted.plot.barh(x='lab', y='val', rot=0).plot()

# Modify the axes labels and plot title for better readability
plt.xlabel("Cosine distance")
plt.ylabel("")
plt.title("Most similar books to 'On the Origin of Species'")
```

Task 12: Instructions

Show which books are more similar to each other.

- Compute the clusters from the similarity matrix using the `'ward'` variance minimization algorithm from SciPy's `hierarchy.linkage()` function.
- Display this result as a horizontal dendrogram using the `hierarchy.dendrogram()` function. The following arguments are recommended: `leaf_font_size=8`, `labels=sim_df.index`, `orientation="left"`.

In order to prevent the `hierarchy.dendrogram()` function to print a long text output before the plot, assign the function call to a variable (e.g., `a = hierarchy.dendrogram(...)`) so that only the plot is displayed.

Helpful links:

- SciPy's hierarchy.linkage() function
- Creating a dendrogram in Python

-

Solution

```
# Import libraries
from scipy.cluster import hierarchy

# Compute the clusters from the similarity matrix,
# using the Ward variance minimization algorithm
Z = hierarchy.linkage(sim_df, 'ward')

# Display this result as a horizontal dendrogram
a = hierarchy.dendrogram(Z,  leaf_font_size=8, labels=sim_df.index,
orientation="left")
```

Extract Stock Sentiment from News Headlines

Task 1: Instructions

Load the HTML file for each stock into memory.

- Create an empty dictionary and assign it to `html_tables`.
- In the for loop, load each stock's HTML file (which is in a directory called `datasets/`) into a BeautifulSoup object.
- Find `'news-table'` in the Soup and load it into `html_table`.
- Add an item to the dictionary with `table_name` as the key and `html_table` as the value.

Good to know

This project lets you apply the skills from Intermediate Python for Data Science, Data Manipulation with pandas, and Introduction to Natural Language Processing in Python. We recommend that you take those courses before starting this project.

Helpful links:

- BeautifulSoup documentation
- BeautifulSoup DataCamp tutorial

-

Solution

```
# Import libraries
from bs4 import BeautifulSoup
import os

html_tables = {}

# For every table in the datasets folder...
for table_name in os.listdir('datasets'):
    #this is the path to the file. Don't touch!
    table_path = f'datasets/{table_name}'
    # Open as a python file in read-only mode
    table_file = open(table_path, 'r')
    # Read the contents of the file into 'html'
    html = BeautifulSoup(table_file)
    # Find 'news-table' in the Soup and load it into 'html_table'
    html_table = html.find(id='news-table')
    # Add the table to our dictionary
    html_tables[table_name] = html_table
```

Task 2: Instructions

Inspect Tesla's HTML file with BeautifulSoup.

- Find all of the `<tr>` tags in Tesla's BeautifulSoup object, which is loaded into `tsla`.
- For each row in Tesla's headlines table:
- Read the text contents of the `<a>` tag.
- Read the contents of the `<td>` tag.
- Print the loop counter.

Helpful links:

- The `<tr>` tag
- The `<td>` tag
- The `<a>` tag
- w3schools, a website with lots of great info on HTML file structure

-

Solution

```
# Read one single day of headlines
tsla = html_tables['tsla_22sep.html']
# Get all the table rows <tr> in the file into 'tesla_tr'
tsla_tr = tsla.findAll('tr')

# For each row...
for i, table_row in enumerate(tsla_tr):
    # Read the text of the element 'a' into 'link_text'
    link_text = table_row.a.get_text()
    # Read the text of the element <td> into 'data_text'
    data_text = table_row.td.get_text()
    # Print the count
    print(f'File number {i+1}:')
    # Print the contents of 'link_text' and 'data_text'
    print(link_text)
    print(data_text)
    # The following exits the loop after four rows to prevent spamming the
notebook, do not touch
    if i == 3:
        break
```

Task 3: Instructions

Extract key information from each stock's BeautifulSoup object.

- Write and if/else control structure to deal with the scraped text into `time` or `date` and `time`.
- Extract the ticker name from the name of the file.

- Save all these fields to the `parsed_news` list, making a list of lists.

The date is only available *once* before the headline, on the first headline of the day. Because we are parsing the headlines chronologically, we only need to read the variable `date` once per day as the `time` changes with every headline (the `date` does not).

This is arguably the hardest part of the project from the coding point of view. In the real world, this is the kind of bottleneck where a lot of Data Scientist struggle, so don't give up and keep polishing your scraping skills!

-

Solution

```
# Hold the parsed news into a list
parsed_news = []
# Iterate through the news
for file_name, news_table in html_tables.items():
    # Iterate through all tr tags in 'news_table'
    for x in news_table.findAll('tr'):
        # Read the text from the tr tag into text
        text = x.get_text()
        # Split the text in the td tag into a list
        date_scrape = x.td.text.split()
        # If the length of 'date_scrape' is 1, load 'time' with the only
element
        # If not, load 'date' with the 1st element and 'time' with the
second
        if len(date_scrape) == 1:
            time = date_scrape[0]
        else:
            date = date_scrape[0]
            time = date_scrape[1]

        # Extract the ticker from the file name, get the string up to the
1st '_'
        ticker = file_name.split("_")[0]
        # Append ticker, date, time and headline as a list to the
'parsed_news' list
        parsed_news.append([ticker, date, time, x.a.text])
```

Task 4: Instructions

Tune NLTK/VADER to fit the context of financial news headlines.

- Make an instance of `SentimentIntensityAnalyzer` using the existing lexicon.
- Update the lexicon with new words.

Before being able to load in the VADER lexicon, you have to download it using the command:

```
nltk.download('stopwords')
```

But in this project, this step has already been done for you.

Helpful links:

- VADER source code
-

Solution

```
# NLTK VADER for sentiment analysis
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# New words and values
new_words = {
    'crushes': 10,
    'beats': 5,
    'misses': -5,
    'trouble': -10,
    'falls': -100,
}

# Instantiate the sentiment intensity analyzer with the existing lexicon
vader = SentimentIntensityAnalyzer()
# Update the lexicon
vader.lexicon.update(new_words)
```

Task 5: Instructions

Score the headlines and save them in a DataFrame.

- Convert the list of lists from task 3 into a pandas DataFrame.
- Get the polarity scores for all headlines iterating through it.
- Convert the result above into a DataFrame.
- Join both DataFrames in order to add the score to it.

Calculating the scores is relatively easy to the slightly awkward `pandas` needed here, as `vader` was not designed with `pandas` in mind. There are several ways of doing this, for example, `df.apply`, but there is no way of avoiding having to convert back the results to `DataFrame` and merge back with a `df.join` or similar.

VADER is more high level than most of NLTK and was not designed specifically for the purpose of analyzing journalists headlines. If you are considering doing something like this with real money, I would suggest building something like VADER yourself starting from core NLTK.

Helpful links:

- The difference between using Vader and more low level NLTK
- pandas joins guide

-

Solution

```
import pandas as pd
# Use these column names
columns = ['ticker', 'date', 'time', 'headline']
# Convert the list of lists into a DataFrame
scored_news = pd.DataFrame(parsed_news, columns=columns)

# Iterate through the headlines and get the polarity scores
scores = [vader.polarity_scores(headline) for headline in
scored_news.headline]
# Convert the list of dicts into a DataFrame
scores_df = pd.DataFrame(scores)
scored_news.columns = columns
# Join the DataFrames
scored_news = scored_news.join(scores_df)
# Convert the date column from string to a date
scored_news['date'] = pd.to_datetime(scored_news.date).dt.date
```

Task 6: Instructions

Plot a bar chart with all the mean polarity scores.

- Group the headlines by date and ticker name.
- Unstack the column ticker.
- Plot the bar chart with the resulting DataFrame.

Helpful links:

- `df.plot.bar()` function documentation
- pandas MultiIndexes documentation

-

Solution

```
import matplotlib.pyplot as plt
plt.style.use("fivethirtyeight")
%matplotlib inline

# Group by date and ticker columns from scored_news and calculate the mean
mean_c = scored_news.groupby(['date', 'ticker']).mean()
# Unstack the column ticker
mean_c = mean_c.unstack('ticker')
# Get the cross-section of compound in the 'columns' axis
```

```
mean_c = mean_c.xs("compound", axis="columns")
# Plot a bar chart with pandas
mean_c.plot.bar(figsize = (10, 6));
```

Task 7: Instructions

Take care of weekends and duplicates by discarding them.

- Count the number of headlines.
- Drop duplicates using the `drop_duplicates` method, setting the `subset` based on `ticker` and `headline`.
- Count the number of headlines after dropping duplicates.
- Print the result of the before and after in order to check that what you did had an effect.

-

Solution

```
# Count the number of headlines in scored_news (store as integer)
num_news_before = scored_news.headline.count()
# Drop duplicates based on ticker and headline
scored_news_clean = scored_news.drop_duplicates(subset=['headline',
'ticker'])
# Count number of headlines after dropping duplicates (store as integer)
num_news_after = scored_news_clean.headline.count()
# Print before and after numbers to get an idea of how we did
f"Before we had {num_news_before} headlines, now we have {num_news_after}"
```

Task 8: Instructions

Extract the 3rd of January for the Facebook stock.

- Select the cross-section of the Facebook row.
- Select the 3rd of January.
- Convert the datetime string column to be a time column exclusively.
- Set the index to the time column and sort by it.

-

Solution

```
# Set the index to ticker and date
single_day = scored_news_clean.set_index(['ticker', 'date'])
# Cross-section the fb row
single_day = single_day.xs('fb')
# Select the 3rd of January of 2019
single_day = single_day.loc['2019-01-03']
```

```
# Convert the datetime string to just the time
single_day['time'] = pd.to_datetime(single_day['time']).dt.time
# Set the index to time and sort by it
single_day = single_day.set_index('time')
# Sort it
single_day = single_day.sort_index()
```

Task 9: Instructions

Visualize the scores for the day and score we selected.

- Drop the `compound` and `headline` columns.
- Change the score columns names to `negative`, `positive`, and `neutral`.
- Plot a stacked bar chart with the result.

Note from the Instructor

Thanks for making it to the end! Here are a few plotting tips and the last plotting line for you as a bonus. :-)

Don't forget to add the colors and the title. Also, it does not hurt to make the plot bigger (if your screen resolution allows for it) and to add a legend on the side.

Pretty plots are not just for aesthetics, they also allow better communication and understanding of the data:

```
plot_day.plot.bar(stacked = True,
                  figsize=(10, 6),
                  title = TITLE,
                  color = COLORS).legend(bbox_to_anchor=(1.2, 0.5))
```

pandas does not like annotating the y-axis, but I do. Also, a semicolon after the last Matplotlib line of code will stop Jupyter from rendering the plot object string name:

```
plt.ylabel("scores");
```

I hope you had fun going through this project! If you are interested in the topic, here's some more info on NLTK.

-

Solution

```
TITLE = "Negative, neutral, and positive sentiment for FB on 2019-01-03"
COLORS = ["red","orange", "green"]
# Drop the columns that aren't useful for the plot
plot_day = single_day.drop(['compound', 'headline'], 1)
# Change the column names to 'negative', 'positive', and 'neutral'
plot_day.columns = ['negative', 'neutral', 'positive']
# Plot a stacked bar chart
plot_day.plot.bar(stacked = True, figsize=(10, 6), title = TITLE, color =
COLORS).legend(bbox_to_anchor=(1.2, 0.5))
plt.ylabel("scores");
```

Data Science for Social Good: Crime Study

Task 1: Instructions

Prepare your environment by loading packages and data.

- Load the `tidyverse` and `lubridate` packages.
- Using `read_csv()` load `datasets/downsample_police-department-incidents.csv` and assign to the variable `incidents`.
- Using `read_csv()` load `datasets/downsample_police-department-calls-for-service.csv` and assign to the variable `calls`.

Note: this project is soft launched, which means you may experience some bugs. Please click "Report an Issue" in the top-right corner of the screen to provide feedback.

Good to know

Before starting this project you should be comfortable manipulating and summarizing data with `dplyr`, as well as joing data. Familiarity with the syntax of `ggplot2` will be helpful. We recommended the following courses for a primer on exploratory data analysis:

- Data Manipulation with dplyr
- Introduction to Data Visualization with ggplot2
- Chapter 1 of Working with Geospatial Data in R

Helpful links:

- tidyverse cheat sheet
- dplyr join functions cheat sheet
- What do backticks do in R?
- The original and expanded data source

Code samples will often utilize the pipe operator (`%>%`), which is an integral part of the `dplyr` package. You can find more information about the pipe here.

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load required packages
library(tidyverse)
library(lubridate)
```

```
# Read datasets and assign to variables
incidents <- read_csv("datasets/downsample_police-department-
incidents.csv")
calls <- read_csv("datasets/downsample_police-department-calls-for-
service.csv")
print('Done!')

str(incidents)
str(calls)
```

Task 2: Instructions

Inspect the data and generate a frequency statistic.

- `glimpse()` the `incidents` and `calls` datasets to understand their structure.
- `count()` the number of reported incidents by `Date` and rename the column of counts to `n_incidents`. Assign the output to `daily_incidents`.
- `count()` the number of civilian calls for police service by `Date` and rename the column of counts to `n_calls`. Assign the output to `daily_calls`.

Inspecting the data allows you to understand the naming scheme of columns as well as data dimensionality. This is essential as variables in distinct datasets may reference the same measures but have different names, such as the date columns of interest in this case. In addition, be wary of [how to reference column names with spaces]().

-

Solution

```
# Glimpse the structure of both datasets
glimpse(incidents)
glimpse(calls)

# Aggregate the number of reported incidents by date
daily_incidents <- incidents %>%
    count(Date, sort = TRUE) %>%
    rename(n_incidents = n)

# Aggregate the number of calls for police service by date
daily_calls <- calls %>%
    count(Date, sort = TRUE) %>%
    rename(n_calls = n)
```

Task 3: Instructions

Join the datasets by date.

- Use `inner_join()` to join `daily_calls` to `daily_incidents` and assign the output to a variable named `shared_dates`.

- Inspect the new data frame structure.

`dplyr` has four mutating join functions: `inner_join()`, `left_join()`, `right_join()` and `full_join()`. It can be confusing as to which function to choose to accomplish your data analysis goals and may require some trial and error to understand how the functions differ.

Helpful links:

- `dplyr`'s `inner_join()` function documentation
-

Solution

```
# Join data frames to create a new "mutated" set of information
shared_dates <- inner_join(daily_incidents, daily_calls, by = c("Date" =
"Date"))

# Take a glimpse of this new data frame
glimpse(shared_dates)
```

Task 4: Instructions

Reshape the data and visualize the trends for incidents and calls on the same graph.

- Create a "long format" data frame called `plot_shared_dates` from the `shared_dates` data frame by using `gather()`. Name `key = report` and `value = count`.
- Use `ggplot()` to visualize `Date` vs. `count` of `plot_shared_dates`, and color by `report`. Overlay a linear model to visualize trends in the data.

The `gather()` function can be tricky, as it is not immediately intuitive. You can leave variables out of the `key` column when gathering by using `-`. Passing `-Date` means "do not select this column". In terms of visualization, a simple scatter plot may do well here. `ggplot2` has the awesome capability of creating linear models via the `geom_smooth()` function. For generalizability, you will want to use `method = "lm"` within the call to `geom_smooth()`.

Helpful links:

- `tidyr`'s `gather()` function documentation
- Section 12.3.1 of R for Data Science by Garrett Grolemund and Hadley Wickham
- `geom_smooth()` examples
-

Solution

```
# Gather into long format using the "Date" column to define observations
plot_shared_dates <- shared_dates %>%
  gather(key = report, value = count, -Date)

# Plot points and regression trend lines
ggplot(plot_shared_dates, aes(x = Date, y = count, color = report)) +
  geom_point() +
  geom_smooth(method = "lm", formula = y ~ x)
```

Task 5: Instructions

Calculate the correlation coefficient between the frequency of incidents and calls.

- Determine the correlation between the daily number of incidents and the daily number of calls in `shared_dates`. Assign the output to `daily_cor` and print it.
- Create a new column, `month`, from the `Date` column of `shared_dates`, and `group_by()` this new column in order to `summarize()` new frequency counts.
- Calculate the correlation coefficient between monthly frequencies. Assign to a new variable `monthly_cor` and print.

The `cor()` function allows you to calculate the correlation coefficient between two vectors of equal length. To easily extract units of date-time from the `Date` column, the `lubridate` package supplies many useful functions, such as `month()`. You will need to think about how to count the number of incidents and calls that occur each month by using `sum()` within `summarize()`.

Helpful links:

- `cor()` function documentation
- A useful Stack Overflow answer: How to extract Month from date in R
- `summarize()` function documentation

-

Solution

```
# Calculate correlation coefficient between daily frequencies
daily_cor <- cor(shared_dates$n_incidents, shared_dates$n_calls)
daily_cor

# Summarise frequencies by month
correlation_df <- shared_dates %>%
  mutate(month = month(Date)) %>%
  group_by(month) %>%
  summarize(n_incidents = sum(n_incidents),
            n_calls = sum(n_calls))

# Calculate correlation coefficient between monthly frequencies
```

```
monthly_cor <- cor(correlation_df$n_incidents, correlation_df$n_calls)
monthly_cor
```

Task 6: Instructions

Filter the `incidents` and `calls` data frames by their `shared_dates`.

- Use `semi_join()`, a filtering join, to subset the `calls` dataset.
- Make sure the sanity check prints `TRUE` in order to ensure you are using the `semi_join()` function appropriately.
- Use a filtering join to subset the `incidents` dataset.

`dplyr` has two types of filtering joins. To subset data based on another data frame, you will use either `semi_join()` or `anti_join()`. These preserve cases from the left-hand data frame while respectively keeping or throwing out matching rows. It is good practice to check you are using `dplyr`'s joins correctly if you are unsure of their application. With `identical()`, you can check if the dates in `calls_shared_dates` match the dates in the object we used to filter `calls`.

-

  Solution

```
# Filter calls to police by shared_dates
calls_shared_dates <- calls %>%
  semi_join(shared_dates, by = c("Date" = "Date"))

# Perform a sanity check that we are using this filtering join function
appropriately
identical(sort(unique(shared_dates$Date)),
sort(unique(calls_shared_dates$Date)))

# Filter recorded incidents by shared_dates
incidents_shared_dates <- incidents %>%
  semi_join(shared_dates, by = c("Date" = "Date"))
```

Task 7: Instructions

Rank the top call and incident crime types by frequency and visualize with a histogram.

- Subset the top 15 crime types of `calls_shared_dates` in descending order by count and use the pipe to pass this information to `ggplot()` and visualize using `geom_bar(stat = "identity")`.
- Subset the top 15 crime types of `incidents_shared_dates` in descending order by count and use the pipe to pass this information to `ggplot()` and visualize using `geom_bar(stat = "identity")`.
- Output the saved plots.

To filter only the top frequency call and incident crime types, you will have to `count()` the data and use `top_n()` to select the top 15 ranked crime types. Histograms can be tricky to order correctly when working with categorical data types such as factors. A useful way to obtain a descending order of crime types is to use the `reorder()` function to reset the order of your crime type column by your counts column (`n`).

Helpful links:

- `reorder()` function documentation.
- `ggplot2`'s `geom_bar()` function documentation.
-

Solution

```
# Create a bar chart of the number of calls for each crime
plot_calls_freq <- calls_shared_dates %>%
  count(Descript) %>%
  top_n(15, n) %>%
  ggplot(aes(x = reorder(Descript, n), y = n)) +
  geom_bar(stat = 'identity') +
  ylab("Count") +
  xlab("Crime Description") +
  ggtitle("Calls Reported Crimes") +
  coord_flip()


# Create a bar chart of the number of reported incidents for each crime
plot_incidents_freq <- incidents_shared_dates %>%
  count(Descript) %>%
  top_n(15, n)  %>%
  ggplot(aes(x = reorder(Descript, n), y = n)) +
  geom_bar(stat = 'identity') +
  ylab("Count") +
  xlab("Crime Description") +
  ggtitle("Incidents Reported Crimes") +
  coord_flip()

# Output the plots
plot_calls_freq
plot_incidents_freq
```

Task 8: Instructions

Compare the top locations of stolen vehicles to civilian reports of stolen vehicles.

- `filter()` for the appropriate crime type in `calls_shared_dates` and `count()` the location of the address variable. Assign the output to `location_calls`.
- `filter()` for the appropriate crime type in `incidents_shared_dates` and `count()` the location of the address variable. Assign the output to `location_incidents`.
- Print the top 10 locations for each dataset to compare.

As previously mentioned, comparing tables is an inefficient way to understand data patterns. However, tables can be useful when manual comparison is required due to non-standard data types. In this case it will be useful to look at the top 10 addresses where auto thefts occur for each dataset, as the addresses are not standardized between the two sets of information. You will want to `arrange()` the data in descending order by location count, and then subset the `top_n()` locations.

-

Solution

```
# Arrange the top 10 locations of called in crimes in a new variable
location_calls <- calls_shared_dates %>%
  filter(Descript == "Auto Boost / Strip") %>%
  count(Address) %>%
  arrange(desc(n))%>%
  top_n(10, n)

# Arrange the top 10 locations of reported incidents in a new variable
location_incidents <- incidents_shared_dates %>%
  filter(Descript == "GRAND THEFT FROM LOCKED AUTO") %>%
  count(Address) %>%
  arrange(desc(n))%>%
  top_n(10, n)

# Output the top locations of each dataset for comparison
location_calls
location_incidents
```

Task 9: Instructions

Visualize a 2D density plot on a map of San Francisco.

- Load `ggmap`.
- Read in a preprocessed map of San Francisco as `sf_map`.
- Use `filter()` to subset `incidents_shared_dates` by grand theft auto and save this data frame as `auto_incidents`.
- Use `ggmap()` to plot the map of San Francisco, and overlay `auto_incidents` latitude and longitude data using `stat_density_2d()`.

The `ggmap` package provides a suite of tools for plotting geospatial data. In this case, a map of San Francisco is already saved for you. You will need to read this object in and pass it to `ggmap()`. Graphics layers, such as `stat_density_2d()` can be added with a `+`, just like in `ggplot2`.

Helpful links:

- Vignette for `ggmap`.
- A blog using `ggmap` to plot data in a variety of ways.

-

Solution

```
# load ggmap
library(ggmap)

# Read in a static map of San Francisco
sf_map <- readRDS("datasets/sf_map.RDS")

# Filter grand theft auto incidents
auto_incidents <- incidents_shared_dates %>%
  filter(Descript == "GRAND THEFT FROM LOCKED AUTO")

# Overlay a density plot of auto incidents on the map
ggmap(sf_map) +
  stat_density_2d(
    aes(x = X, y = Y, fill = ..level..), alpha = 0.15,
    size = 0.01, bins = 30, data = auto_incidents,
    geom = "polygon")
```

Planning Public Policy in Argentina

Task 1: Instructions

Load the libraries and the dataset for the project.

- Load in the `tidyverse` package.
- Read in `datasets/argentina.csv` using `read_csv` and assign it to the variable `argentina`.
- Check the number of rows with `nrow()` and print the top rows of `argentina` using `head()`.

Good to know

The data for this project was published by INDEC, the agency in charge of Argentina's official statistics. The dataset has 22 rows because Tierra del Fuego is not included and the autonomous city of Buenos Aires is merged with the rest of the province of Buenos Aires.

Make sure you use `read_csv()` from `readr`, not `read.csv()` from `utils`, to read in the data.

This project assumes you can manipulate data frames using `dplyr`, make simple plots using `ggplot2`, and understand principal component analysis and K-Means clustering. You can learn these skills from DataCamp's Introduction to the Tidyverse and Unsupervised Learning in R.

-

Solution

```
# Load the tidyverse
library(tidyverse)

# Read in the dataset
argentina <- read_csv("datasets/argentina.csv")

# Inspect the first rows of the dataset
nrow(argentina)
head(argentina)
```

Task 2: Instructions

Find the four richest provinces and the provinces with more than one million people.

- Add `gdp_per_cap` to `argentina` by dividing `gdp` by `pop`.

- Create and print `rich_provinces` by arranging `argentina` by `gdp_per_cap` in descending order, selecting `province` and `gdp_per_cap`, and taking the top four rows using `top_n()`.
- Create and print `bigger_pops` by arranging `argentina` by `pop` in descending order, selecting `province` and `pop`, and filtering for `pop` greater than one million.

Adding parentheses around a variable is a shorthand way to print the variable.

-

Solution

```
# Add gdp_per_capita column to argentina
argentina <- argentina %>%
  mutate(gdp_per_cap = gdp / pop)

# Find the four richest provinces
( rich_provinces  <- argentina %>%
  arrange(-gdp_per_cap) %>%
  select(province, gdp_per_cap) %>%
  top_n(4) )

# Find the provinces with populations over 1 million
( bigger_pops <- argentina %>%
  arrange(-pop) %>%
  select(province, pop) %>%
  filter(pop > 1e6) )
```

Task 3: Instructions

Convert the numeric columns of `argentina` to a matrix.

- Use `select_*()` with `is.numeric()` to select the numeric columns in `argentina` and pipe the result to `as.matrix()`.
- Print the first few rows using `head()`.

Helpful links:

- Check the examples on this page if you are not familiar with `select_if()`.

-

Solution

```
# Select numeric columns and cast to matrix
argentina_matrix <- argentina %>%
  select_if(is.numeric) %>%
  as.matrix
```

```
# Print the first lines of the result
head(argentina_matrix)
```

Task 4: Instructions

Apply Principal Component Analysis to the data.

- Load `FactoMineR`.
- Apply `PCA()` to the to `argentina_matrix`, assign the output to `argentina_pca` and print its contents.

Remember to scale the data before running PCA by setting `scale.unit=TRUE`.

Helpful links:

- FactoMineR: PCA
- Dimensionality reduction from DataCamp's Unsupervised Learning in R course
-

Solution

```
# Load FactoMineR
library(FactoMineR)

# Apply PCA and print results
( argentina_pca <- PCA(argentina_matrix, scale.unit = TRUE) )
```

Task 5: Instructions

Plot the correlation circle and calculate the proportion of total variance preserved by the first two principal components.

- Load the `factoextra` package.
- Use `fviz_pca_var()` with `argentina_pca` as input to plot the correlation circle and assign the plot to `pca_var_plot`.
- Sum the variance preserved by the first two components from `argentina_pca` and store the result in `variance_first_two_components`.

The output from `PCA()` is a nested list. The first component of this list, `argentina_pca$eig` is a matrix with rows representing each principal component and columns reporting key metrics for each component. Use `str()` and `colnames` to understand its structure.

Helpful links:

- Plotting PCA output with factoextra
- Advanced R's chapter on subsetting R objects

-

Solution

```
# Load factoextra
library(factoextra)

# Set the size of plots in this notebook
options(repr.plot.width=7, repr.plot.height=5)

# Plot the original variables and the first 2 components and print the plot
object
( pca_var_plot <- fviz_pca_var(argentina_pca) )

# Sum the variance preserved by the first two components. Print the result.
( variance_first_two_pca <- argentina_pca$eig[1,2] + argentina_pca$eig[2,2]
)
```

Task 6: Instructions

Plot the position of the provinces in the first two principal components.

- Use `fviz_pca_ind()` with `argentina_pca` as input to visualize the position of provinces in the first two components. Add "Provinces - PCA" setting the `title` argument.

Remember that "individuals" in the terminology of `factoMineR` are the provinces.

-

Solution

```
# Visualize Dim2 vs. Dim1
fviz_pca_ind(argentina_pca, title = "Provinces - PCA")
```

Task 7: Instructions

Partition the provinces into four clusters.

- Set the seed to `1234` to ensure reproducibility.
- Use `tibble` to create a data frame called `argentina_comps` with the coordinates of each province in the first two principal components. Extract this coordinates from `argentina_pca`.
- Use `kmeans()` with `argentina_comps` as input. Set `centers = 4` to create four clusters, also set `nstart = 20` and `iter.max = 50`.

The coordinates of the provinces in the principal components are available in the `argentina_pca$ind$coord` matrix. Each row corresponds to an individual (or province) and each column is the position of that province in the corresponding component.

-

Solution

```
# Set seed to 1234 for reproducibility
set.seed(1234)

# Create an intermediate data frame with pca_1 and pca_2
argentina_comps <- tibble(pca_1 = argentina_pca$ind$coord[,1],
                          pca_2 = argentina_pca$ind$coord[,2])

# Cluster the observations using the first 2 components and print its
contents
( argentina_km <- kmeans(argentina_comps, centers = 4, nstart = 20,
iter.max = 50) )
```

Task 8: Instructions

Recreate the plot from Task 6, coloring each point by its assigned cluster.

- Extract the vector of clusters assigned to each province from `argentina_km` and convert it to a factor using `factor`. Store this value in a variable named `clusters_as_factor`.
- Use `fviz_pca_ind` again to recreate the plot from Task 6 passing `clusters_as_factor` in the `habillage` argument. Set the title to "Clustered Provinces - PCA".

You can find the vector of assigned clusters in `argentina_km$cluster`.

Helpful links:

- Visualizing PCA with `factoextra`

-

Solution

```
# Convert assigned clusters to factor
clusters_as_factor <- factor(argentina_km$cluster)

# Plot individulas colored by cluster
fviz_pca_ind(argentina_pca,
             title = "Clustered Provinces - PCA",
             habillage=clusters_as_factor)
```

Task 9: Instructions

Explore the variability in `gdp` between clusters with a scatterplot.

- Load `ggrepel`.
- Use `mutate()` to add `clusters_as_factor` as a column in `argentina`.
- Use `ggplot()` with a `geom_point()` layer to create a scatterplot of `gdp` vs. `cluster`, coloring the points by `cluster` for readability. Add labels for the x and y axes and annotate each point with its `province` name using `geom_text_repel()`.

Remember that plotting `y` vs. `x` means that `x` is mapped to the horizontal axis and `y` to the vertical axis.

Helpful links:

- Creating a scatter plot with ggplot
- How to use `ggrepel`

-

Solution

```
# Load ggrepel
library(ggrepel)

# Add cluster column to argentina
argentina <- argentina %>%
               mutate(cluster=clusters_as_factor)


# Make a scatterplot of gdp vs. cluster, colored by cluster
ggplot(argentina, aes(cluster, gdp, color=cluster)) +
  geom_point() +
  geom_text_repel(aes(label=province), show.legend=FALSE) +
  labs(x="Cluster", y="GDP")
```

Task 10: Instructions

Explore the variability in `gdp_per_cap` between clusters with a scatterplot.

- Use `ggplot()` with a `geom_point()` layer to create a scatterplot of `gdp_per_cap` vs. `cluster`. Add labels to the x and y axes, color the points by `cluster` and annotate them with `province` using `geom_text_repel()`.

It is widely believed that Argentina's previous government - led by Cristina Fernández - tampered with the official statistics published by INDEC in order to lower debt payments tied

to the inflation rate. The agency was overhauled by the new government and confidence in the published statistics was reestablished since, but you can never be too careful with the sources of your data when doing data analysis!

-

Solution

```
# Make a scatterplot of GDP per capita vs. cluster, colored by cluster
ggplot(argentina, aes(cluster, gdp_per_cap, color=cluster)) +
  geom_point() +
  geom_text_repel(aes(label=province), show.legend=FALSE) +
  labs(x="Cluster", y="GDP per capita")
```

Task 11: Instructions

Explore the variability in `poverty` between clusters with a scatterplot.

- Use `ggplot()` with a `geom_point()` layer to create a scatterplot of `poverty` vs. `cluster`. Add labels to the x and y axes, color the points by `cluster` and annotate them with `province` using `geom_text_repel()`.

By 1900, Argentina had one of the top ten highest per capita GDP in the world, surpassing France and Italy. After a century of disappointing economic performance, however, its current ranking in this index is 62, well below countries in the developed world.

-

Solution

```
# Make scatterplot of poverty vs. cluster, colored by cluster
ggplot(argentina, aes(cluster, poverty, color = cluster)) +
  geom_point() +
  geom_text_repel(aes(label=province), show.legend = FALSE) +
  labs(x="Cluster", y="Poverty rate")
```

Task 12: Instructions

Determine what provinces will participate in the pilot project.

- Assign the proposal number (1, 2, or 3) with the most diverse set of provinces to `pilot_provinces`.

-

Solution

```
# Assign pilot provinces to the most diverse group
pilot_provinces <- 3
```

Clustering Bustabit Gambling Behavior

Task 1: Instructions

Perform an initial exploratory data analysis of the Bustabit data.

- Load the `tidyverse` package.
- Read in the Bustabit gambling data (`datasets/bustabit.csv`) using the `read_csv()` function.
- Look at the first five rows of the data.
- Find the highest multiplier, `BustedAt`, ever achieved in a game.

Good to know

This Project lets you apply the skills from Introduction to the Tidyverse, including filtering, grouping and summarizing, and visualizing data with ggplot2. Introduction to Function Writing in R is also suggested as a prerequisite. Basic knowledge and understanding of the concepts of clustering (see Chapter 3 of Cluster Analysis in R) is a plus but is not required.

Helpful links:

- Tidyverse cheat sheet
- dplyr's `arrange()` function documentation

-

Solution

```
# Load the tidyverse
library(tidyverse)

# Read in the bustabit gambling data
bustabit <- read_csv("datasets/bustabit.csv")

# Look at the first five rows of the data
head(bustabit, n = 5)

# Find the highest multiplier (BustedAt value) achieved in a game
bustabit %>%
    arrange(desc(BustedAt)) %>%
    slice(1)
```

Task 2: Instructions

Derive additional features from the variables provided.

- Derive and create the new feature variables.

- Look at the first five rows of the features data.

Helpful links:

- dplyr's `mutate()` function documentation
- Mutate exercises in the Introduction to the Tidyverse course

-

Solution

```
# Create the new feature variables
bustabit_features <- bustabit %>%
  mutate(CashedOut = ifelse(is.na(CashedOut), BustedAt + .01, CashedOut),
         Profit = ifelse(is.na(Profit), 0, Profit),
         Losses = ifelse(Profit == 0, -1 * Bet, 0),
         GameWon = ifelse(Profit == 0, 0, 1),
         GameLost = ifelse(Profit == 0, 1, 0))

# Look at the first five rows of the data
head(bustabit_features, n = 5)
```

Task 3: Instructions

Create a per-player features dataset for clustering purposes.

- Group by the players and summarize to create per-player statistics.
- View the first five rows of the data.

Helpful links:

- dplyr's `summarise()` function documentation

-

Solution

```
# Group by players to create per-player summary statistics
bustabit_clus <- bustabit_features %>%
  group_by(Username) %>%
  summarize(AverageCashedOut = mean(CashedOut),
            AverageBet = mean(Bet),
            TotalProfit = sum(Profit),
            TotalLosses = sum(Losses),
            GamesWon = sum(GameWon),
            GamesLost = sum(GameLost))

# View the first five rows of the data
head(bustabit_clus, n = 5)
```

Task 4: Instructions

Standardize the numeric variables in preparation for clustering.

- Create a function that takes a numeric vector and performs mean-sd standardization (computes a Z-score).
- Apply the function to each numeric variable in `bustabit_clus`.
- Summarize our standardized data.

Helpful links:

- General information on statistical normalization
- Specific information about Z-scores

-

Solution

```
# Create the mean-sd standardization function
mean_sd_standard <- function(x) {
    (x - mean(x)) / sd(x)
}

# Apply this function to each numeric variable in the clustering set
bustabit_standardized <- bustabit_clus %>%
    mutate_if(is.numeric, mean_sd_standard)

# Summarize our standardized data
summary(bustabit_standardized)
```

Task 5: Instructions

Use the `kmeans()` function to cluster Bustabit players.

- Choose 20190101 as our random seed.
- Cluster the `bustabit_standardized` data using `kmeans()` with five clusters.
- Store the cluster assignments back into the clustering data frame object as a **factor**.
- Look at the distribution of cluster assignments.

Remember to remove the `Username` variable, the first column, because `kmeans()` expects only numeric data.

Helpful links:

- `kmeans()` function documentation
- Random number generation documentation

-

Solution

```
# Choose 20190101 as our random seed
set.seed(20190101)

# Cluster the players using kmeans with five clusters
cluster_solution <- kmeans(bustabit_standardized[,-1], centers = 5)

# Store the cluster assignments back into the clustering data frame object
bustabit_clus$cluster <- factor(cluster_solution$cluster)

# Look at the distribution of cluster assignments
table(bustabit_clus$cluster)
```

Task 6: Instructions

Analyze the clusters by observing group averages across the variables used to cluster.

- Group the clustering data frame by the cluster assignment and use `summarize_if()` to average each numeric variables across each cluster assignment.
- View the resulting table.

Helpful links:

- dplyr's `summarise_if()` function documentation
- Suzan Baert's `summarise_if()` function tutorial

-

Solution

```
# Group by the cluster assignment and calculate averages
bustabit_clus_avg <- bustabit_clus %>%
    group_by(cluster) %>%
    summarize_if(is.numeric, mean)

# View the resulting table
bustabit_clus_avg
```

Task 7: Instructions

Visualize the clustering solution with a PCP.

- Create a function that takes a numeric vector and performs min-max scaling.
- Apply the function to each numeric variable in the `bustabit_clus_avg` object.

- Load the `GGally` package.
- Use the `ggparcoord()` function to create a parallel coordinate plot of the values.

Min-max standardization is performed by taking the vector of values, subtracting the minimum, and then dividing by the range (max - min).

Helpful links:

- The `ggparcoord()` function documentation
-

Solution

```
# Create the min-max scaling function
min_max_standard <- function(x) {
    (x - min(x)) / (max(x) - min(x))
}

# Apply this function to each numeric variable in the bustabit_clus_avg
object
bustabit_avg_minmax <- bustabit_clus_avg %>%
    mutate_if(is.numeric, min_max_standard)

# Load the GGally package
library(GGally)

# Create a parallel coordinate plot of the values
ggparcoord(bustabit_avg_minmax, columns = 2:ncol(bustabit_avg_minmax),
           groupColumn = "cluster", scale = "globalminmax", order =
"skewness")
```

Task 8: Instructions

Produce a plot of the clustering solution across the two Principal Components.

- Calculate the principal components of the `bustabit_standardized` data (**numeric columns only**) and store them as a data frame.
- Assign the cluster assignments to the new data frame.
- Use `ggplot()` to produce a scatterplot (`geom_point()`) of PC2 vs. PC1, and color by cluster assignment.
- View the resulting plot.

Helpful links:

- The ggplot2 cheatsheet
- Further information on plotting a Principal Component Analysis

-

Solution

```
# Extract the first two principle components from the data
my_pc <- as.data.frame(prcomp(bustabit_standardized[,-1])$x)

# Store the cluster assignment in this object
my_pc$cluster <- bustabit_clus$cluster

# Use ggplot to plot PC2 vs PC1, and color by the cluster assignment
p1 <- ggplot(data = my_pc, aes(x = PC1, y = PC2, color = cluster)) +
    geom_point()

# View the resulting plot
p1
```

Task 9: Instructions

Assign cluster assignments to the Bustabit gambling user groups.

- Use the parallel coordinate plot and cluster means table to match the cluster descriptions from above to the cluster assignments.
- Append the cluster names to cluster means table (`bustabit_clus_avg`).
- View the cluster means table with your appended cluster names.

Congratulations, you've just completed a clustering analysis! In a real-world application of this procedure, your likely next step would be to take the cluster assignments for all 4000+ players and assign them back into the original data. From there, you can explore the data again with your cluster assignments in mind. What games did the Risk Takers participate in? Are the Strategic Addicts playing most consistently over time? Do High Rollers indeed cash out at the highest multiplier values? All these questions and much more are available for you to explore now that you've assigned and interpreted your clustering solution. We encourage you to see what you can find!

Helpful links:

- Berkeley's documentation on interpreting a cluster analysis

-

Solution

```
# Assign names to clusters 1 through 5 in order
cluster_names <- c(
    "Risky Commoners",
    "High Rollers",
    "Risk Takers",
    "Cautious Commoners",
    "Strategic Addicts"
)
```

```
# Append the cluster names to the cluster means table
bustabit_clus_avg_named <- bustabit_clus_avg %>%
    cbind(Name = cluster_names)

# View the cluster means table with your appended cluster names
bustabit_clus_avg_named
```

Give Life: Predict Blood Donations

Task 1: Instructions

Inspect the file that contains the dataset.

- Print out the first 5 lines from `datasets/transfusion.data` using the `head` shell command.

Good to know

Welcome to the Project! Make sure to first read the narrative for each task in the notebook on the right before reading the more detailed instructions here.

To complete this Project, you need to know some Python, pandas, and logistic regression. We recommend one is familiar with the content in DataCamp's Data Manipulation with pandas, Preprocessing for Machine Learning in Python, and Introduction to Predictive Analytics in Python courses.

To run a shell command in a notebook, you prefix it with `!`, e.g. `!ls` will list directory contents.

Helpful links for this task:

- Inspecting the start of a file exercise
-

Solution

```
# Print out the first 5 lines from the transfusion.data file
!head -n5 datasets/transfusion.data
```

Task 2: Instructions

Load the dataset.

- Import the pandas library.
- Load the `transfusion.data` file from `datasets/transfusion.data` and assign it to the `transfusion` variable.
- Display the first rows of the DataFrame with the `head()` method to verify the file was loaded correctly.

If you print the first few rows of data, you should see a table with only 5 columns.

Helfpul links:

- Loading a CSV file with pandas exercise
-

Solution

```
# Import pandas
import pandas as pd

# Read in dataset
transfusion = pd.read_csv('datasets/transfusion.data')

# Print out the first rows of our dataset
transfusion.head
```

Task 3: Instructions

Inspect the DataFrame's structure.

- Print a concise summary of the `transfusion` DataFrame with the `info()` method.

DataFrame's `info()` method prints some useful information about a DataFrame:

- index type
- column types
- non-null values
- memory usageincluding the index dtype and column dtypes, non-null values and memory usage.
-

Solution

```
# Print a concise summary of transfusion DataFrame
transfusion.info()
```

Task 4: Instructions

Rename a column.

- Rename `whether he/she donated blood in March 2007` to `target` for brevity.
- Print the first **2** rows of the DataFrame with the `head()` method to verify the change was done correctly.

By setting the `inplace` parameter of the `rename()` method to `True`, the `transfusion` DataFrame is changed in-place, i.e., the `transfusion` variable will now point to the updated DataFrame as you'll verify by printing the first 2 rows.

-

Solution

```
# Rename target column as 'target' for brevity
transfusion.rename(
    columns={'whether he/she donated blood in March 2007': 'target'},
    inplace=True
)

# Print out the first 2 rows
transfusion.head(2)
```

Task 5: Instructions

Print target incidence.

- Use `value_counts()` method on `transfusion.target` column to print target incidence proportions, setting `normalize=True` and rounding the output to 3 decimal places.

By default, `value_counts()` method returns counts of unique values. By setting `normalize=True`, the `value_counts()` will return the relative frequencies of the unique values instead.

You can learn more about target incidence in [Introduction to Predictive Analytics in Python] course.

-

Solution

```
# Print target incidence proportions, rounding output to 3 decimal places
transfusion.target.value_counts(normalize=True).round(3)
```

Task 6: Instructions

Split the `transfusion` DataFrame into train and test datasets.

- Import `train_test_split` from `sklearn.model_selection` module.
- Split `transfusion` into `X_train`, `X_test`, `y_train` and `y_test` datasets, stratifying on the `target` column.
- Print the first 2 rows of the `X_train` DataFrame with the `head()` method.

Writing the code to split the data into the 4 datasets needed would require a lot of work. Instead, you will use the `train_test_split()` method in the `scikit-learn` library.

For more on data preprocessing and why it is necessary, check out the [Preprocessing for Machine Learning in Python] course.

-

Solution

```
# Import train_test_split method
from sklearn.model_selection import train_test_split

# Split transfusion DataFrame into
# X_train, X_test, y_train and y_test datasets,
# stratifying on the `target` column
X_train, X_test, y_train, y_test = train_test_split(
    transfusion.drop(columns='target'),
    transfusion.target,
    test_size=0.25,
    random_state=42,
    stratify=transfusion.target
)

# Print out the first 2 rows of X_train
X_train.head(2)
```

Task 7: Instructions

Use the TPOT library to find the best machine learning pipeline.

- Import `TPOTClassifier` from `tpot` and `roc_auc_score` from `sklearn.metrics`.
- Create an instance of `TPOTClassifier` and assign it to `tpot` variable.
- Print `tpot_auc_score`, rounding it to 4 decimal places.
- Print `idx` and `transform` in the for-loop to display the pipeline steps.

You will adapt the classification example from the TPOT's documentation. In particular, you will specify `scoring='roc_auc'` because this is the metric that you want to optimize for and add `random_state=42` for reproducibility. You'll also use `TPOT light` configuration with only fast models and preprocessors.

The nice thing about TPOT is that it has the same API as `scikit-learn`, i.e., you first instantiate a model and then you train it, using the `fit` method.

Data pre-processing affects the model's performance, and `tpot`'s `fitted_pipeline_` attribute will allow you to see what pre-processing (if any) was done in the best pipeline.

Helpful links:

- TPOT tutorial
- Format strings using Python 3's f-strings tutorial
-

Solution

```
# Import TPOTClassifier
from tpot import TPOTClassifier
# Import roc_auc_score
from sklearn.metrics import roc_auc_score

# Instantiate TPOTClassifier
tpot = TPOTClassifier(
    generations=5,
    population_size=20,
    verbosity=2,
    scoring='roc_auc',
    random_state=42,
    disable_update_check=True,
    config_dict='TPOT light'
)
tpot.fit(X_train, y_train)

# AUC score for tpot model
tpot_auc_score = roc_auc_score(y_test, tpot.predict_proba(X_test)[:, 1])
print(f'\nAUC score: {tpot_auc_score:.4f}')

# Print best pipeline steps
print('\nBest pipeline steps:', end='\n')
for idx, (name, transform) in enumerate(tpot.fitted_pipeline_.steps,
start=1):
    # Print idx and transform
    print(f'{idx}. {transform}')
```

Task 8: Instructions

Check the variance.

- Print `X_train`'s variance using `var()` method and round it to 3 decimal places.

`pandas.DataFrame.var()` method returns column-wise variance of a DataFrame, which makes comparing the variance across the features in `X_train` simple and straightforward.

-

Solution

```
# X_train's variance, rounding the output to 3 decimal places
X_train.var().round(3)
```

Task 9: Instructions

Correct for high variance.

- Copy `X_train` and `X_test` into `X_train_normed` and `X_test_normed` respectively.
- Assign the column name (a string) that has the highest variance to `col_to_normalize` variable.
- For `X_train` and `X_test` DataFrames:
  - Log normalize `col_to_normalize` to add it to the DataFrame.
  - Drop `col_to_normalize`.
- Print `X_train_normed` variance using `var()` method and round it to 3 decimal places.

`X_train` and `X_test` must have the same structure. To keep your code "DRY" (Don't Repeat Yourself), you are using a for-loop to apply the same set of transformations to each of the DataFrames.

Normally, you'll do pre-processing *before* you split the data (it could be one of the steps in machine learning pipeline). Here, you are testing various ideas with the goal to improve model performance, and therefore this approach is fine.

-

Solution

```
# Import numpy
import numpy as np

# Copy X_train and X_test into X_train_normed and X_test_normed
X_train_normed, X_test_normed = X_train.copy(), X_test.copy()

# Specify which column to normalize
col_to_normalize = 'Monetary (c.c. blood)'

# Log normalization
for df_ in [X_train_normed, X_test_normed]:
    # Add log normalized column
    df_['monetary_log'] = np.log(df_[col_to_normalize])
    # Drop the original column
    df_.drop(columns=col_to_normalize, inplace=True)

# Check the variance
X_train_normed.var().round(3)
```

Task 10: Instructions

Train the logistic regression model.

- Import `linear_model` from `sklearn`.
- Create an instance of `linear_model.LogisticRegression` and assign it to `logreg` variable.
- Train `logreg` model using the `fit()` method.
- Print `logreg_auc_score`.

The `scikit-learn` library has a consistent API when it comes to fitting a model:

1. Create an instance of a model you want to train.
2. Train it on your `train` datasets using the `fit` method.

You may recognise this pattern from when you trained TPOT model. This is the beauty of the `scikit-learn` library: you can quickly try out different models with only a few code changes.

Foundations of Predictive Analytics in Python (Part 1) course is a great resource if you want to have more practice with linear regression models in Python.

-

Solution

```
# Importing modules
from sklearn import linear_model

# Instantiate LogisticRegression
logreg = linear_model.LogisticRegression(
    solver='liblinear',
    random_state=42
)

# Train the model
logreg.fit(X_train_normed, y_train)

# AUC score for tpot model
logreg_auc_score = roc_auc_score(y_test,
logreg.predict_proba(X_test_normed)[:, 1])
print(f'\nAUC score: {logreg_auc_score:.4f}')
```

Task 11: Instructions

Sort your models based on their AUC score from highest to lowest.

- Import `itemgetter` from `operator` module.

- Sort the list of `(model_name, model_score)` pairs from highest to lowest using `reverse=True` parameter.

Congratulations, you've made it to the end! If you haven't tried it already, you can check your Project by clicking the 'Check Project' button.

Good luck and keep on learning!

If you are interested in learned what makes linear models so powerful and widely used, Statistical Modeling in R is a great resource! The coding is done in `R`, but it's the theoretical concepts that will help you to interpret the models you are building.

-

Solution

```
# Importing itemgetter
from operator import itemgetter

# Sort models based on their AUC score from highest to lowest
sorted(
    [('tpot', tpot_auc_score), ('logreg', logreg_auc_score)],
    key=itemgetter(1),
    reverse=True
)
```

Find Movie Similarity from Plot Summaries

Task 1: Instructions

Import the necessary modules and load the dataset.

- Import `numpy` as `np`.
- Import `pandas` as `pd`.
- Import `nltk`.
- Read `datasets/movies.csv` into `movies_df` using the pandas `read_csv()` function.

This Project lets you apply the skills from [Natural Language Processing Fundamentals in Python] and [Unsupervised Learning in Python]. We recommend that you are familiar with the content in those courses before starting this Project.

Helpful links:

- pandas `read_csv()` function [documentation]

If you experience odd behavior you can reset the Project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the Project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Import modules
import numpy as np
import pandas as pd
import nltk

# Set seed for reproducibility
np.random.seed(5)

# Read in IMDb and Wikipedia movie data (both in same file)
movies_df = pd.read_csv("datasets/movies.csv")

print("Number of movies loaded: %s " % (len(movies_df)))

# Display the data
movies_df
```

Task 2: Instructions

Combine the two plot summary columns into one.

- Create a new column named `plot` by adding the text in `wiki_plot` and `imdb_plot` columns together.

Be sure to put `wiki_plot` first and `imdb_plot` second in your addition code.

-

Solution

```
# Combine wiki_plot and imdb_plot into a single column
movies_df["plot"] = movies_df["wiki_plot"].astype(str) + "\n" + \
                    movies_df["imdb_plot"].astype(str)

# Inspect the new DataFrame
movies_df.head()
```

Task 3: Instructions

Tokenize and filter sentences, first breaking them into sentences and then breaking them into words.

- Store tokenized sentences into `sent_tokenized` by using `nltk.sent_tokenize()` method on the paragraph.
- Store tokenized words into `words_tokenized` by using `nltk.word_tokenize()` method on the sentence.
- Filter out raw tokens and punctuation from the words using RegEx.

Helpful links:

- [Python Regular Expression Tutorial]

-

Solution

```
# Tokenize a paragraph into sentences and store in sent_tokenized
sent_tokenized = [sent for sent in nltk.sent_tokenize("""
                        Today (May 19, 2016) is his only daughter's
wedding.
                        Vito Corleone is the Godfather.
                        """)]

# Word Tokenize first sentence from sent_tokenized, save as words_tokenized
words_tokenized = [word for word in nltk.word_tokenize(sent_tokenized[0])]

# Remove tokens that do not contain any letters from words_tokenized
import re
```

```
filtered = [word for word in words_tokenized if re.search('[a-zA-Z]',
word)]

# Display filtered words to observe words after tokenization
filtered
```

Task 4: Instructions

Import the SnowBall Stemmer and perform stemming on the filtered words.

- Import the `SnowballStemmer` sub-module from the `nltk.stem.snowball` module.
- Perform stemming on the `filtered` word list using the `stem()` method of the `SnowballStemmer`.

Helpful links:

- NLTK stemmer how-to
-

Solution

```
# Import the SnowballStemmer to perform stemming
from nltk.stem.snowball import SnowballStemmer

# Create an English language SnowballStemmer object
stemmer = SnowballStemmer("english")

# Print filtered to observe words without stemming
print("Without stemming: ", filtered)

# Stem the words from filtered and store in stemmed_words
stemmed_words = [stemmer.stem(t) for t in filtered]

# Print the stemmed_words to observe words after stemming
print("After stemming:   ", stemmed_words)
```

Task 5: Instructions

Create a function that tokenizes and stems the words.

- Create a function `tokenize_and_stem()` which takes a single argument `text`.
- Sentence tokenize then word tokenize the text.
- Read the line of code provided that filters out the raw tokens from the text.
- Perform stemming on the filtered tokens.

-

Solution

```
# Define a function to perform both stemming and tokenization
def tokenize_and_stem(text):

    # Tokenize by sentence, then by word
    tokens = [word for sent in nltk.sent_tokenize(text) for word in
nltk.word_tokenize(sent)]

    # Filter out raw tokens to remove noise
    filtered_tokens = [token for token in tokens if re.search('[a-zA-Z]',
token)]

    # Stem the filtered_tokens
    stems = [stemmer.stem(t) for t in filtered_tokens]

    return stems

words_stemmed = tokenize_and_stem("Today (May 19, 2016) is his only
daughter's wedding.")
print(words_stemmed)
```

Task 6: Instructions

Create a `TfidfVectorizer` object.

- First, import the `TfidfVectorizer` from the `sklearn.feature_extraction.text` module.
- Next, while creating the `TfidfVectorizer` object, set the `stopwords` parameter to use the English Language and the `tokenizer` parameter to use our `tokenize_and_stem()` function.

Helpful links:

- `TfidfVectorizer` documentation

-

Solution

```
# Import TfidfVectorizer to create TF-IDF vectors
from sklearn.feature_extraction.text import TfidfVectorizer

# Instantiate TfidfVectorizer object with stopwords and tokenizer
# parameters for efficient processing of text
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
                                   min_df=0.2, stop_words='english',
                                   use_idf=True, tokenizer=tokenize_and_stem,
                                   ngram_range=(1,3))
```

Task 7: Instructions

Fit and transform the text to a vectorizer.

- Use the `fit_transform()` method of the `TfidfVectorizer` object that you created on the movie plots.

Helpful links:

- `TfidfVectorizer` documentation
-

Solution

```
# Fit and transform the tfidf_vectorizer with the "plot" of each movie
# to create a vector representation of the plot summaries
tfidf_matrix = tfidf_vectorizer.fit_transform([x for x in
movies_df["plot"]])

print(tfidf_matrix.shape)
```

Task 8: Instructions

Cluster the movies based on their similarity.

- First, use the `KMeans()` method to create a classifier object and store it as `km`.
- Next, fit the `tfidf_matrix` to the k-means classifier object using the method `fit()`.
- Now, store the created cluster labels into the `movies_df` DataFrame as a new column `cluster`.

-

Solution

```
# Import k-means to perform clustering
from sklearn.cluster import KMeans

# Create a KMeans object with 5 clusters and save as km
km = KMeans(n_clusters=5)

# Fit the k-means object with tfidf_matrix
km.fit(tfidf_matrix)

clusters = km.labels_.tolist()
```

```
# Create a column cluster to denote the generated cluster for each movie
movies_df["cluster"] = clusters

# Display number of films per cluster (clusters from 0 to 4)
movies_df['cluster'].value_counts()
```

Task 9: Instructions

Calculate the similarity distances between the movies.

- Import the `cosine_similarity()` method from `sklearn.metrics.pairwise` module.
- Use the `cosine_similarity()` method on `tfidf_matrix` to calculate the similarity distances.

-

Solution

```
# Import cosine_similarity to calculate similarity of movie plots
from sklearn.metrics.pairwise import cosine_similarity

# Calculate the similarity distance
similarity_distance = 1 - cosine_similarity(tfidf_matrix)
```

Task 10: Instructions

Importing plotting modules for later use.

- Import `matplotlib.pyplot` under the alias of `plt`.
- Import `linkage` and `dendrogram` from `scipy.cluster.hierarchy`.

Since a tree requires a clearly defined linking between its various leaves, we shall require the linkage function provided in `scipy.cluster.hierarchy`.

-

Solution

```
# Import matplotlib.pyplot for plotting graphs
import matplotlib.pyplot as plt

# Configure matplotlib to display the output inline
%matplotlib inline

# Import modules necessary to plot dendrogram
```

```
from scipy.cluster.hierarchy import linkage, dendrogram
```

Task 11: Instructions

Create the linking between the movies and plot the dendrogram for their similarity.

- Use the `linkage()` function on `similarity_distance` and store it as `mergings`.
- Plot the `mergings` using the `dendrogram()` function. You can use the `title` Series of `movies_df` DataFrame for labels.

Helpful links:

- Learn more about Clustering Methods with SciPy
-

Solution

```
# Create mergings matrix
mergings = linkage(similarity_distance, method='complete')

# Plot the dendrogram, using title as label column
dendrogram_ = dendrogram(mergings,
                labels=[x for x in movies_df["title"]],
                leaf_rotation=90,
                leaf_font_size=16,
)

# Adjust the plot
fig = plt.gcf()
_ = [lbl.set_color('r') for lbl in plt.gca().get_xmajorticklabels()]
fig.set_size_inches(108, 21)

# Show the plotted dendrogram
plt.show()
```

Task 12: Instructions

Which movies are most similar?

- Look at the dendrogram and find the name of the movie which is most similar to the movie *Braveheart*. Hint: look at the yellow leaves.
- Set `ans` to the name of the movie which is most similar to the movie *Braveheart*.

Congratulations on reaching the end of the Project! If you'd like to continue building your Python skills, all of DataCamp's Python courses are listed here.

Right-click on the dendrogram plot and open the image in a New Tab to be able to zoom onto the labels of the image. Alternatively, you can save the image and open it in any image viewer of your choice.

-

Solution

```
# Answer the question
ans = "Gladiator"
print(ans)
```

The Impact of Climate Change on Birds

Task 1: Instructions

Import the prepared climate data.

- Start by loading the relevant packages. We will need the `tidyverse`, `sf`, and `raster`.
- Read the `climate_raster.rds` file from the `datasets` folder and assign it to a variable, `climate`.
- Print the column names of `climate`.
- Understand the code to convert the `raster` objects to `SpatialPixelDataFrame` objects for plotting.

Good to know

This project is aimed at intermediate to advanced students. It lets you apply the skills from courses such as Intermediate Functional Programming with purrr, Spatial Analysis in R with sf and raster, and Hyperparameter Tuning in R. We recommend that you complete these courses before starting this project.

Helpful links:

- tidyverse cheat sheet
- dplyr's `mutate()` function documentation
- mutate exercises in the Introduction to the Tidyverse course

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load in the tidyverse, raster, and sf packages
library(tidyverse)
library(raster)
library(sf)

# Read the climate data from an rds file
climate <- read_rds("datasets/climate_raster.rds")

# Have a look at the variables in the climate data
colnames(climate)

# Convert to SpatialPixelDataFrame for plotting
climate_df <- mutate(
```

```
  .data = climate,
  rasters = map(
    .x = rasters,
    ~ as_tibble(as(.x, "SpatialPixelsDataFrame")))) %>%
  unnest(cols = c(rasters))
```

Task 2: Instructions

Create a map to visualize a sample of the data.

- Filter the climate data to compare 1970 and 2010.
- Create a plot using the data from the column `minimum.temperature`.
- Style the plot as a map with `theme_map()` and `coord_equal()`.
- Display the map.

-

Solution

```
library(ggthemes)

# Filter the data to plot
ggp_temperature <- climate_df %>%
  filter(decade %in% c(1970, 2010)) %>%
  # Create the plot
  ggplot(aes(x = x, y = y)) + geom_tile(aes(fill = minimum.temperature)) +
  # Style the plot with options ideal for maps
  theme_map() + coord_equal() +
  facet_grid(~ decade) + scale_fill_distiller(palette = "Spectral") +
  theme(legend.title = element_blank(), legend.position = "bottom") +
  labs(title = "Minimum of Average Monthly Temperature (Celsius)", caption
= 'Source: MetOffice UK')

ggp_temperature
```

Task 3: Instructions

Obtain species records for our analysis from GBIF.

- Use `occ_search()` from the `rgbif` package to call the GBIF API and retrieve species occurrence information.
- Inspect the returned object by showing its `class()` and `names()`.
- From the elements in `gbif_response`, pick the `data` element and print its first six lines using `head()`.

NOTE: We have modified the `occ_search()` function to pull data from the DataCamp servers. When using the function on your local machine, the syntax is the same as it is here.

-

Solution

```
library(rgbif)
source("datasets/occ_search.R")

# Call the API to get the occurrence records of this species
gbif_response <- occ_search(
  scientificName = "Loxia scotica", country = "GB",
  hasCoordinate = TRUE, hasGeospatialIssue = FALSE, limit = 2000)

# Inspect the class and names of gbif_response
class(gbif_response)
names(gbif_response)

# Show a sample of the data, ignoring the metadata
head(gbif_response[["data"]])
```

Task 4: Instructions

Process the species occurrence records.

- Calculate the decade each record belongs to using `eventDate`.
- Filter for records between 1970 and 2010.
-

Solution

```
library(lubridate)

birds_dated <- mutate(
  .data = gbif_response$data,
  # Create a new column specifying the decade of observation
  decade = ymd_hms(eventDate) %>% round_date("10y") %>% year())

birds_cleaned <- birds_dated %>%
  filter(
    issues == "" &
    str_detect(license, "http://creativecommons.org/") &
    # No records before 1970s decade or after 2010s decade
    decade >= 1970 & decade <= 2010
  ) %>%
  transmute(decade = decade, x = decimalLongitude, y = decimalLatitude) %>%
  arrange(decade)
```

Task 5: Instructions

Created a nested data frame and count the birds.

- Create `birds_nested` by first grouping `birds_cleaned` by `decade`, then use the `nest()` function. Set the `.key` argument to `"presences"` to name the nested column.

- Calculate the total number of records per decade by mapping the `nrow()` function to the nested column. Pay attention to the parentheses!

Functions from the `purrr` package work exceptionally well with list columns.

Helpful links:

- `tidyr`'s nest function
- `purrr`'s map functions

-

Solution

```
# "Nest" the bird data
birds_nested <- birds_cleaned %>%
  group_by(decade) %>%
  nest(.key = "presences")

head(birds_nested)

# Calculate the total number of records per decade
birds_counted <- birds_nested %>%
  mutate(n = map_dbl(.x = presences, .f = nrow))

head(birds_counted)
```

Task 6: Instructions

Turn observations into locations.

- Define the two required CRS strings using `st_crs()`.
- Specify the projection of the occurrence records (latitude longitude).
- Transform the projection to match that of the climate data (UK grid).

We can find the CRS string for any specific projection at spatialreference.org.

Helpful links:

- `st_crs()` documentation

-

Solution

```
# Define geographical projections
proj_latlon <- st_crs("+proj=longlat +datum=WGS84 +ellps=WGS84
+towgs84=0,0,0")
```

```
proj_ukgrid <- st_crs("+init=epsg:27700")

# Convert records to spatial points and project them
birds_presences <- mutate(birds_counted,
  presences = map(presences, ~ .x %>%
    # Specify the current projection
    st_as_sf(coords = c("x", "y"), crs = proj_latlon) %>%
    # Transform to new projection
    st_transform(crs = proj_ukgrid)))
```

Task 7: Instructions

Get the climatic conditions at the observed locations.

- Combine `birds_presences` and `climate` by joining them on `decade`.
- Use `raster::extract()` to obtain the values of the climatic variables. Remember that the function used inside `map2_df()` takes two arguments, `x` and `y`!

Helpful links:

- `raster::extract()`
-

Solution

```
# Combine the bird data and the climate data in one data frame
birds_climate <- full_join(birds_presences, climate, by = "decade")

presence_data <- map2_df(
  .x = birds_climate[["rasters"]],
  .y = birds_climate[["presences"]],
  # extract the raster values at presence locations
  ~ raster::extract(x = .x, y = .y) %>%
    as_tibble() %>%
    mutate(observation = "presence"))
```

Task 8: Instructions

Create a set of pseudo-absences.

- Look at the helper function for sampling the rasters. Don't replace the `...` here. They are part of a concept called ellipsis.
- Using the column names as input variables, use `pmap_df` to apply the helper function to the climate data for each decade. The result will be a data frame for each decade, nested in a list column.
- Combine the `presence_data` and `pseudo_absence_data`.
-

Solution

```
# Define helper function for creating pseudo-absence data
create_pseudo_absences <- function(rasters, n, ...) {
    set.seed(12345)
    sampleRandom(rasters, size = n * 5, sp = TRUE) %>%
    raster::extract(rasters, .) %>% as_tibble() %>%
    mutate(observation = "pseudo_absence")
}

# Create pseudo-absence proportional to the total number of records per
decade
pseudo_absence_data <- pmap_df(.l = birds_climate, .f =
create_pseudo_absences)

# Combine the two datasets
model_data <- bind_rows(presence_data, pseudo_absence_data) %>%
  mutate(observation = factor(observation)) %>% na.omit()
```

Task 9: Instructions

Train a model to predict the probability of occurrence.

- Load `caret` and set the reproducible seed to `12345`.
- Create `tuneGrid` to optimise the hyperparameters `alpha` and `lambda`.
- Use `trainControl()` to create settings for model training.
- Fit a statistical model to the data with `train()` and plot the model fit.

`glmnet` models have two hyperparameters, `alpha` and `lambda`. If you would like to learn what they do, have a look at the vignette.

-

Solution

```
# Load caret and set a reproducible seed
library(caret)
set.seed(12345)

# Create a tuning grid with sets of hyperparameters to try
tuneGrid <- expand.grid(alpha = c(0, 0.5, 1), lambda = c(.003, .01, .03,
.06))

# Create settings for model training
trControl <- trainControl(method = 'repeatedcv', number = 5, repeats = 1,
  classProbs = TRUE, verboseIter = FALSE, summaryFunction =
twoClassSummary)

# Fit a statistical model to the data and plot
model_fit <- train(
  observation ~ ., data = model_data,
  method = "glmnet", family = "binomial", metric = "ROC",
  tuneGrid = tuneGrid, trControl = trControl)
```

```
plot(model_fit)
```

Task 10: Instructions

Use the fitted model object to make predictions for all grid cells.

- Predict the entire `climate_df` dataset using the `predict()` function and the model object from the previous task.
-

Solution

```
# Use our model to make a prediction
climate_df$prediction <- predict(
    object = model_fit,
    newdata = climate_df,
    type = "prob")[["presence"]]

head(climate_df)
```

Task 11: Instructions

Create a faceted visualization of your predictions.

- Create a `ggplot` from `climate_df` which visualizes our predictions on a map and assign it to `ggp_changemap`.
- Style the plot as a map just as before in Task 2.
- Add `facet_grid` by `decade` to create separate sub-plots spread across five columns.
-

Solution

```
# Load packages gganimate and viridis
library(viridis)

# Create the plot
ggp_changemap <- ggplot(data = climate_df, aes(x = x, y = y, fill =
prediction)) +
  geom_tile() +
  # Style the plot with the appropriate settings for a map
  theme_map() + coord_equal() +
  scale_fill_viridis(option = "A") + theme(legend.position = "bottom") +
  # Add faceting by decade
  facet_grid(~ decade) +
  labs(title = 'Habitat Suitability', subtitle = 'by decade',
       caption = 'Source:\nGBIF data and\nMetOffice UK climate data',
       fill = 'Habitat Suitability [0 low - high 1]')

# Display the plot
```

ggp_changemap

Are You Ready for the Zombie Apocalypse?

Task 1: Instructions

Load the zombie data and create a gallons-of-water-per-person variable.

- Use `read.csv()` to bring in the data from `datasets/zombies.csv` and name it `zombies`.
- Summarize `zombies` with the `summary()` command.
- Create a `water.person` variable by dividing `water` by `household`.
- Use `summary()` to check the new variable.

Good to know

This Project uses Base R functions and skills from Multiple and Logistic Regression, including estimating a logistic model, computing and interpreting odds ratios, predicting probabilities, and examining model fit.

Helpful links:

- `ggplot2` documentation
- the `apply()` family functions DataCamp tutorial
- more on the `apply()` functions at R-bloggers

You can reset the Project by clicking the circular arrow in the bottom-right corner of the screen if you are experiencing odd behavior. Resetting the Project will also discard all the code you have written so be sure to save it offline first.

-

Solution

```
# Read in the data
zombies <- read.csv("datasets/zombies.csv")

# Examine the data with summary()
summary(zombies)

# Create water-per-person
zombies$water.person <- zombies$water / zombies$household

# Examine the new variable
summary(zombies$water.person)
```

Task 2: Instructions

Examine age and water per person by zombie status.

- Load `ggplot2` and `gridExtra`.
- Plot `age` by `zombie` using `ggplot()` with `geom_density()`. Set alpha to 0.3 for transparency.
- Plot `water.person` by `zombie` using `ggplot()` with `geom_density()`. Set `alpha` to 0.3 for transparency and add `theme_minimal()`.
- View both graphs with `grid.arrange()` from the `gridExtra` package.

Helpful links:

- ggplot's themes documentation
- ggplot's density plot documentation
- gridExtra documentation

-

Solution

```
# Load ggplot2 and gridExtra
library(ggplot2); library(gridExtra)

# Create the ageZombies graph
ageZombies <- ggplot(data = zombies, aes(x = age, fill = zombie)) +
  geom_density(alpha = 0.3) +
  theme_minimal() +
  theme(legend.position = "bottom", legend.title = element_blank())

# Create the waterPersonZom graph
waterPersonZom <- ggplot(data = zombies, aes(x = water.person, fill =
zombie)) +
  geom_density(alpha = 0.3) +
  theme_minimal() +
  theme(legend.position = "bottom", legend.title = element_blank())

# Display plots side by side
grid.arrange(ageZombies, waterPersonZom, ncol = 2)
```

Task 3: Instructions

Determine the percentage of zombies for each category of the factor variables.

- Use `sapply()` with `is.factor()` to create a data frame with only the factors.
- Write a function to compute percent zombies and humans for each factor. Use `lapply()` to apply the function to `zombies.factors`.
- Print the results.

In `prop.table()` use `margin = 1` for row percents and `margin = 2` for column percents.

Helpful links:

- `lapply()` function documentation
- `sapply()` function documentation
- DataCamp course on Introduction to Function Writing in R

-

Solution

```
# Make a subset of data with only factors
zombies.factors <- zombies[ , sapply(zombies, is.factor)]

# Write a function to get percent zombies
perc.zombies <- lapply(zombies.factors,
                       function(x){
                           return(prop.table(table(x,
zombies.factors$zombie),
                                                margin = 1))
                           })
# Print the data
perc.zombies
```

Task 4: Instructions

Recode missing values to appropriate categories for the clothing and document variables.

- Add "No clothing" as a level of `clothing` and recode `NA` to "No clothing".
- Add "No documents" as a level of `documents` and recode `NA` to "No documents".
- Check recoding using `summary()` on `zombies`.

Helpful links:

- The levels attribute of factor variables can be confusing, see the documentation for more information

-

Solution

```
# Add new level and recode NA to "No clothing"
levels(zombies$clothing) <- c(levels(zombies$clothing), "No clothing")
zombies$clothing[is.na(zombies$clothing)] <- "No clothing"

# Add new level and recode NA to "No documents"
levels(zombies$documents) <- c(levels(zombies$documents), "No documents")
zombies$documents[is.na(zombies$documents)] <- "No documents"

# Check recoding
summary(zombies)
```

Task 5: Instructions

Determine which characteristics are associated with zombie status.

- Use `sapply()` and `is.factor()` to take a subset of factors in `zombies`.
- Use `lapply()` to conduct chi-squared tests on variables in the `zombies.factors` subset with `zombie`.
- Use `t.test()` to see if `age` and `water.person` are associated with `zombie` in the `zombies` data frame.
- Examine chi-squared and t-test results.

Helpful links:

- learn more about `chisq.test()`here
- `t.test()` compares means across groups, see documentation
- `lapply()` function documentation
- `sapply()` function documentation
- DataCamp course on Introduction to Function Writing in R

-

Solution

```
# Update subset of factors
zombies.factors <- zombies[ , sapply(zombies, is.factor)]

# Chi-squared for factors
chi.zombies <- lapply(zombies.factors,
                      function(x){
                          return(chisq.test(x, zombies.factors$zombie))
                          })

# T-tests for numeric
ttest.age <- t.test(zombies$age ~ zombies$zombie)
ttest.water <- t.test(zombies$water.person ~ zombies$zombie)

# Examine the results
chi.zombies; ttest.age; ttest.water
```

Task 6: Instructions

Develop and evaluate a logistic regression model.

- Use the `glm()` command to model `zombie`.
- Run `odds.n.ends()` to get model significance, fit, and odds ratios.
- Print the results of `odds.n.ends()`.

Helpful links:

- `odds.n.ends()` package documentation
- See the logistic regression chapter in the Multiple and Logistic Regression DataCamp course for more information

-

Solution

```
# Create zombie model
zombie.model <- glm(zombie ~ age + water.person + food +
                    rurality + medication + sanitation,
                   data = zombies, family = binomial(logit))

# Model significance, fit, and odds ratios with 95% CI
library(odds.n.ends)
zombie.model.fit <- odds.n.ends(zombie.model)

# Print the results of the odds.n.ends command
zombie.model.fit
```

Task 7: Instructions

Check the no multicollinearity and linearity assumptions for the zombie model.

- Load `car` and use `vif()` to check multicollinearity for `zombie.model`.
- Calculate the logit of `zombie` to use for checking linearity.
- Create scatter plots to check linearity for `age` and `water.person`. The arguments in `geom_smooth()` are the same for both plots.
- Use `grid.arrange()` to view and examine both plots.

For linearity assumption checking, scatter plots will show two lines, a straight regression line and a curvy Loess curve, which captures more subtle localized detail. If the Loess curve is close to being straight, the relationship between x and y is linear. If it deviates a lot from being straight, the relationship is not linear.

Helpful links:

- more about variance inflation factors from the `vif()` command
- learn about Loess smoothing here and its use with `ggplot()` here

-

Solution

```
# Compute GVIF
library(car)
vif(zombie.model)
```

```
# Make a variable of the logit of the outcome
zombies$logitZombie <- log(zombie.model$fitted.values/(1-
zombie.model$fitted.values))

# Graph the logit variable against age and water.person
ageLinearity <- ggplot(data = zombies, aes(x = age, y = logitZombie))+
  geom_point(color = "gray") +
  geom_smooth(method = "loess", se = FALSE, color = "orange") +
  geom_smooth(method = "lm", se = FALSE, color = "gray") +
  theme_bw()

waterPersonLin <- ggplot(data = zombies, aes(x = water.person, y =
logitZombie))+
  geom_point(color = "gray") +
  geom_smooth(method = "loess", se = FALSE, color = "orange") +
  geom_smooth(method = "lm", se = FALSE, color = "gray") +
  theme_bw()

grid.arrange(ageLinearity, waterPersonLin, ncol = 2)
```

Task 8: Instructions

Use the model to predict the probability that two people are zombies.

- Create a new data frame with `data.frame()` and enter the relevant data for the two people.
- Use `predict()` on `zombie.model` to predict probabilities for the people in the new data frame.
- Print the predicted probabilities.

Helpful links:

- the base R `predict()` command [documentation]

-

Solution

```
# Make a new data frame with the relatives data in it
newdata <- data.frame(age = c(71, 40),
                      water.person = c(5, 3),
                      food = c("Food", "Food"),
                      rurality = c("Suburban", "Urban"),
                      medication = c("Medication", "Medication"),
                      sanitation = c("Sanitation", "Sanitation"))

# Use the new data frame to predict
predictions <- predict(zombie.model, newdata, type = "response")

# Print the predicted probabilities
predictions
```

Task 9: Instructions

Add your data to the `newdata` data frame and predict your zombie probability.

- Add your data to the `newdata` data frame.
- Use `predict()` to predict the probabilities for the `newdata` data frame.
- Print the predictions to review your zombie probability!

-

Solution

```
# Add your data to the newdata data frame
newdata <- data.frame(age = c(71, 40, 48),
                      water.person = c(5, 3, 0),
                      food = c("Food", "Food", "Food"),
                      rurality = c("Suburban", "Urban", "Suburban"),
                      medication = c("Medication", "Medication",
"Medication"),
                      sanitation = c("Sanitation", "Sanitation",
"Sanitation"))

# Use the new data frame to predict
predictions <- predict(zombie.model, newdata, type = "response")

# Print the predictions
predictions
```

Task 10: Instructions

How prepared are you?

- What is your probability of becoming a zombie? Assign your probability of becoming a zombie to `me`.
- How prepared are you for a real emergency? Assign one of the text lines below to `preparedness_level`.
  - "I got this!"
  - "Okay, but I should probably pick up a few emergency items at the store."
  - "I'm not going to fair well. Let's make an emergency kit!"

Helpful links: [CDC preparedness website]

-

Solution

```
# What is your probability of becoming a zombie?
me <- 0.2
```

```
# How prepared are you for a real emergency?
preparedness_level  <- "I got this!"
```

Trends in Maryland Crime Rates

Task 1: Instructions

Import data and start to tidy it.

- Read in the crime data, `"datasets/Violent_Crime_by_County_1975_to_2016.csv"`, using `read_csv()` and save it as `crime_raw`.
- Use `select()` to select the columns (in this order) `JURISDICTION`, `YEAR`, and `POPULATION`, and rename `` `VIOLENT CRIME RATE PER 100,000 PEOPLE` `` to `crime_rate`. Use `mutate()` to create `YEAR_2` from `YEAR` using `year(mdy_hms())`.
- Peek at the data using `head()`.

Good to know

Backticks (` `) are required for column names with spaces or other non-alpha-numeric characters.

This project is similar to a project a consulting statistician or data analyst would do for a client. To complete this project, you will need knowledge of Tidyverse syntax, as well as familiarity with regression analysis.

Prerequisite skills are taught in DataCamp Courses such as Introduction to the Tidyverse, Introduction to Data Visualization with ggplot2, and Hierarchical and Mixed Effects Models and its prerequisite Generalized Linear Models in R.

Helpful links:

- lme4's `lmer()` documentation
- `read_csv()` exercise in the Importing Data in R (Part 1) course
- tidyverse cheat sheet, which includes `select()`, `mutate()`, `group_by()`, and `summarize()`
- lubridate's `ymd()` and `year()` functions, both described on the package's webpage

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load the packages
```

```
library(tidyverse)
library(lubridate)

# Read in the crime data
crime_raw <- read_csv("datasets/Violent_Crime_by_County_1975_to_2016.csv")

# Select and mutate columns the needed columns
crime_use <- crime_raw %>%
    select(JURISDICTION, YEAR, POPULATION, crime_rate = `VIOLENT CRIME RATE
PER 100,000 PEOPLE`) %>%
    mutate(YEAR_2 = year(mdy_hms(YEAR)))

# Peek at the data
head(crime_use)
```

Task 2: Instructions

Visualize the data and trends for each county.

- Use `crime_use` to make a line graph of `crime_rate` vs. `YEAR_2` grouped by `JURISDICTION`. Add linear trend lines with `method = "lm"`, remove the standard error shading, and set the size of the trend line to `0.5`.

Helpful links:

- ggplot2's homepage
- ggplot2's `geom_line()` documentation
- ggplot2's `stat_smooth()` documentation

-

Solution

```
# Plot the data and linear trend lines
ggplot(crime_use, aes(x = YEAR_2, y = crime_rate, group = JURISDICTION)) +
    geom_line() +
    stat_smooth(method = 'lm', se = FALSE, size = 0.5)
```

Task 3: Instructions

Re-scale the data.

- Create a new variable, `YEAR_3`, by subtracting the minimum value of `YEAR_2` from `YEAR_2`.

Helpful links:

- `mutate()` documentation
-

Solution

```
# Mutate data to create another year column, YEAR_3
crime_use <-
  crime_use %>%
  mutate(YEAR_3 = YEAR_2 - min(YEAR_2))
```

Task 4: Instructions

Build a linear mixed-effects regression model to predict the crime rate.

- Load the `lmerTest` package.
- Using `crime_use` data, build a `lmer()` with `crime_rate` predicted by `YEAR_3` as a fixed-effect slope, and `YEAR_3` as a random-effect slope with `JURISDICTION` as a random-effect intercept. Save this as `lmer_crime`.
- Inspect the `lmer_crime` output.

Helfpul links:

- lme4's `lmer()` documentation
- lme4's `lmer()` vignette
-

Solution

```
# load the lmerTest package
library(lmerTest)

# Build a lmer and save it as lmer_crime
lmer_crime <- lmer(crime_rate ~ YEAR_3 + (YEAR_3|JURISDICTION), crime_use)

# Print the model output
lmer_crime
```

Task 5: Instructions

Examine the model outputs.

- Use `summary()` to look at `lmer_crime`.
- Use `fixef()` to extract only the fixed-effects from `lmer_crime`.
- Use `ranef()` to extract only the random-effects from `lmer_crime`.

Helpful links:

- lmerTest's `summary()` documentation
- lmer4's `fixef()` documentation
- lmer4's `ranef()` documentation

-

Solution

```
# Examine the model outputs using summary
summary(lmer_crime)

# This is for readability
noquote("**** Fixed-effects ****")

# Use fixef() to view fixed-effects
fixef(lmer_crime)

# This is for readability
noquote("**** Random-effects ****")

# Use ranef() to view random-effects
ranef(lmer_crime)
```

Task 6: Instructions

Create a data frame containing the slope estimates and county names.

- Use `fixef()` to extract the fixed-effect slope estimate for `YEAR_3`, and use `ranef()` to extract the estimates for each `JURISDICTION`'s `YEAR_3` random-effect slope. Add the two together and save the output as `county_slopes`.
- Convert the row names, which are the county names, to a new column called `county` using `rownames_to_column()`.

The row names of `county_slopes` are the county names for the counties.

Accessing a column `x` of a data frame `dat` using `dat[ "x"]` produces a data frame with row names whereas `dat[, "x"]` produces a vector without names.

Helpful links:

- tibble's `rownames_to_column()` function documentation
- Hadley Wickham's reasons for disliking row names as described in Advanced R.

-

Solution

```
# Add the fixed-effect to the random-effect and save as county_slopes
county_slopes <- fixef(lmer_crime)["YEAR_3"] + ranef(lmer_crime)
$JURISDICTION["YEAR_3"]

# Add a new column with county names
county_slopes <-
    county_slopes %>%
    rownames_to_column("county")
```

Task 7: Instructions

Format the county data for plotting.

- Load the `usmap` package.
- Within the call to `us_map()`, select `counties` as the *region* of interest and *include* Maryland (`"MD"`).

Helpful links:

- usmap's `us_map()` function documentation
- usmap's vignette

-

Solution

```
# Load usmap package
library(usmap)

# load and filter map data
county_map <- us_map(regions = 'counties', include = "MD")
```

Task 8: Instructions

Make sure the county names are the same in both the regression output and coordinate datasets.

- Use the `anti_join()` function to see which names differ between `county_slopes` and `county_map`. Run the function twice, changing the order of the inputs.
- Reformat the character strings in `county_slopes` to match the format of the county names in `map_names` with `ifelse()`.

Helpful links:

- dplyr's `anti_join()` function documentation

-

Solution

```
# See which counties are not in both datasets
county_slopes %>% anti_join(county_map, by = "county")
county_map %>% anti_join(county_slopes, by = "county")

# Rename crime data
county_slopes <-
  county_slopes %>%
  mutate(county = ifelse(county =="Baltimore City", "Baltimore city",
county))
```

Task 9: Instructions

Merge the county crime and map data frames.

- Using `full_join()`, join `county_map` (1st argument) to `county_slopes` (2nd argument) and call the new data frame `both_data`.
- Look at the new data frame using `head()`

Helpful links:

- dplyr's `full_join()` function [documentation]

-

Solution

```
# Merge the map and slope data frames
both_data <-
  county_map %>%
  full_join(county_slopes, by = "county")

# Peek at the data
head(both_data)
```

Task 10: Instructions

Create a map of the regression coefficients by Maryland counties.

- Using `both_data`, plot `long` and `lat`, group by `county`, and fill by `YEAR_3`.
- Include a `geom_polygon()`.
- Change the continuous fill to have a *low* value of `"skyblue"` and a *high* value of `"gold"`. Change the fill's legend `name =` argument to `expression(atop("Change in crime rate","(Number year"^-1*")"))`.
- Save the map as `crime_map` and look at the map.

Helpful links:

- ggplot2's `scale_fill_continuous()` function documentation
- base's `expression()` function documentation
- grDevices's plotmath documentation, which includes documentation on `atop()`. `demo(plotmath)` also shows more `expression()` options.

-

Solution

```
# set the notebook's plot settings to display nicely
options(repr.plot.width=10, repr.plot.height=5)

# Plot the results
crime_map <-
    ggplot(data = both_data, aes(x = long, y = lat,
                                 group = county,
                                fill= YEAR_3)) +
    geom_polygon() +
    scale_fill_continuous(name = expression(atop("Change in crime
rate","(Number year"^-1*")")),
                          low = 'skyblue', high = 'gold')

# Look at the map
crime_map
```

Task 11: Instructions

Polish the figure to make it presentable.

- Create `crime_map_final` by setting the theme to `theme_minimal()`, setting `xlab()` and `ylab()` to be blank (use `""`), and setting the `theme()` options `axis.line`, `axis.text`, `panel.grid.major`, `panel.grid.minor`, `panel.border`, and `panel.background` to `element_blank()`.
- Look at the new map.

Helpful links:

- ggplot2's label documentation
- ggplot2's theme documentation

-

Solution

```
# Plot options
options(repr.plot.width=10, repr.plot.height=5)
```

```
# Polish figure
crime_map_final <- crime_map +
  theme_minimal() +
  ylab("") +
  xlab("") +
  theme(axis.line=element_blank(),axis.text=element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank())

# Look at the map
print(crime_map_final)
```

Task 12: Instructions

Examine if population impacts crime rate.

- Add `POPULATION` as a second-fixed effect in the linear mixed-effects regression model you previously built (Task 4). Save the new model as `lmer_pop`. Notice the warning message. Think about how you might fix this.
- Inspect the results.

Including `POPULATION` changed the slope estimate for crime. Does this change how you think about the data?

Throughout this project, most of the code we wrote was to either wrangle data or format results. We used less than a quarter (3/12 Tasks) to run models. In my experience as a data scientist, this is often the case.

-

Solution

```
# build a lmer with both year and population
lmer_pop <- lmer(crime_rate ~ YEAR_3 + POPULATION + (YEAR_3|JURISDICTION),
crime_use)

summary(lmer_pop)
ranef(lmer_pop)
```

Health Survey Data Analysis of BMI

Task 1: Instructions

Import the NHANES data and view its structure.

- Load the `NHANES` and `dplyr` packages.
- Load the `NHANESraw` dataset from the `NHANES` package with the `data()` function.
- Examine the structure of the `NHANESraw` with `glimpse()`.

Good to know

This project lets you apply the skills from Analyzing Survey Data in R, including calculating survey-weighted means and proportions, visualizing data, and multiple linear regression. You can learn more about statistical inference with regression in the course Multiple and Logistic Regression. This project also lets you apply many skills from Introduction to the Tidyverse, including summarizing data and visualizing with ggplot2. We recommend that you take these courses before starting this project.

Helpful links:

- tidyverse cheat sheet
- `data()` function documentation
- `NHANES` R package documentation including the description of the NHANES data
- vignette from `survey` package
- Exploratory Data Analysis in R: Case Study, Chapter 3 has examples of using `broom` to tidy linear regression output

You can reset the project by clicking the circular arrow in the bottom-right corner of the screen if you are experiencing odd behavior. Resetting the project will also discard all the code you have written so be sure to save it offline first.

-

Solution

```
# Load the NHANES and dplyr packages
library(NHANES)
library(dplyr)

# Load the NHANESraw data
data("NHANESraw")

# Take a glimpse at the contents
glimpse(NHANESraw)
```

Task 2: Instructions

Construct the appropriate sample weight and view its distribution by race category.

- Load the `ggplot2` package.
- Use `mutate()` to add a column to the `NHANESraw` data called `WTMEC4YR` that is equal to `WTMEC2YR` divided by 2.
- Calculate the sum of the new weight variable `WTMEC4YR`.
- Using `NHANESraw`, make a box and whisker plot of `WTMEC4YR` grouped by `Race1`.

Helpful links:

- dplyr's `mutate()` function documentation
- Mutate exercises in the Introduction to the Tidyverse course
- ggplot2's `geom_boxplot()` function documentation

-

Solution

```
# Load the ggplot2 package
library(ggplot2)

# Use mutate to create a 4-year weight variable and call it WTMEC4YR
NHANESraw <- NHANESraw %>% mutate(WTMEC4YR = WTMEC2YR/2)

# Calculate the sum of this weight variable
NHANESraw %>% summarize(sum(WTMEC4YR))

# Plot the sample weights using boxplots, with Race1 on the x-axis
ggplot(NHANESraw, aes(x = Race1, y = WTMEC4YR)) + geom_boxplot()
```

Task 3: Instructions

Specify the sampling design of the survey data by specifying the strata, clustering ids, and weight variables.

- Load the `survey` package.
- Specify the survey sampling design as an object named `nhanes_design`. Use the weight variable we created `WTMEC4YR`.
- Print a summary of the survey design object `nhanes_design`.

Helpful links:

- For more detailed information on the NHANES sampling design, see the CDC website.

- survey's `svydesign()` function documentation
- Analyzing Survey Data in R, Chapter 1, Exercise 4 describes how to specify the elements of the survey structure with the NHANES specific design shown in Exercise 11.

-

Solution

```
# Load the survey package
library(survey)

# Specify the survey design
nhanes_design <- svydesign(
    data = NHANESraw,
    strata = ~SDMVSTRA,
    id = ~SDMVPSU,
    nest = TRUE,
    weights = ~WTMEC4YR)

# Print a summary of this design
summary(nhanes_design)
```

Task 4: Instructions

Create a survey design object containing only adults who are at least 20 years old.

- Create a subset of `nhanes_design` filtering by `Age` and save it as `nhanes_adult`.
- Print a summary of `nhanes_adult`.
- Compare the number of observations of the full data and the adult data by printing the number of rows with `nrow`.

Helpful links:

- survey's `subset()` function documentation

-

Solution

```
# Select adults of Age >= 20 with subset
nhanes_adult <- subset(nhanes_design, Age >= 20)

# Print a summary of this subset
summary(nhanes_adult)

# Compare the number of observations in the full data to the adult data
nrow(nhanes_design)
nrow(nhanes_adult)
```

Task 5: Instructions

Compare mean BMI in `NHANESraw` with the estimate of mean BMI in the US and visualize BMI's distribution.

- Calculate mean BMI in adults of at least 20 years of age in the `NHANESraw` data.
- Use survey methods to estimate the mean BMI in the US population.
- Draw a histogram of `BMI` with `WTMEC4YR` mapped to `weight` and add a vertical line showing the estimated mean BMI.

Helpful links:

- survey's `svymean()` function documentation
- ggplot2's `geom_histogram()` function documentation
- Analyzing Survey Data with R, Chapter 2, Excercise 8 shows how to draw a weighted histogram.

-

Solution

```
# Calculate the mean BMI in NHANESraw
bmi_mean_raw <- NHANESraw %>%
    filter(Age >= 20) %>%
    summarize(mean(BMI, na.rm=TRUE))
bmi_mean_raw

# Calculate the survey-weighted mean BMI of US adults
bmi_mean <- svymean(~BMI, design = nhanes_adult, na.rm = TRUE)
bmi_mean

# Draw a weighted histogram of BMI in the US population
NHANESraw %>%
  filter(Age >= 20) %>%
    ggplot(mapping = aes(x = BMI, weight = WTMEC4YR)) +
    geom_histogram()+
    geom_vline(xintercept = coef(bmi_mean), color="red")
```

Task 6: Instructions

Compare mean BMI between physically active and non-physically active people.

- Load the `broom` package.
- Create a box and whisker plot of `BMI` with `PhysActive` mapped to the x-axis and weighted by `WTMEC4YR`.
- Compare the mean BMI between the two physical activity groups with a survey-weighted t-test.
- Use `tidy()` in the `broom` package to print a summary of the t-test.

Helpful links:

- ggplot2's `geom_boxplot()` documentation
- survey's `svyttest()` documentation
- broom's `tidy()` documentation
- Analyzing Survey Data in R, Chapter 3, Exercise 11 shows an example of using a survey-weighted t-test.

-

Solution

```
# Load the broom library
library(broom)

# Make a boxplot of BMI stratified by physically active status
NHANESraw %>%
  filter(Age>=20) %>%
    ggplot(mapping = aes(x = PhysActive, y = BMI, weight = WTMEC4YR)) +
    geom_boxplot()

# Conduct a t-test comparing mean BMI between physically active status
survey_ttest <- svyttest(BMI~PhysActive, design = nhanes_adult)

# Use broom to show the tidy results
tidy(survey_ttest)
```

Task 7: Instructions

Estimate and visualize the proportion of physically active people stratified by current smoking status.

- Calculate the survey-weighted proportion (mean of binary variable) of people in each `PhysActive` category ("Yes" and "No") stratified by smoking status `SmokeNow` and save the output as `phys_by_smoke`.
- Create a bar plot with `geom_col()` showing the proportion of physically active people with `SmokeNow` categories on the x-axis.
- Label the y-axis with "Proportion Physically Active".

Helpful links:

- survey's `svyby()` documentation
- ggplot2's `geom_col()` documentation
- Analyzing Survey Data in R, Chapter 3, Exercise 5 shows an example of bar plots of survey-weighted means.

-

Solution

```
# Estimate the proportion who are physically active by current smoking
status
phys_by_smoke <- svyby(~PhysActive, by = ~SmokeNow,
                       FUN = svymean,
                       design = nhanes_adult,
                       keep.names = FALSE)

# Print the table
phys_by_smoke

# Plot the proportions with y-label
ggplot(data = phys_by_smoke,
       aes(y = PhysActiveYes, x = SmokeNow, fill = SmokeNow)) +
    geom_col() +
    ylab("Proportion Physically Active")
```

Task 8: Instructions

Calculate mean BMI and visualize the distribution of BMI by current smoking status.

- Use survey-weighted methods to estimate mean `BMI` stratified by `SmokeNow`.
- Make a survey-weighted box and whisker plot of `BMI` with `SmokeNow` on the x-axis.

Helpful links:

- survey's `svyby()` documentation
- ggplot2's `geom_boxplot()` documentation
- Analyzing Survey Data in R, Chapter 3, Exercise 5 shows an example of using `svyby()` to calculate the mean of a quantitative variable stratified by a grouping variable.

-

Solution

```
# Estimate mean BMI by current smoking status
BMI_by_smoke <- svyby(~BMI, by = ~SmokeNow,
      FUN = svymean,
      design = nhanes_adult,
      na.rm = TRUE)
BMI_by_smoke

# Plot the distribution of BMI by current smoking status
NHANESraw %>%
  filter(Age>=20, !is.na(SmokeNow)) %>%
    ggplot(mapping = aes(x = SmokeNow, y = BMI, weight = WTMEC4YR)) +
    geom_boxplot()
```

Task 9: Instructions

Plot the distribution of BMI by current smoking and physical activity status.

- Create a survey-weighted box and whisker plot of `BMI` with `SmokeNow` mapped to the x-axis and `PhysActive` mapped to `color`.

Helpful links:

- ggplot2's `geom_boxplot()` function documentation
-

Solution

```
# Plot the distribution of BMI by smoking and physical activity status
NHANESraw %>%
  filter(Age>=20) %>%
    ggplot(mapping = aes(x = SmokeNow,
                         y = BMI,
                         weight = WTMEC4YR,
                         color = PhysActive)) +
    geom_boxplot()
```

Task 10: Instructions

Fit a survey-weighted multiple regression model of BMI on smoking and physical activity.

- Fit a multiple regression model of BMI that includes an interaction term of `SmokeNow` and `PhysActive`.
- Create a tidy data frame of the regression results.
- Calculate the expected mean decrease of BMI associated with physical activity within current non-smokers.
- Calculate the expected mean decrease of BMI associated with physical activity within current smokers.

Helpful links:

- survey's `svyglm()` documentation
- Analyzing Survey Data in R, Chapter 4, Exercise 12 shows how to use `svyglm()` to build multiple regression models.
- The following courses show how to build and interpret regressions with interaction terms:
  - Multiple and Logistic Regression, Chapter 2, Exercise 4
  - Supervised Learning in R: Regression, Chapter 3, Exercise 5
  - Causal Inference for with R - Regression, Chapter 2, Exercise 16

-

Solution

```
# Fit a multiple regression model
mod1 <- svyglm(BMI ~ PhysActive*SmokeNow, design = nhanes_adult)

# Tidy the model results
tidy_mod1 <- tidy(mod1)
tidy_mod1

# Calculate expected mean difference in BMI for activity within non-smokers
diff_non_smoke <- tidy_mod1 %>%
    filter(term=="PhysActiveYes") %>%
    select(estimate)
diff_non_smoke

# Calculate expected mean difference in BMI for activity within smokers
diff_smoke <- tidy_mod1 %>%
    filter(term%in%c("PhysActiveYes","PhysActiveYes:SmokeNowYes")) %>%
    summarize(estimate = sum(estimate))
diff_smoke
```

Task 11: Instructions

Add race, alcohol use, and gender to the multiple regression model.

- Build on `mod1` by adding `Race1`, `Alcohol12PlusYr`, and `Gender` as predictors, and save the model object as `mod2`.
- Print a tidy version of the regression output.

Helpful links:

- survey's `svyglm()` documentation
- Analyzing Survey Data in R, Chapter 4, Exercise 12 shows how to use `svyglm()` to build multiple regression models with and without interactions.

-

Solution

```
# Adjust mod1 for other possible confounders
mod2 <- svyglm(BMI ~ PhysActive*SmokeNow + Race1 + Alcohol12PlusYr +
Gender,
               design = nhanes_adult)

# Tidy the output
tidy(mod2)
```

Task 1: Instructions

Load the packages and data.

- Load the `tidyverse` package.
- Load the CSV files `datasets/super_bowls.csv`, `datasets/tv.csv`, and `datasets/halftime_musicians.csv` into `super_bowls`, `tv`, and `halftime_musicians` respectively.
- Display the first six rows of each tibble using `head()`.

Good to know

This project gives you an opportunity to apply the skills from Introduction to the Tidyverse and Introduction to Data Visualization with ggplot2. DataCamp projects are completed in Jupyter Notebooks. If you would like more information on Jupyter Notebooks, check out this introduction (although it isn't necessary to complete this Project).

Helpful links for this task:

- Home of the Tidyverse
- `read_csv()` documentation

Projects are more open-ended than courses — you are welcome to explore the data on your own! Use the **"Check Project"** button to see if your code is correct. You can check your project even if you have not completed all the tasks. The Jupyter Notebook will provide error messages if your code causes an error. Consult the hints at the end of the instructions to see potential solutions.

The **hints** for this project consist of the solution code filled in with dummy variables.

If you experience odd behavior within the project, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load packages
library(tidyverse)

# Load the CSV data
super_bowls  <-  read_csv("datasets/super_bowls.csv")
tv  <-  read_csv("datasets/tv.csv")
halftime_musicians  <-  read_csv("datasets/halftime_musicians.csv")

# Display the first six rows of each tibble
```

```
head(super_bowls)
head(tv)
head(halftime_musicians)
```

Task 2: Instructions

Look at a summary of the datasets.

- Use the `summary()` to inspect `tv`.
- Use the `summary()` to inspect `halftime_musicians`.

Helpful links:

- `summary()` documentation
-

Solution

```
# Summary of the TV data
summary(tv)

# Summary of the halftime musician data
summary(halftime_musicians)
```

Task 3: Instructions

Plot a histogram of combined points and display the Super Bowls with the highest and lowest scores.

- Set the plot size options (this is done for you).
- Using `super_bowls` and `ggplot()`, create a histogram of the `combined_pts` and set the binwidth to `5`.
- Display the games with combined points greater than **70** OR combined points less than **25**.

Instead of finding the just the minimum combined score and the maximum combined sore, we're looking for games with combined scores above and below a threshold.

Remember, the hints have dummy variables in them.

Helpful hints:

- logical operators
-

Solution

```
# Reduce the size of the plots
options(repr.plot.width = 5, repr.plot.height = 4)

# Plot a histogram of combined points
ggplot(super_bowls, aes(combined_pts)) +
 geom_histogram(binwidth = 5) +
 labs(x = "Combined Points", y = "Number of Super Bowls")

# Display the highest- and lowest-scoring Super Bowls
super_bowls  %>%
 filter(combined_pts > 70 | combined_pts < 25)
```

Task 4: Instructions

Plot a histogram of difference in points between teams and display the closest game and the largest blow out.

- Set the plot size options (this is done for you).
- Using `super_bowls` and `ggplot()`, create a histogram of the `difference_pts` and set the binwidth to `2`.
- Display the games with minimum and maximum point differences.

Helpful links:

- ggplot's [histogram geom]
-

Solution

```
# Reduce the size of the plots
options(repr.plot.width = 5, repr.plot.height = 4)

# Plot a histogram of point differences
ggplot(super_bowls, aes(difference_pts)) +
 geom_histogram(binwidth = 2) +
 labs(x = "Point Difference", y = "Number of Super Bowls")

# Display the closest game and largest blow out
super_bowls  %>%
 filter(difference_pts == min(difference_pts) | difference_pts ==
max(difference_pts))
```

Task 5: Instructions

Filter and join datasets to plot household shares vs. point difference.

- Filter `tv` dataset to remove Super Bowl I, and join the data to `super_bowls` by `super_bowl`.
- Use the new data, `games_tv`, to create a scatter plot of `difference_pts` on the x-axis and `share_household` on the y-axis. Use `geom_smooth()` with `method = "lm"` to add a linear regression line.

We are removing Super Bowl I because it was broadcast on two networks.

Remember, `!=` is spoken, "not equal to".

Helpful links:

- ggplot's `geom_smooth()` documentation
-

Solution

```
# Filter out Super Bowl I and join the game data and TV data
games_tv <- tv  %>%
 filter(super_bowl != 1)  %>%
 inner_join(super_bowls, by = "super_bowl")

# Create a scatter plot with a linear regression model
ggplot(games_tv, aes(difference_pts, share_household)) +
 geom_point() +
 geom_smooth(method = "lm") +
 labs(x = "Point Difference", y = "Viewership (household share)")
```

Task 6: Instructions

Reformat the data and make line graphs of viewership, ratings, and advertisement costs over time.

- Use `gather()` to convert the data format for plotting. The three columns that will become `value` are: `avg_us_viewers`, `rating_household`, and `ad_cost`.
- Use `mutate()` to create `cat_name`.
- Use the new data, `games_tv_plot` to create a line graph of `super_bowl` on the x-axis and `value` on the y-axis. Facet the data by `cat_name` to produce line graphs for each category.
    - Use `theme_minimal()` to reduce clutter in the graph.

`gather()` and `case_when()` are great functions not covered in the beginner courses, but you'll use them often in your data science endeavors. We're using `case_when()` here to recode the values in `category`, so they look nicer in the plot.

Helpful links:

- `gather()` documentation
- `case_when()` documentation

-

Solution

```
# Convert the data format for plotting
games_tv_plot  <- games_tv %>%
    gather(key = "category", value = "value", avg_us_viewers,
rating_household, ad_cost)  %>%
    mutate(cat_name = case_when(category == "avg_us_viewers" ~ "Average
number of US viewers",
                                category == "rating_household" ~ "Household
rating",
                                category == "ad_cost" ~ "Advertisement cost
(USD)",
                                TRUE ~ as.character(category)))

# Plot the data
ggplot(games_tv_plot, aes(super_bowl, value)) +
 geom_line() +
 facet_wrap(~ cat_name, scales = "free", nrow = 3) +
 labs(x = "Super Bowl", y = "") +
 theme_minimal()
```

Task 7: Instructions

Filter and display the Super Bowl musicians before and including Michael Jackson.

- Filter `halftime_musicians` for all super bowls before and including Super Bowl XXVII (27).

Surrounding objects or lines of code with `( )` in R is a short cut to printing the output.

-

Solution

```
# Filter and diplay halftime musicians before and including Super Bowl
XXVII
(pre_MJ  <- halftime_musicians  %>%
 filter(super_bowl <= 27) )
```

Task 8: Instructions

Display the musicians who performed more than once at the Super Bowl.

- Use `count()` with the parameter `sort = TRUE` to find the number of times a `musician` performed a halftime show, then filter for all counts greater than one.

Adding `sort = TRUE` to `count()` will arrange the counts in descending order.

Helpful links:

- `count()` documentation
-

Solution

```
# Display the musicians who performed more than once
halftime_musicians  %>%
    count(musician, sort = TRUE)  %>%
    filter(n > 1)
```

Task 9: Instructions

Create a histogram of the number of songs per performance and list the musicians with more than four songs in a halftime show.

- Create `musicians_songs` using `filter()` with `str_detect()` to remove musicians with "Marching" and "Spirit" in `musician`, then keep only data after Super Bowl XX (20).
- Create a histogram of the `num_songs` per halftime performance and set the `binwidth =` to one.
- Filter the data for `num_songs` greater than or equal to four, and arrange `num_songs` in descending order.

`filter(!str_detect(variable, "pattern"))` is a great way to to remove rows that match the given string pattern.

Remember that each row (observation) in this dataset corresponds to a musician/Super Bowl combination.

Helpful links:

- stringr
-

Solution

```
# Remove marching bands and data before Super Bowl XX
musicians_songs  <- halftime_musicians  %>%
    filter(!str_detect(musician, "Marching"),
           !str_detect(musician, "Spirit"),
          super_bowl > 20)

# Plot a histogram of the number of songs per performance
ggplot(musicians_songs, aes(num_songs)) +
    geom_histogram(binwidth = 1) +
    labs(x = "Number of songs per halftime show", y = "Number of
musicians")

# Display the musicians with more than four songs per show
musicians_songs  %>%
    filter(num_songs > 4)  %>%
    arrange(desc(num_songs))
```

Task 10: Instructions

The New England Patriots and Los Angeles Rams are playing in Super Bowl LIII. Who do you think will win?

- Assign either `patriots` or `rams` to `super_bowl_LIII_winner`.

Congratulations on reaching the end of the Project!

-

Solution

```
# 2018-2019 conference champions
patriots  <-  "New England Patriots"
rams  <-  "Los Angeles Rams"

# Who will win Super Bowl LIII?
super_bowl_LIII_winner  <-  "Elvis"
paste('The winner of Super Bowl LIII will be the', super_bowl_LIII_winner)
```

Comparing Cosmetics by Ingredients

Task 1: Instructions

Import and inspect the dataset.

- Import `pandas` aliased as `pd` and `numpy` as `np`. Import `TSNE` from `sklearn.manifold`.
- Read the CSV file, `"datasets/cosmetics.csv"`, into a pandas DataFrame and name it `df`.
- Display a sample of five rows of the data using the `sample()` method inside the `display()` function.
- Display counts of types of product using the `value_counts()` method on the `Label` column of `df`.

Good to know

This project lets you apply the skills from Manipulating DataFrames with pandas, Chapter 1 of Dimensionality Reduction in Python, and Interactive Data Visualization with Bokeh. This project also includes the concepts of natural language processing and word embedding, which you can learn about in Natural Language Processing Fundamentals in Python. For a deeper dive into word embedding, you can read this article.

Helpful links:

- `read_csv()` function documentation
- How `output_notebook()` is used to display Bokeh plots inline in Jupyter notebooks

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Import libraries
import pandas as pd
import numpy as np
from sklearn.manifold import TSNE

# Load the data
df = pd.read_csv('datasets/cosmetics.csv')

# Display a sample of five rows
display(df.sample(5))
```

```
# Counts of product types
df.Label.value_counts()
```

Task 2: Instructions

Filter the data for moisturizers and dry skin.

- Filter `df` for "Moisturizer" in the `Label` column and store the result in `moisturizers`.
- Filter `moisturizers` for 1 in the `Dry` column and store the result in `moisturizers_dry`.
- Drop the current index of `moisturizers_dry` and replace it with a new one using the `reset_index()` method, setting `drop = True`.

Helpful links:

- `reset_index()` documentation
-

Solution

```
# Filter for moisturizers
moisturizers = df[df['Label'] == 'Moisturizer']

# Filter for dry skin as well
moisturizers_dry = moisturizers[moisturizers['Dry'] == 1]

# Reset index
moisturizers_dry = moisturizers_dry.reset_index(drop = True)
```

Task 3: Instructions

Tokenize the ingredients and create a bag of words.

- Inside the outer `for` loop:
  - Make each product's ingredients list lowercase.
  - Split the lowercase text into tokens by specifying `', '` as the separator.
  - Append `tokens` (which itself is a list) to the list `corpus`.
- Inside the inner `for` loop, if the ingredient is not yet in `ingredient_idx` dictionary:
  - Add an entry to `ingredient_idx` with the key being the new `ingredient` and the value being the current `idx` value.
  - Increment `idx` by 1.

-

Solution

```
# Initialize dictionary, list, and initial index
ingredient_idx = {}
corpus = []
idx = 0

# For loop for tokenization
for i in range(len(moisturizers_dry)):
    ingredients = moisturizers_dry['Ingredients'][i]
    ingredients_lower = ingredients.lower()
    tokens = ingredients_lower.split(', ')
    corpus.append(tokens)
    for ingredient in tokens:
        if ingredient not in ingredient_idx:
            ingredient_idx[ingredient] = idx
            idx += 1

# Check the result
print("The index for decyl oleate is", ingredient_idx['decyl oleate'])
```

Task 4: Instructions

Initialize a document-term matrix.

- Get the total number of products in the `moisturizers_dry` DataFrame. Assign it to `M`.
- Get the total number of ingredients in the `ingredient_idx` dictionary. Assign it to `N`.
- Create a matrix of zeros with size `MxN`. Assign it to `A`.

Helpful links:

- Stack Overflow answer for getting the number of elements in a list
- `numpy.zeros()` documentation

-

Solution

```
# Get the number of items and tokens
M = len(moisturizers_dry)
N = len(ingredient_idx)

# Initialize a matrix of zeros
A = np.zeros((M, N))
A.shape
```

Task 5: Instructions

Create a function named `oh_encoder`.

- Initialize a matrix of zeros with width `N` (i.e., the same width as matrix `A`).
- Get the index values for each ingredient from `ingredient_idx`.
- Put 1 at the corresponding indices.
- Return the matrix `x`.

-

Solution

```
# Define the oh_encoder function
def oh_encoder(tokens):
    x = np.zeros(N)
    for ingredient in tokens:
        # Get the index for each ingredient
        idx = ingredient_idx[ingredient]
        # Put 1 at the corresponding indices
        x[idx] = 1
    return x
```

Task 6: Instructions

Get the binary value of the tokens for each row of the matrix `A`.

- Inside the `for` loop:
  - Apply `oh_encoder()` to get a one-hot encoded matrix for each list of `tokens` in `corpus` (i.e., each product's ingredients list).
  - Increment `i` by 1.

-

Solution

```
# Make a document-term matrix
i = 0
for tokens in corpus:
    A[i, :] = oh_encoder(tokens)
    i += 1
```

Task 7: Instructions

Reduce the dimensions of the matrix using t-SNE.

- Create a `TSNE` instance with `n_components = 2`, `learning_rate = 200`, and `random_state = 42`. Assign it to `model`.

- Apply the `fit_transform()` method of `model` to the matrix `A`. Assign the result to `tsne_features`.
- Assign the first column of `tsne_features` to `moisturizers_dry['X']`.
- Assign the second column of `tsne_features` to `moisturizers_dry['Y']`.

Helpful links:

- t-SNE documentation
- Learning rates above 200 produce the ball effect that the t-SNE documentation warns about. The sweet spot appears to be learning rate values between 50 and 200. See this video or this video for more details.

-

Solution

```
# Dimension reduction with t-SNE
model = TSNE(n_components = 2, learning_rate = 200, random_state = 42)
tsne_features = model.fit_transform(A)

# Make X, Y columns
moisturizers_dry['X'] = tsne_features[:, 0]
moisturizers_dry['Y'] = tsne_features[:, 1]
```

Task 8: Instructions

Plot a scatter plot with the vectorized items.

- Create a `ColumnDataSource` with `moisturizers_dry`. Assign it to `source`.
- Label the x-axis as `T-SNE 1` and the y-axis as `T-SNE 2`.
- Add a circle renderer using `plot.circle()`, setting `x = 'X'`, `y = 'Y'`, and `source` to the `ColumnDataSource` you created.

Helpful links:

- `ColumnDataSource` documentation
- Bokeh plotting guide

-

Solution

```
from bokeh.io import show, output_notebook, push_notebook
from bokeh.plotting import figure
from bokeh.models import ColumnDataSource, HoverTool
output_notebook()

# Make a source and a scatter plot
```

```
source = ColumnDataSource(moisturizers_dry)
plot = figure(x_axis_label = 'T-SNE 1',
              y_axis_label = 'T-SNE 2',
              width = 500, height = 400)
plot.circle(x = 'X',
            y = 'Y',
            source = source,
            size = 10, color = '#FF7373', alpha = .8)
```

Task 9: Instructions

Add a hover tool.

- Set the `tooltips` argument to `('Item', '@Name')`, `('Brand', '@Brand')`, `('Price', '$@Price')`, and `('Rank', '@Rank')`.
- Add the new `hover` object to the plot.

Helpful links:

- Basic tooltips in Bokeh guide
-

Solution

```
# Create a HoverTool object
hover = HoverTool(tooltips = [('Item', '@Name'),
                              ('Brand', '@Brand'),
                              ('Price', '$@Price'),
                              ('Rank', '@Rank')])
plot.add_tools(hover)
```

Task 10: Instructions

Display the plot.

- Use the `show()` function to display the plot.

-

Solution

```
# Plot the map
show(plot)
```

Task 11: Instructions

Print out the ingredients for two similar products.

- Run the cell as is to print out the data and ingredients for Color Control Cushion Compact Broad Spectrum SPF 50+ and BB Cushion Hydra Radiance SPF 50.

Congratulations on reaching the end of the project!

You are welcome to compare other cosmetics by modifying the cosmetic name in the `cosmetic_1` and `cosmetic_2` code. Another interesting comparison: "Argan Cleansing Oil", "Phoenix Cell Regenerating Facial Oil", and "Juno Antioxidant + Superfood Face Oil".

-

Solution

```
# Print the ingredients of two similar cosmetics
cosmetic_1 = moisturizers_dry[moisturizers_dry['Name'] == "Color Control
Cushion Compact Broad Spectrum SPF 50+"]
cosmetic_2 = moisturizers_dry[moisturizers_dry['Name'] == "BB Cushion Hydra
Radiance SPF 50"]

# Display each item's data and ingredients
display(cosmetic_1)
print(cosmetic_1.Ingredients.values)
display(cosmetic_2)
print(cosmetic_2.Ingredients.values)
```

Kidney Stones and Simpson's Paradox

Task 1: Instructions

Load and inspect the data.

- Load the `readr` and `dplyr` packages.
- Read in `kidney_stone_data.csv` from the `datasets` folder using `read_csv()`.
- Inspect the first six rows of the data with `head()`.

Good to know

This project lets you apply the skills from Introduction to the Tidyverse, including filtering, grouping and summarizing data, and visualizing with ggplot2. We recommend that you take that course before starting this project. You will also revisit the methods covered in Logistic regression and display model outputs using the `broom` package.

Helpful links:

- `tidyverse` cheat sheet
- `ggplot2` cheat sheet
- `broom` introduction

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load the readr and dplyr packages
library(readr)
library(dplyr)

# Read datasets kidney_stone_data.csv into data
data <- read_csv("datasets/kidney_stone_data.csv")

# Take a look at the first few rows of the dataset
head(data)
```

Task 2: Instructions

Calculate the frequency of each treatment.

- Group by `treatment` and `success`, then find the total number of patients using `n()`, and calculate the frequency based on the sum of `N` of each treatment. Round to the third decimal.

Helpful links:

- `dplyr` cheat sheet
-

Solution

```
# Calculate the number and frequency of success and failure of each
treatment
data %>%
  group_by(treatment, success) %>%
  summarise(N = n()) %>%
  mutate(Freq = round(N/sum(N),3))
```

Task 3: Instructions

Calculate number and frequency of success and failure by stone size for each treatment and save the new data frame.

- Group by `treatment`, `stone_size` and `success`, then find the total number of patients using `n()`, and calculate the frequency based on the sum of `N` of each treatment and stone size combination. Round to the third decimal.
- Print out this data frame you just created.

Helpful links:

- `dplyr` cheat sheet
-

Solution

```
# Calculate number and frequency of success and failure by stone size for
each treatment
sum_data <-
  data %>%
  group_by(treatment, stone_size, success) %>%
  summarise(N = n()) %>%
  mutate(Freq = round(N/sum(N),3))

# Print out the data frame we just created
sum_data
```

Task 4: Instructions

Create a bar plot to show stone size count within each treatment.

- Load `ggplot2`.
- Use `ggplot()` to make a bar graph of count `N` as a function of `treatment`.
- In the `geom_bar()` aesthetics, set the `fill` to `stone_size` to represent large and small stones.
- Change the default `stat` setting so the heights of the bars represent values in the data.

Helpful links:

- `geom_bar()` function documentation
- `stat =` aesthetic options and behavior

-

Solution

```
# Load ggplot2
library(ggplot2)

# Create a bar plot to show stone size count within each treatment
sum_data %>%
  ggplot(aes(x = treatment, y = N)) +
  geom_bar(aes(fill = stone_size), stat = "identity")
```

Task 5: Instructions

Use the Chi-squared test to test if stone size is related to treatment assignment.

- Load the `broom` package.
- Use `chisq.test()` with the two variables to test as arguments.
- Use `tidy()` from `broom` to convert the test object into a tidy format (i.e., a data frame).

Helpful links:

- `chisq.test()` documentation
- `broom` package documentation

-

Solution

```
# Load the broom package
library(broom)

# Run a Chi-squared test
trt_ss <- chisq.test(data$treatment, data$stone_size)
```

```
# Print out the result in tidy format
tidy(trt_ss)
```

Task 6: Instructions

Fit a multiple logistic regression.

- Use `glm()` to run a generalized linear model. Add the correct dependent variable ("y"), the independent variables ("x1" and "x2"), and the family of the link function.
- Use `tidy()` to view the model coefficient table in a data frame format.

The model formula for a logistic regression takes the form `y ~ x1 + x2, family = "binomial"`.

Helpful links:

- Logistic regression
-

Solution

```
# Run a multiple logistic regression
m <- glm(data = data, success ~ stone_size + treatment, family =
'binomial')

# Print out model coefficient table in tidy format
tidy(m)
```

Task 7: Instructions

Visualize model output.

- Save the model coefficient table from the last task as the object `tidy_m`.
- Set up `ggplot()` canvas with the point estimate on the y-axis and the model terms on the x-axis. Calculate the upper bound of the 95% CI within `geom_pointrange()` and add a horizontal line at zero using `geom_hline()`.

Helpful links:

- Ways to draw vertical intervals including `geom_pointrange()` examples
- Adding `ggplot()` reference lines
-

Solution

```
# Save the tidy model output into an object
tidy_m <- tidy(m)

# Plot the coefficient estimates with 95% CI for each term in the model
tidy_m %>%
  ggplot(aes(x=term, y=estimate)) +
  geom_pointrange(aes(ymin=estimate-1.96*std.error,
                      ymax=estimate+1.96*std.error)) +
  geom_hline(yintercept = 0)
```

Task 8: Instructions

From the model coefficient table, make inference on what you learned from the data.

- Fill in the blanks with a character string "Yes" or "No" to answer the questions.
-

Solution

```
# Is small stone more likely to be a success after controlling for
treatment option effect?
# Options: Yes, No (as string)
small_high_success <- "Yes"

# Is treatment A significantly better than B?
# Options: Yes, No (as string)
A_B_sig <- "No"
```

What Makes a Pokémon Legendary?

Task 1: Instructions

Load and prepare the Pokédex.

- Load the `tidyverse`.
- Convert `type` and `is_legendary` to factors.
- Look at the first six rows of the Pokédex.
- Examine the structure.

Good to know

This project uses the `tidyverse` suite of packages, particularly `dplyr` and `ggplot2`, so it would be useful to have some familiarity with those packages beforehand. If you need to brush up, work through DataCamp's Introduction to the Tidyverse course before you begin.

Students should also have a general knowledge of classification problems in machine learning, as taught through Supervised Learning in R: Classification. In particular, they should have experience with tree-based models including classification trees and random forests, which are covered in detail in Machine Learning with Tree-Based Models in R.

For a concise introduction to decision trees in R, see James Le's tutorial.

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Load the tidyverse
library(tidyverse)

# Import the dataset and convert variables
pokedex <- read_csv("datasets/pokedex.csv",
                    col_types = cols(name = col_factor(),
                                     type = col_factor(),
                                     is_legendary = col_factor()))

# Look at the first six rows
head(pokedex)

# Examine the structure
str(pokedex)
```

Task 2: Instructions

Count the number of legendary/non-legendary Pokémon.

- Count the number of observations in each group of `is_legendary`.
- Divide by the number of rows in the Pokédex.
- Print the data frame.

The `count()` function from `dplyr` tallies the number of observations in each group of the variable you pass to it. The following code – `df %>% count(x)` – is simply shorthand for:

`df %>% group_by(x) %>% summarize(n = sum(x)) %>% ungroup()`.

Helpful links:

- `dplyr` cheatsheet
- `count()` function documentation

-

Solution

```
# Prepare the data
legendary_pokemon <- pokedex %>%
  count(is_legendary) %>%
  mutate(prop = n / nrow(pokedex))

# Print the data frame
legendary_pokemon
```

Task 3: Instructions

Compare the height and weight of legendary and non-legendary Pokémon.

- Map `x` and `y` to `height_m` and `weight_kg` respectively.
- Map the `color` of `geom_point()` to `is_legendary`.
- Set the `label` to print if `height_m > 7.5` OR if `weight_kg > 600`.
- Expand the limit of the x-axis to 16.

The `expand_limits()` function from `ggplot2` allows you to expand the plot limits, either on the x-axis or the y-axis.

Helpful links:

- `ggplot2` cheatsheet

- `expand_limits()` function documentation

-

Solution

```
# Prepare the plot
legend_by_heightweight_plot <- pokedex %>%
  ggplot(aes(x = height_m, y = weight_kg)) +
  geom_point(aes(color = is_legendary), size = 2) +
  geom_text(aes(label = ifelse(height_m > 7.5|weight_kg > 600,
as.character(name), '')),
            vjust = 0, hjust = 0) +
  geom_smooth(method = "lm", se = FALSE, col = "black", linetype =
"dashed") +
  expand_limits(x = 16) +
  labs(title = "Legendary Pokemon by height and weight",
       x = "Height (m)",
       y = "Weight (kg)") +
  guides(color = guide_legend(title = "Pokemon status")) +
  scale_color_manual(labels = c("Non-Legendary", "Legendary"),
                     values = c("#F8766D", "#00BFC4"))

# Print the plot
legend_by_heightweight_plot
```

Task 4: Instructions

Examine the proportion of legendary/non-legendary Pokémon by type.

- Group the data by `type`.
- Calculate the proportion of legendary pokemon (`prop_legendary`) in each type by taking the mean of `is_legendary`.
- Reorder `type` by the proportion of legendary Pokémon.
- Map both `y` and `fill` to the proportion of legendary Pokémon.

`fct_reorder()` is a handy function from `forcats` that allows you to reorder factor levels according to the values of another variable, rather than alphabetically. It is particularly useful when you want to reorder categorical data for a bar or column chart.

Helpful links:

- `forcats` package documentation
- `fct_reorder()` function documentation

-

Solution

```
# Prepare the data
legend_by_type <- pokedex %>%
```

```
    group_by(type) %>%
    mutate(is_legendary = as.numeric(is_legendary) - 1) %>%
    summarise(prop_legendary = mean(is_legendary)) %>%
    ungroup() %>%
    mutate(type = fct_reorder(type, prop_legendary))

# Prepare the plot
legend_by_type_plot <- legend_by_type %>%
    ggplot(aes(x = type, y = prop_legendary, fill = prop_legendary)) +
    geom_col() +
    labs(title = "Legendary Pokemon by type") +
    coord_flip() +
    guides(fill = FALSE)

# Print the plot
legend_by_type_plot
```

Task 5: Instructions

Compare the fighter stats of legendary/non-legendary Pokémon.

- Select the six relevant columns from the Pokédex.
- Gather the data into key-value pairs called `fght_stats` and `value` respectively, while excluding `is_legendary`.
- Use `facet_wrap()` so that each facet represents a value from `fght_stats`.
- Remove the legend using the `guides()` function.

Faceting is an efficient way to visualize data when it can be split by one or more categorical variables. There are two ways to facet data in `ggplot2`: `facet_wrap()` and `facet_grid()`. These two functions are similar and can produce identical output. As a rule of thumb, however, `facet_wrap()` is more convenient when you want to facet by a single variable, while `facet_grid()` is better for when you need to facet by two.

Helpful links:

- `facet_wrap()` function documentation
- `facet_grid()` function documentation
- `gather()` function documentation
- "Remove ggplot legend" question on Stack Overflow

-

Solution

```
# Prepare the data
legend_by_stats <- pokedex  %>%
  select(is_legendary, attack, sp_attack, defense, sp_defense, hp, speed)
%>%
  gather(key = "fght_stats", value = "value", -is_legendary)

# Prepare the plot
```

```
legend_by_stats_plot <- legend_by_stats %>%
 ggplot(aes(x = is_legendary, y = value, fill = is_legendary)) +
 geom_boxplot(varwidth = TRUE) +
 facet_wrap(~fght_stats) +
 labs(title = "Pokemon fight statistics",
        x = "Legendary status") +
 guides(fill = FALSE)

# Print the plot
legend_by_stats_plot
```

Task 6: Instructions

Split the Pokédex into a training set and a test set.

- Set the seed to `1234`.
- Save the number of rows in the Pokédex to `n`.
- Multiply `n` by 0.6 to generate a 60% sample.
- Create training and test sets by using the `sample_rows` object.

The `sample()` function from base R takes a sample of specified size from the elements of some vector, `x`. For example, if you wanted to take a sample of 100 observations from `x`, you would write `sample(x, 100)`. By default, the function samples *without* replacement (which is preferable in this case).

Helpful links:

- `sample()` function documentation
-

Solution

```
# Set seed for reproducibility
set.seed(1234)

# Save number of rows in dataset
n <- nrow(pokedex)

# Generate 60% sample of rows
sample_rows <- sample(n, 0.6 * n)

# Create training set
pokedex_train <- pokedex  %>%
  filter(row_number() %in% sample_rows)

# Create test set
pokedex_test <- pokedex  %>%
  filter(!row_number() %in% sample_rows)
```

Task 7: Instructions

Fit and plot a decision tree.

- Load the `rpart` and `rpart.plot` packages and set the seed to `1234`.
- Fit the decision tree to the training data.
- Omit incomplete observations.
- Plot the decision tree using `rpart.plot()`.

`rpart()` is the flagship function from the `rpart` package. It uses the standard formula interface, where the syntax `y ~ a + b` is used to model `y` as a function of both `a` and `b`. `rpart()` can be used to fit both regression and classification models, but since we are fitting a classification tree, we set the `method` argument to `"class"`.

Helpful links:

- `rpart` package documentation
- `rpart()` function documentation
- `rpart.plot()` function documentation

-

Solution

```
# Load packages and set seed
library(rpart)
library(rpart.plot)
set.seed(1234)

# Fit decision tree
model_tree <- rpart(is_legendary ~ attack + defense + height_m +
                    hp + sp_attack + sp_defense + speed + type + weight_kg,
                       data = pokedex_train,
                       method = "class",
                       na.action = na.omit)

# Plot decision tree
rpart.plot(model_tree)
```

Task 8: Instructions

Fit the random forest.

- Load the `randomForest` package and set the seed to `1234`.
- Fit the random forest to the training data.
- Omit incomplete observations.
- Print the model output.

`randomForest()` is the flagship function from the `randomForest` package. It uses the standard formula interface, where the syntax `y ~ a + b` is used to model `y` as a function of `a`

and `b`. In order to return both of the variable importance measures associated with random forests (see Task 10), we set the `importance` argument equal to `TRUE`.

Helpful links:

- `randomForest` package documentation
- `randomForest()` function documentation

-

Solution

```
# Load package and set seed
library(randomForest)
set.seed(1234)

# Fit random forest
model_forest <- randomForest(is_legendary ~ attack + defense + height_m +
                          hp + sp_attack + sp_defense + speed + type +
weight_kg,
                          data = pokedex_train,
                          importance = TRUE,
                          na.action = na.omit)

# Print model output
model_forest
```

Task 9: Instructions

Plot ROC curves for the decision tree and random forest.

- Using the random forest model, predict the probability of being legendary for the Pokémon in the test set.
- Create a `prediction` object for the random forest.
- Create a `performance` object for the random forest.
- Plot the ROC curves for `perf_tree` and `perf_forest`.

The `ROCR` package holds a number of useful functions for evaluating the performance of classification models. Every evaluation starts with the creation of a `prediction` object, which transforms the input data into a standardized format. We can then use the `performance` function to return various performance measures, including true positive rate (`tpr`), false positive rate (`fpr`), accuracy (`acc`) and error rate (`err`).

Helpful links:

- `ROCR` package documentation

-

Solution

```
# Load the ROCR package
library(ROCR)

# Create prediction and performance objects for the decision tree
probs_tree <- predict(model_tree, pokedex_test, type = "prob")
pred_tree <- prediction(probs_tree[,2], pokedex_test$is_legendary)
perf_tree <- performance(pred_tree, "tpr", "fpr")

# Create prediction and performance objects for the random forest
probs_forest <- predict(model_forest, pokedex_test, type = "prob")
pred_forest <- prediction(probs_forest[,2], pokedex_test$is_legendary)
perf_forest <- performance(pred_forest, "tpr", "fpr")

# Plot the ROC curves
plot(perf_tree, col = "red", main = "ROC curves")
plot(perf_forest, add = TRUE, col = "blue")
legend(x = "bottomright",  legend = c("Decision Tree", "Random Forest"),
fill = c("red", "blue"))
```

Task 10: Instructions

Analyze the variable importance results from the random forest.

- Print variable importance using the `importance()` function.
- Plot variable importance using the `varImpPlot()` function.

It is worth noting that variable importance measures are specific to the model you have fitted. It is therefore possible that changing the specification of your model – for example by adding/removing variables or tuning hyperparameters – will also change your interpretation of relative variable importance. Nevertheless, the variable importance results of a random forest are more stable than for a single classification tree and can therefore be interpreted with greater confidence.

Helpful links:

- `importance()` function documentation
- `varImpPlot()` function documentation

-

Solution

```
# Print variable importance measures
importance_forest <- importance(model_forest)
importance_forest

# Create a dotchart of variable importance
varImpPlot_forest <- varImpPlot(model_forest)
varImpPlot_forest
```

Task 11: Instructions

Answer Professor Oak's questions about the variable importance results.

- Answer Q1 and Q2 with regard to `MeanDecreaseAccuracy`.
- Answer Q3 and Q4 with regard to `MeanDecreaseGini`.

Make sure that your response to each question is a string, e.g. `"attack"` or `"defense"`.

One aspect of machine learning not covered in this project is **hyperparameter tuning**, which helps you to achieve the most accurate results for your predictive model. For more on this topic, take DataCamp's [Hyperparameter Tuning in R]() course.

-

Solution

```
# According to the MeanDecreaseAccuracy plot:

# Q1. Is the `attack` or `defense` variable more important?
answer1 <- "attack"

# Q2. Is the `weight_kg` or `height_m` variable more important?
answer2 <- "weight_kg"

# According to the MeanDecreaseGini plot:

# Q3. Is the `attack` or `defense` variable more important?
answer3 <- "defense"

# Q4. Is the `weight_kg` or `height_m` variable more important?
answer4 <- "weight_kg"
```

Analyze Your Runkeeper Fitness Data

Task 1: Instructions

Load `pandas` and the training activities data.

- Import `pandas` under the alias `pd`.
- Use the `read_csv()` function to load the dataset (`runkeeper_file`) into a variable called `df_activities`. Parse the dates with the `parse_dates` parameter and set the index to the `Date` column using the `index_col` parameter.
- Display 3 random rows from `df_activities` using the `sample()` method.
- Print a summary of `df_activities` using the `info()` method.

This project lets you apply the skills from Data Manipulation with pandas, Manipulating Time Series Data in Python, and Visualizing Time Series Data in Python.

Helpful links:

- `read_csv()` function documentation
- `sample()` method documentation

-

Solution

```
# Import pandas
import pandas as pd

# Define file containing dataset
runkeeper_file = 'datasets/cardioActivities.csv'

# Create DataFrame with parse_dates and index_col parameters
df_activities = pd.read_csv(runkeeper_file, parse_dates=True,
index_col='Date')

# First look at exported data: select sample of 3 random rows
display(df_activities.sample(n=3))

# Print DataFrame summary
df_activities.info()
```

Task 2: Instructions

Implement the following data preprocessing tasks:

- Delete unnecessary columns from `df_activities` with the `drop()` method, setting the `columns` parameter to the `cols_to_drop` list.
- Calculate the activity type counts using the `value_counts()` method on the `Type` column.
- Rename the 'Other' values to 'Unicycling' in the `Type` column using `str.replace()`.
- Count the missing values in each column using `isnull().sum()`.

Helpful links:

- `drop()` function documentation
- `str.replace()` function documentation
- `isnull()` function documentation

-

Solution

```
# Define list of columns to be deleted
cols_to_drop = ['Friend\'s Tagged','Route Name','GPX File','Activity
Id','Calories Burned', 'Notes']

# Delete unnecessary columns
df_activities.drop(columns=cols_to_drop, inplace=True)

# Count types of training activities
display(df_activities['Type'].value_counts())

# Rename 'Other' type to 'Unicycling'
df_activities['Type'] = df_activities['Type'].str.replace('Other',
'Unicycling')

# Count missing values for each column
df_activities.isnull().sum()
```

Task 3: Instructions

Implement mean imputation for missing values.

- Calculate the sample mean for `Average Heart Rate (bpm)` for the 'Cycling' activity type. Assign the result to `avg_hr_cycle`.
- Filter the `df_activities` for the 'Cycling' activity type. Create a copy of the result using `copy()` and assign the copy to `df_cycle`.
- Fill in the missing values for `Average Heart Rate (bpm)` in `df_cycle` with `int(avg_hr_cycle)` using the `fillna()` method.
- Count the missing values for all columns in `df_run`.

Helpful links:

- `fillna()` method documentation
-

Solution

```
# Calculate sample means for heart rate for each training activity type
avg_hr_run = df_activities[df_activities['Type'] == 'Running']['Average
Heart Rate (bpm)'].mean()
avg_hr_cycle = df_activities[df_activities['Type'] == 'Cycling']['Average
Heart Rate (bpm)'].mean()

# Split whole DataFrame into several, specific for different activities
df_run = df_activities[df_activities['Type'] == 'Running'].copy()
df_walk = df_activities[df_activities['Type'] == 'Walking'].copy()
df_cycle = df_activities[df_activities['Type'] == 'Cycling'].copy()

# Filling missing values with counted means
df_walk['Average Heart Rate (bpm)'].fillna(110, inplace=True)
df_run['Average Heart Rate (bpm)'].fillna(int(avg_hr_run), inplace=True)
df_cycle['Average Heart Rate (bpm)'].fillna(int(avg_hr_cycle),
inplace=True)

# Count missing values for each column in running data
df_run.isnull().sum()
```

Task 4: Instructions

Plot running data from 2013 through 2018.

- Subset `df_run` for data from 2013 through 2018. Take into account that observations in dataset stored in chronological order - most recent records first. Assign the result to `runs_subset_2013_2018`.
- In the plotting code, enable subplots by setting the `subplots` parameter to `True`. Don't use spaces around the `=` sign when used to indicate a keyword argument, as recommended in PEP 8 style guide for Python code.
- Show the plot using `plt.show()`.

Helpful links:

- Subset time series data exercise from Visualizing Time Series Data in Python
- `pandas.DataFrame.plot` documentation
- `matplotlib` cheat sheet
- PEP 8 guide: Other recommendations
-

Solution

```
%matplotlib inline
```

```
# Import matplotlib, set style and ignore warning
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
plt.style.use('ggplot')
warnings.filterwarnings(
    action='ignore', module='matplotlib.figure', category=UserWarning,
    message=('This figure includes Axes that are not compatible with
tight_layout, so results might be incorrect.')
)

# Prepare data subsetting period from 2013 till 2018
runs_subset_2013_2018 = df_run['2018':'2013']

# Create, plot and customize in one step
runs_subset_2013_2018.plot(subplots=True,
                           sharex=False,
                           figsize=(12,16),
                           linestyle='none',
                           marker='o',
                           markersize=3,
                          )

# Show plot
plt.show()
```

Task 5: Instructions

Calculate annual and weekly means for `Distance (km)`, `Average Speed (km/h)`, `Climb (m)` and `Average Heart Rate (bpm)`.

- Subset `df_run` for data from 2015 through 2018. Assign the result to `runs_subset_2015_2018`.
- Count the annual averages using `resample()` with 'A' alias, and the `mean()` method for `runs_subset_2015_2018`.
- Count the average weekly statistics using `resample()` with 'W' alias, and the `mean()` method twice.
- Filter from dataset column `Distance (km)` and count the average number of trainings per week using `resample()` with the `count()` and `mean()` methods. Assign the result to `weekly_counts_average`.

Helpful links:

- Resampling time series data exercise from Manipulating Time Series Data in Python
- `resample()` function documentation

-

Solution

```
# Prepare running data for the last 4 years
runs_subset_2015_2018 = df_run['2018':'2015']

# Calculate annual statistics
print('How my average run looks in last 4 years:')
display(runs_subset_2015_2018.resample('A').mean())

# Calculate weekly statistics
print('Weekly averages of last 4 years:')
display(runs_subset_2015_2018.resample('W').mean().mean())

# Mean weekly counts
weekly_counts_average = runs_subset_2015_2018['Distance
(km)'].resample('W').count().mean()
print('How many trainings per week I had on average:',
weekly_counts_average)
```

Task 6: Instructions

Prepare data and create a plot.

- Select information for distance and then for heart rate from `runs_subset_2015_2018` and assign to `runs_distance` and `runs_hr`, respectively.
- Create two subplots with shared x-axis using the `plt.subplots()` method, setting the first positional parameter to 2, `sharex` to `True`, and `figsize` to `(12,8)`. Assign the output to `fig, (ax1, ax2)` variables.
- Plot distance on the first subplot, setting parameter `ax` to `ax1`.
- On the second subplot (`ax2`), add a horizontal line with `axhline()` for the average value of heart rate counted as `runs_hr.mean()`. Set `color` to `'blue'`, `linewidth` to `1`, and `linestyle` to `'-.'`.

-

Solution

```
# Prepare data
runs_subset_2015_2018 = df_run['2018':'2015']
runs_distance = runs_subset_2015_2018['Distance (km)']
runs_hr = runs_subset_2015_2018['Average Heart Rate (bpm)']

# Create plot
fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

# Plot and customize first subplot
runs_distance.plot(ax=ax1)
ax1.set(ylabel='Distance (km)', title='Historical data with averages')
ax1.axhline(runs_distance.mean(), color='blue', linewidth=1,
linestyle='-.')

# Plot and customize second subplot
runs_hr.plot(ax=ax2, color='gray')
ax2.set(xlabel='Date', ylabel='Average Heart Rate (bpm)')
```

```
ax2.axhline(runs_hr.mean(), color='blue', linewidth=1, linestyle='-.')

# Show plot
plt.show()
```

Task 7: Instructions

Prepare data and create a plot.

- Subset `df_run` for data from 2013 through 2018 and select the `Distance (km)` column. Count annual totals with `resample()` and `sum()`. Assign the result to `df_run_dist_annual`.
- Create a plot with `plt.figure()`, setting `figsize` to define a plot of size 8.0 inches x 5.0 inches.
- Customize the plot with horizontal span from 0 to 800 km with `ax.axhspan()`. Set `color` to `'red'` and `alpha` to `0.2`.
- Show the plot with `plt.show()`.

-

Solution

```
# Prepare data
df_run_dist_annual = df_run['2018':'2013']['Distance
(km)'].resample('A').sum()

# Create plot
fig = plt.figure(figsize=(8, 5))

# Plot and customize
ax = df_run_dist_annual.plot(marker='*', markersize=14, linewidth=0,
color='blue')
ax.set(ylim=[0, 1210],
       xlim=['2012','2019'],
       ylabel='Distance (km)',
       xlabel='Years',
       title='Annual totals for distance')

ax.axhspan(1000, 1210, color='green', alpha=0.4)
ax.axhspan(800, 1000, color='yellow', alpha=0.3)
ax.axhspan(0, 800, color='red', alpha=0.2)

# Show plot
plt.show()
```

Task 8: Instructions

Create a plot with observed distance of runs and decomposed trend.

- Import the `statsmodels.api` under the alias `sm`.

- Subset `df_run` from 2013 through 2018, select `Distance (km)` column, resample weekly, and fill NaN values with the `bfill()` method. Assign to `df_run_dist_wkly`.
- Create a plot with `plt.figure()`, defining plot size by setting `figsize` to `(12,5)`.

Helpful links:

- `seasonal_decompose()` using moving averages documentation
-

Solution

```
# Import required library
import statsmodels.api as sm

# Prepare data
df_run_dist_wkly = df_run['2018':'2013']['Distance
(km)'].resample('W').bfill()
decomposed = sm.tsa.seasonal_decompose(df_run_dist_wkly,
extrapolate_trend=1, freq=52)

# Create plot
fig = plt.figure(figsize=(12,5))

# Plot and customize
ax = decomposed.trend.plot(label='Trend', linewidth=2)
ax = decomposed.observed.plot(label='Observed', linewidth=0.5)

ax.legend()
ax.set_title('Running distance trend')

# Show plot
plt.show()
```

Task 9: Instructions

Create a customized histogram for heart rate distribution.

- Subset `df_run` from March 2015 through 2018 then select the `Average Heart Rate (bpm)` column. Assign the result to `df_run_hr_all`.
- Create a plot with `plt.subplots()`, setting `figsize` to `(8,5)`. Assign the result to `fig, ax`.
- Create customized x-axis ticks with `ax.set_xticklabels()`. Set the parameters `labels` to `zone_names`, `rotation` to `-30`, and `ha` to `'left'`.
- Show the plot with `plt.show()`.

-

Solution

```
# Prepare data
hr_zones = [100, 125, 133, 142, 151, 173]
zone_names = ['Easy', 'Moderate', 'Hard', 'Very hard', 'Maximal']
zone_colors = ['green', 'yellow', 'orange', 'tomato', 'red']
df_run_hr_all = df_run['2018':'2015-03']['Average Heart Rate (bpm)']

# Create plot
fig, ax = plt.subplots(figsize=(8,5))

# Plot and customize
n, bins, patches = ax.hist(df_run_hr_all, bins=hr_zones, alpha=0.5)
for i in range(0, len(patches)):
    patches[i].set_facecolor(zone_colors[i])

ax.set(title='Distribution of HR', ylabel='Number of runs')
ax.xaxis.set(ticks=hr_zones)
ax.set_xticklabels(labels=zone_names, rotation=-30, ha='left')

# Show plot
plt.show()
```

Task 10: Instructions

Create a summary report.

- Concatenate the `df_run` DataFrame with `df_walk` and `df_cycle` using `append()`, then sort based on the index in descending order. Assign the result to `df_run_walk_cycle`.
- Group `df_run_walk_cycle` by activity type, then select the columns in `dist_climb_cols`. Sum the result using `sum()`. Assign the result to `df_totals`.
- Use the `stack()` method on `df_summary` to show a compact reshaped form of the full summary report.

Helpful links:

- `describe()` function documentation
- `stack()` method documentation

-

Solution

```
# Concatenating three DataFrames
df_run_walk_cycle =
df_run.append(df_walk).append(df_cycle).sort_index(ascending=False)

dist_climb_cols, speed_col = ['Distance (km)', 'Climb (m)'], ['Average
Speed (km/h)']
```

```
# Calculating total distance and climb in each type of activities
df_totals = df_run_walk_cycle.groupby('Type')[dist_climb_cols].sum()

print('Totals for different training types:')
display(df_totals)

# Calculating summary statistics for each type of activities
df_summary = df_run_walk_cycle.groupby('Type')[dist_climb_cols +
speed_col].describe()

# Combine totals with summary
for i in dist_climb_cols:
    df_summary[i, 'total'] = df_totals[i]

print('Summary statistics for different training types:')
df_summary.stack()
```

Task 11: Instructions

Use FUN FACTS data to answer some fun questions.

- Calculate the instructor's average shoes per lifetime. Use number of 'Total number of km run' from FUN FACTS and divide by the number of pairs of shoes gone through.
- Calculate an estimated number of shoes gone through for Forrest Gump's route. Use 'Total number of km run' from FORREST RUN FACTS, then divide (using floor division) by the result from the previous step.

Congratulations on reaching the end of the project!

-

Solution

```
# Count average shoes per lifetime (as km per pair) using our fun facts
average_shoes_lifetime = 5224 / 7

# Count number of shoes for Forrest's run distance
shoes_for_forrest_run = 24700 // average_shoes_lifetime

print('Forrest Gump would need {} pairs of
shoes!'.format(shoes_for_forrest_run))
```

Modeling the Volatility of US Bond Yields

Task 1: Instructions

Load and prepare the time series data of US yields.

- Load the `xts` and `readr` packages.
- Read in the dataset `datasets/FED-SVENY.csv` with `read_csv()`.
- Convert the data to an xts object using `as.xts()`. Pass the data set without the first column, then use the `Date` column as the `order.by` parameter.
- Finally look at the last rows of the 1st, 5th, 10th, 20th and 30th columns.

The **eXtensible Time Series** (`xts`) package is a useful tool to store, manipulate, and analyze time series data. Converting a dataset into an xts object is done with `as.xts()`. The first parameter is the object containing the data without the column of dates. The second parameter is a vector of dates. Use the dollar sign ($) to select the date column.

Helpful links:

- `read_csv()` function documentation
- `as.xts()` function documentation

Good to know

This project assumes a general knowledge of xts objects and GARCH models. If you do not know what these are, you are encouraged to take the following DataCamp courses.

- Introduction to Time Series Analysis presents the statistical tools used to analyze time series data.

- Manipulating Time Series Data in R with xts & zoo presents `xts` and advanced tips and tricks for working with time series data in R.

- GARCH Models in R is a good course on GARCH models and the `rugarch` package.

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

-

  Solution

```
# Load the packages
library(xts)
library(readr)

# Load the data
yc_raw <- read_csv("datasets/FED-SVENY.csv")

# Convert the data into xts format
yc_all <- as.xts(x = yc_raw[, -1], order.by = yc_raw$Date)

# Show only the 1st, 5th, 10th, 20th and 30th columns
yc_all_tail <- tail(yc_all[, c(1, 5, 10, 20, 30)])
yc_all_tail
```

Task 2: Instructions

Make a plot of the yields over time.

- Define the plot arguments. `yields` is the xts object and `plot.type` is "single". Set `plot.palette` to 30 colors from `viridis()`, and `asset.names` to the column names of the xts object.
- Use `plot.zoo()` to visualize the time series.
- Add the `legend()` and set the legend parameter to `asset.names`.

The `plot.zoo()` function extends base R plots. Its parameter, `plot.type`, controls how multiple time series appear in a plotting window - as one plot or separately in many small plots.

Helpful links:

- `plot.zoo()` function documentation
- `legend()` function documentation
- `viridis()` function documentation

-

Solution

```
library(viridis)

# Define plot arguments
yields  <- yc_all
plot.type  <- "single"
plot.palette <- viridis(30)
asset.names <- colnames(yc_all)

# Plot the time series
plot.zoo(x = yields, plot.type = plot.type, col = plot.palette)

# Add the legend
legend(x = "topleft", legend = asset.names,
       col = plot.palette, cex = 0.45, lwd = 3)
```

Task 3: Instructions

Calculate the yield differences in the time series.

- Use the `diff.xts()` function on `yc_all` and assign the output to `ycc_all`.
- Print the last few rows of the differentiated series, showing only the 1st, 5th, 10th, 20th and 30th columns.

Helpful links:

- `diff.xts()` function documentation
-

Solution

```
# Differentiate the time series
ycc_all <- diff.xts(yc_all)

# Show the tail of the 1st, 5th, 10th, 20th and 30th columns
ycc_all_tail <- tail(ycc_all[, c(1, 5, 10, 20, 30)])
ycc_all_tail
```

Task 4: Instructions

Plot the differentiated series.

- Define the plot arguments. `yields.change` is the differentiated xts object and `plot.type` is "multiple".
- Use `plot.zoo()` to visualize the time series.

The `plot.zoo()` function extends base R plots. Its parameter, `plot.type`, controls how multiple time series appear in a plotting window - as one plot or separately in many small plots. `plot.palette` is still defined from Task 2.

Helpful links:

- `plot.zoo()` function documentation
-

Solution

```
# Define the plot parameters
yield.changes <- ycc_all
plot.type <- "multiple"
```

```
# Plot the differtianted time series
plot.zoo(x = yield.changes, plot.type = plot.type,
     ylim = c(-0.5, 0.5), cex.axis = 0.7,
     ylab = 1:30, col = plot.palette)
```

Task 5: Instructions

Filter the data and perform a more detailed analysis.

- Filter `ycc_all` for observations from 2000 onward.
- Assign the 1-year and 20-year maturities to new variables to `x_1` and `x_20` respectively.
- Use the `acf()` function to plot the autocorrelations for the two series.
- Compute the absolute values of the two series inside the `acf()` function.

Helpful links:

- Filter and select variables in time series with ease: [xts Cheat Sheet: Time Series in R]
- Combine multiple plots: `par()` function [documentation].
- `acf()` function [documentation]
- `abs()` function [documentation]

-

Solution

```
# Filter for changes in and after 2000
ycc <- ycc_all["2000/", ]

# Save the 1-year and 20-year maturity yield changes into separate
variables
x_1 <- ycc[, "SVENY01"]
x_20 <- ycc[, "SVENY20"]

# Plot the autocorrelations of the yield changes
par(mfrow=c(2,2))
acf_1 <- acf(x_1)
acf_20 <- acf(x_20)

# Plot the autocorrelations of the absolute changes of yields
acf_abs_1 <- acf(abs(x_1))
acf_abs_20 <- acf(abs(x_20))
```

Task 6: Instructions

Build a GARCH model to explain volatility and plot the results.

- Use `ugarchspec()` to set the distribution parameter to the skewed t-distribution, `"sstd"`.
- Run the model using `ugarchfit()` on the 1-year maturity yield changes, `x_1`.
- Use `sigma()` and `residuals()` functions to extract the volatilities and residuals from the fitted model. The latter should be standardized.
- Merge the time series data, `x_1`, and modeled volatilities and residuals using `merge.xts()`, then plot them using the `plot.zoo()` function.

`ugarchspec()` defines the form of the GARCH model through the `distribution.model` parameter. The actual modeling is done by `ugarchfit()`.

`sigma()` and `residuals()` extract time-varying volatilities and residuals respectively. Standardized residuals are residuals adjusted by the estimated volatilities. Residuals are also scaled to the same standard deviation as the original series for comparison.

`merge.xts()` takes different time series data and joins them by their time indices.

Helpful links:

- `ugarchspec()` function documentation
- `ugarchfit()` function documentation
- `sigma()` function documentation
- `residuals()` function documentation
- `merge.xts()` function documentation

-

Solution

```
library(rugarch)
# Specify the GARCH model with the skewed t-distribution
spec <- ugarchspec(distribution.model = "sstd")

# Fit the model
fit_1 <- ugarchfit(x_1, spec = spec)

# Save the volatilities and the rescaled residuals
vol_1 <- sigma(fit_1)
res_1 <- scale(residuals(fit_1, standardize = TRUE)) * sd(x_1) + mean(x_1)

# Plot the yield changes with the estimated volatilities and residuals
merge_1 <- merge.xts(x_1, vol_1, res_1)
plot.zoo(merge_1)
```

Task 7: Instructions

Build and plot a GARCH model using the 20-year maturity yield changes.

- Using the same distribution model, fit a GARCH using `ugarchfit()` on the 20-year maturity yield changes, `x_20`.
- Use `sigma()` and `residuals()` functions to extract the volatilities and residuals from the fitted model. The latter should be standardized.
- Merge the time series data, `x_20`, and modeled volatilities and residuals using merge.xts(), then plot them using `plot.zoo()`.

Helpful links:

- `ugarchspec()` function documentation
- `ugarchfit()` function documentation
- `sigma()` function documentation
- `residuals()` function documentation
- `merge.xts()` function documentation

-

Solution

```
# Fit the model
fit_20 <- ugarchfit(x_20, spec = spec)

# Save the volatilities
vol_20 <- sigma(fit_20)
res_20 <- scale(residuals(fit_20, standardize = TRUE)) * sd(x_20) +
mean(x_20)

# Plot the yield changes with the estimated volatilities and residuals
merge_20 <- merge.xts(x_20, vol_20, res_20)
plot.zoo(merge_20)
```

Task 8: Instructions

Plot the density of the three distributions.

- Calculate the `density()` of the original 1-year yield changes and the rescaled residuals.
- Plot the density kernel of the 1-year yield with `plot()` and the density kernel of the residuals with `lines()`.
- For comparison, add the density of the normal distribution. Use the `dnorm()` to specify it with the appropriate `mean` and `sd` parameters.
- Add the following labels to the `legend()`: *Before GARCH*, *After GARCH*, *Normal distribution*.

Helpful links:

- `density()` function documentation
- `lines()` function documentation

- `dnorm()` function documentation
- `legend()` function documentation

-

Solution

```
# Calculate the kernel density for the 1-year maturity and residuals
density_x_1 <- density(x_1)
density_res_1 <- density(res_1)

# Plot the density digaram for the 1-year maturity and residuals
plot(density_x_1)
lines(density_res_1, col = "red")

# Add the normal distribution to the plot
norm_dist <- dnorm(seq(-0.4, 0.4, by = .01), mean = mean(x_1), sd =
sd(x_1))
lines(seq(-0.4, 0.4, by = .01),
      norm_dist,
      col = "darkgreen"
     )

# Add legend
legend <- c("Before GARCH", "After GARCH", "Normal distribution")
legend("topleft", legend = legend,
       col = c("black", "red", "darkgreen"), lty=c(1,1))
```

Task 9: Instructions

Draw the Q-Q plots for the 1-year yield changes and residuals.

- Define the data to plot: 1-year maturity yield changes and 1-year residuals.
- Use `qnorm` as the benchmark distribution (do not use " ").
- Use `qqnorm()` to plot the empirical quantiles against the theoretical quantiles for the 1-year maturity. Also draw the theoretical line of the normal distribution using `qqline()`.
- Repeat the previous steps to plot the residuals.

Helpful links:

- `qqnorm()` and `qqline()` function documentations.
- `qnorm()` function documentation.

-

Solution

```
# Define plot data: the 1-year maturity yield changes and the residuals
data_orig <- x_1
```

```
data_res <- res_1

# Define the benchmark distribution (qnorm)
distribution <- qnorm

# Make the Q-Q plot of original data with the line of normal distribution
qqnorm(data_orig, ylim = c(-0.5, 0.5))
qqline(data_orig, distribution = distribution, col = "darkgreen")

# Make the Q-Q plot of GARCH residuals with the line of normal distribution
par(new=TRUE)
qqnorm(data_res * 0.623695122815242, col = "red", ylim = c(-0.5, 0.5))
qqline(data_res * 0.623695122815242, distribution = distribution, col =
"darkgreen")
legend("topleft", c("Before GARCH", "After GARCH"), col = c("black",
"red"), pch=c(1,1))
```

Task 10: Instructions

Answer the questions about the GARCH model we used.

- Q1: Do the extracted volatilities in Task 6 and Task 7 explain the changing magnitude of the original series?
- Q2: Do the distributions in Task 8 and Task 9 come closer to the normal distribution after GARCH?
- Q3: Which time series has the highest absolute autocorrelation and largest volatility fluctuations? Look at the results of Task 5, Task 6, and Task 7.

Financial risk modeling is a rich topic and we covered only a small fraction of it. For more on this topic browse DataCamp's R Applied Finance Courses

-

Solution

```
# Q1: Did GARCH revealed how volatility changed over time? # Yes or No?
(Q1 <- "Yes")

# Q2: Did GARCH bring the residuals closer to normal distribution? Yes or
No?
(Q2 <- "Yes")

# Q3: Which time series shows the most erratic behaviour? Choose 1 or 20.
(Q3 <- 1)
```

Disney Movies and Box Office Success

Task 1: Instructions

Load in the dataset.

- Import `pandas` library as `pd`.
- Read the CSV file, `datasets/disney_movies_total_gross.csv`, and assign it to the variable `gross`. Set the `parse_dates` parameter accordingly to parse the `release_date` column as date data.

Good to know

To complete this project, you should be familiar with the content in the following courses:

- Introduction to Linear Modeling in Python
- Introduction to Seaborn

Helpful link for this task:

- `read_csv()` method documentation

If you experience odd behavior, you can reset the Project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the Project will discard all code you have written so be sure to save it offline first.

-

Solution

```
# Import pandas library
import pandas as pd

# Read the file into gross
gross = pd.read_csv('./datasets/disney_movies_total_gross.csv',
parse_dates=['release_date'])

# Print out gross
gross.head()
```

Task 2: Instructions

List the top 10 movies that have earned the most at the box office.

- Sort `gross` by `inflation_adjusted_gross` in descending order using the pandas `sort_values()` method.

- Check the top ten movies using the `head()` method.

Helpful link:

- `sort_values()` method documentation
-

Solution

```
# Sort data by the adjusted gross in descending order
inflation_adjusted_gross_desc =
gross.sort_values(by='inflation_adjusted_gross', ascending=False)

# Display the top 10 movies
inflation_adjusted_gross_desc.head(10)
```

Task 3: Instructions

Add a new column for release year, then compute mean of adjusted gross per genre and per year.

- Create a new column `release_year` by extracting the attribute `year` from the `release_date` column using the pandas `DatetimeIndex()` method.
- Group movies by `genre` and by `release_year`, then compute mean on these groups.
- Use the `reset_index()` method to convert the `group` object to a DataFrame.

Helpful links:

- How to extract month and year from column in Pandas and create new column Erik Rood
- DataCamp video on the pandas `groupby()` method Categoricals and groupby
- Pandas `groupby()` method documentation
- Pandas `reset_index()` method documentation
-

Solution

```
# Extract year from release_date and store it in a new column
gross['release_year'] = pd.DatetimeIndex(gross['release_date']).year

# Compute mean of adjusted gross per genre and per year
group = gross.groupby(['genre','release_year']).mean()

# Convert the GroupBy object to a DataFrame
```

```
genre_yearly = group.reset_index()

# Inspect genre_yearly
genre_yearly.head(10)
```

Task 4: Instructions

Make a plot to see how box office revenues have changed over time.

- Import the `seaborn` library under the alias `sns`.
- Use the seaborn `relplot()` method with `kind='line'` on `genre_yearly` to make a line plot of `inflation_adjusted_gross` by `release_year`. And set the parameter `hue='genre'` to show different genres with different colors.

Helpful link:

- Seaborn `relplot()` method documentation
- Seaborn tutorial on line plots

-

Solution

```
# Import seaborn library
import seaborn as sns

# Plot the data
sns.relplot(x='release_year', y='inflation_adjusted_gross', kind='line',
hue='genre',data=genre_yearly)
```

Task 5: Instructions

Prepare dummy variables for a linear regression model.

- Use the pandas `get_dummies()` method to convert the `genre` variable into dummy variables. Set the parameter `drop_first=True` to discard one dummy variable (to avoid dependency among the variables).

Helpful links:

- The Dummy's Guide to Creating Dummy Variables tutorial
- Handling Categorical Data in Python tutorial (Under the section: One-Hot encoding)
- Pandas `get_dummies()` method documentation

-

Solution

```
# Convert genre variable to dummy variables
genre_dummies = pd.get_dummies(data=gross['genre'], drop_first=True)

# Inspect genre_dummies
genre_dummies.head()
```

Task 6: Instructions

Fit a linear regression model with `genre_dummies` and `inflation_adjusted_gross`.

- Import `LinearRegression` from `sklearn.linear_model`.
- Instantiate `LinearRegression` into a variable named `regr`.
- Fit the DataFrame`genre_dummies` and `inflation_adjusted_gross` to `regr` using the `fit()` method.
- Use `intercept_` to get the intercept term in the linear model.

Helpful links:

- DataCamp video on a linear regression [Introduction to regression]
- Scikit-learn `LinearRegression()` method [documentation]

-

Solution

```
# Import LinearRegression
from sklearn.linear_model import LinearRegression

# Build a linear regression model
regr = LinearRegression()

# Fit regr to the dataset
regr.fit(genre_dummies, gross['inflation_adjusted_gross'])

# Get estimated intercept and coefficient values
action = regr.intercept_
adventure = regr.coef_[[0]][0]

# Inspect the estimated intercept and coefficient values
print((action, adventure))
```

Task 7: Instructions

Set up an index array and initialize replicate arrays for doing pairs bootstrap.

- Use the NumPy `arange()` method to set up an array of indices named `inds` with values going from 0 to `len(gross['genre'])`.

- Use the NumPy `empty()` method to initialize two replicate arrays, named `bs_action_reps` and `bs_adventure_reps` respectively, to be of size `size`.

Helpful links:

- NumPy `arange()` method documentation
- NumPy `empty()` method documentation
- DataCamp video on pairs bootstrap [Pairs Bootstrap](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=11)

-

Solution

```
# Import a module
import numpy as np

# Create an array of indices to sample from
inds = np.arange(len(gross['genre']))

# Initialize 500 replicate arrays
size = 500
bs_action_reps = np.empty(size)
bs_adventure_reps =  np.empty(size)
```

Task 8: Instructions

Perform pairs bootstrap for linear regression.

- Use the NumPy `random.choice()` method to resample the indices `inds` of size `len(inds)`.
- Draw a sample from `inflation_adjusted_gross` using the resampled indices `bs_inds`.
- Use the pandas `get_dummies()` method to convert the `bs_genre` variable into dummy variables named `bs_dummies`.
- Store the estimated intercept from the *ith* iterate into `bs_action_reps[i]`.

Helpful link:

- NumPy `random.choice()` method documentation
- DataCamp exercise on doing pairs bootstrap A function to do pairs bootstrap

-

Solution

```
# Generate replicates
for i in range(size):

    # Resample the indices
    bs_inds = np.random.choice(inds, size=len(inds))

    # Get the sampled genre and sampled adjusted gross
    bs_genre = gross['genre'][bs_inds]
    bs_gross = gross['inflation_adjusted_gross'][bs_inds]

    # Convert sampled genre to dummy variables
    bs_dummies = pd.get_dummies(bs_genre, drop_first=True)

    # Build and fit a regression model
    regr = LinearRegression().fit(bs_dummies, bs_gross)

    # Compute replicates of estimated intercept and coefficient
    bs_action_reps[i] = regr.intercept_
    bs_adventure_reps[i] = regr.coef_[[0]][0]
```

Task 9: Instructions

Perform pairs bootstrap for linear regression.

- Use the NumPy `percentile()` method to compute the 95% confidence intervals for intercept value for `bs_action_reps`.
- Use the NumPy `percentile()` method to compute the 95% confidence intervals for coefficient value for `bs_adventure_reps`.

Helpful links:

- DataCamp exercise on computing confidence interval [Confidence interval on the rate of no-hitters]
- NumPy `percentile()` method [documentation]

-

Solution

```
# Compute 95% confidence intervals for intercept and coefficient values
confidence_interval_action = np.percentile(bs_action_reps, [2.5, 97.5])
confidence_interval_adventure = np.percentile(bs_adventure_reps, [2.5,
97.5])

# Inspect the confidence intervals
print(confidence_interval_action)
print(confidence_interval_adventure)
```

Task 10: Instructions

True or false?

- Given the confidence intervals for the intercept and coefficient, is it `True` or `False` that Disney studios should make more action and adventure movies?

Congratulations on completing the Project! If you'd like to continue building your Python skills, all of DataCamp's Python courses are listed here.

-

Solution

```
# should Disney studios make more action and adventure movies?
more_action_adventure_movies = True
```

Real-time Insights from Social Media Data

Task 1: Instructions

Load and inspect the data.

- Import the `json` module.
- Open the JSON file using the `open()` method with `'datasets/WWTrends.json'` as input parameter -> call the `read()` method on the opened file to read its content -> pass the read JSON string to the `json.loads()` method as input parameter for decoding it -> store the decoded output in `WW_trends`.
- Repeat the same steps for `'datasets/USTrends.json'` and store the output in `US_trends`.
- Inspect `WW_trends` and `US_trends` using the `print()` method.

Warning: some of the tweets in the Twitter datasets contain explicit language.

Good to know

- This Project provides the opportunity to apply the skills covered in DataCamp's Analyzing Social Media Data in Python course.
- If you are familiar with Python and basics of Pandas, you should still be able to complete this Project. It is recommended to take the course as a follow-up for complementary skills like data collection.

You might find Python's official JSON documentation helpful as well. Also, **HINTS** are always there!

Helpful links specific to this Task:

- Python Open
- Python JSON

*If you experience odd behavior **you can reset the Project** by clicking the circular arrow in the bottom-right corner of the screen. Resetting the Project will discard all code you have written so be sure to save it offline first.*

-

Solution

```
# Loading json module
import json

# Loading WW_trends and US_trends data
WW_trends = json.loads(open('datasets/WWTrends.json').read())
```

```
US_trends = json.loads(open('datasets/USTrends.json').read())

# Inspecting data by printing out WW_trends and US_trends variables
print(WW_trends)
print(US_trends)
```

Task 2: Instructions

Pretty-print the output.

- Pass the `WW_trends` object to the `json.dumps()` method, with an additional input parameter `indent` set to 1. `print` the output.
- Repeat the same for `US_trends`.

`json.dumps()` formats data as a JSON string. If you pass *'indent'* to the method (a positive integer), then all the elements in the JSON array are printed with that indent level. This makes it easy to read the results — pretty-printed.

- **Solution**

```
# Pretty-printing the results. First WW and then US trends.

print("WW trends:")
print (json.dumps(WW_trends, indent=1))

print("\n", "US trends:")
print (json.dumps(US_trends, indent=1))
```

Task 3: Instructions

Extract the names of common trends.

- Extract the name field, `trend['name']`, from the list of `trends` in `WW_trends` and `US_trends` using list comprehension(*). You can just use `WW_trends[0]['trends']`and `US_trends[0]['trends']` for iterations to get the names because the trends objects are lists with only one element.
- Call the `intersection()` method on `world_trends` with `us_trends` as input parameter to get the common items between the two; store the output in the variable called `common_trends`.

(*)List comprehension refresher: `[ expression for item in list ]`

Helpful links:

- `sets`

- `lists`
-

Solution

```
# Extracting all the WW trend names from WW_trends
world_trends = set([trend['name']
                    for trend in WW_trends[0]['trends']])

# Extracting all the US trend names from US_trends
us_trends = set([trend['name']
                     for trend in US_trends[0]['trends']])

# Let's get the intersection of the two sets of trends
common_trends = world_trends.intersection(us_trends)

# Inspecting the data
print(world_trends, "\n")
print(us_trends, "\n")
print (len(common_trends), "common trends:", common_trends)
```

Task 4: Instructions

Load and inspect the data.

- Just like in Task 1, use the`open()` method with `'datasets/WeLoveTheEarth.json'` as input parameter to open the file -> call the `read()` method on the opened file to read its content -> pass the read JSON string to the `json.loads()` method as input parameter for decoding it -> store the decoded output in `tweets`.

-

Solution

```
# Loading the data
tweets = json.loads(open('datasets/WeLoveTheEarth.json').read())

# Inspecting some tweets
tweets[0:2]
```

Task 5: Instructions

Extract texts, usernames and hashtags from the tweets.

- For each tweet in the `tweets` object , extract its text field, `tweet['text']`, using list comprehension. Store all the ouput texts in a list called `texts`.
- For each tweet in `tweets`, create an inner loop to iterate through user*mentions, `tweet['entities']['user_mentions']`. From each*

*`user_mention` extract its screen*name field, `user_mention['screen_name']`. Store the output in `names`.
- For each tweet in `tweets`, create an inner loop to iterate through hashtags, `tweet['entities']['hashtags']`. From each `hashtag` extract its text field, `hashtag['text']`. Store the output in `hashtags`.

-

Solution

```
# Extracting the text of all the tweets from the tweet object
texts = [tweet['text']
                 for tweet in tweets ]

# Extracting screen names of users tweeting about #WeLoveTheEarth
names = [user_mention['screen_name']
                 for tweet in tweets
                     for user_mention in tweet['entities']
['user_mentions']]

# Extracting all the hashtags being used when talking about this topic
hashtags = [hashtag['text']
             for tweet in tweets
                 for hashtag in tweet['entities']['hashtags']]

# Inspecting the first 10 results
print (json.dumps(texts[0:10], indent=1),"\n")
print (json.dumps(names[0:10], indent=1),"\n")
print (json.dumps(hashtags[0:10], indent=1),"\n")
```

Task 6: Instructions

Creating frequency distribution.

- Import the `Counter` module from `collections`.
- Call the `Counter()` method with `item` from the for loop as input parameter. (This allows you to keep track of how many times same values are added.)

Helpful links:

- `Counter` [documentation]

-

Solution

```
# Importing modules
```

```
from collections import Counter

# Counting occcurrences/ getting frequency dist of all names and hashtags
for item in [names, hashtags]:
    c = Counter(item)
    # Inspecting the 10 most common items in c
    print (c.most_common(10), "\n")
```

Task 7: Instructions

Extracting data for retweets.

- Get `'retweet_count'`, `'retweeted_status\favorite_count'`,`'retweeted_status\user\followers_count'`,`'retweeted_status\user\screen_name'`, and `'text'` fields for each `tweet` from the given for loop, **respecting this order**.

-

Solution

```
# Extracting useful information from retweets
retweets = [
             (tweet['retweet_count'],
              tweet['retweeted_status']['favorite_count'],
              tweet['retweeted_status']['user']['followers_count'],
              tweet['retweeted_status']['user']['screen_name'],
              tweet['text'])

            for tweet in tweets
                if 'retweeted_status' in tweet
           ]
```

Task 8: Instructions

Creating a table with insights.

- Create a DataFrame using the `pd.DataFrame()` constructor by passing `retweets` object as input. Also set the additional input parameter `columns` to `['Retweets','Favorites', 'Followers', 'ScreenName', 'Text']`.
- Call `groupby()` on the resulting DataFrame with `['ScreenName','Text','Followers']` as input parameter.
- Then call `sum()` on the results of the groupby to compute an aggregate of the numerical columns.
- Finally call `sort_values()` with input paramters `by` set to `['Followers']` and `ascending` to `False` to sort the table by decreasing number of followers.

Helpful links:

- DataFrame from a list (Stack Overflow)
- `groupby()`
- `sort_values()`

-

Solution

```
# Importing modules
import matplotlib.pyplot as plt
import pandas as pd

# Visualizing the data in a pretty and insightful format
df = pd.DataFrame(
    retweets,

columns=['Retweets','Favorites','Followers','ScreenName','Text']).groupby(
    ['ScreenName','Text','Followers']).sum().sort_values(by=['Followers'],
ascending=False)

df.style.background_gradient()
```

Task 9: Instructions

Extracting languages and plotting their frequency distribution.

- For each `tweet` object get its language field, `tweet['lang']`, and append it to the list of languages, `tweets_languages` using the `append()` method.
- Call matplotlib's `plt.hist()` method with `tweets_languages` as input parameter to plot the frequency distribution of languages.

Helpful links:

- `append()` documentation
- How to plot a histogram using Matplotlib in Python with a list of data? (Stack Overflow answer)

-

Solution

```
# Extracting language for each tweet and appending it to the list of
languages
tweets_languages = []
for tweet in tweets:
        tweets_languages.append(tweet['lang'])

# Plotting the distribution of languages
%matplotlib inline
plt.hist(tweets_languages)
```

Task 10: Instructions

Congratulations on completing the project!

- Twitter data is now all yours to explore… Just remember that **"Practice makes perfect!"**

Helpful links:

- Twitter API Reference Index
- Twitter Developers Docs
-

Solution

```
# Congratulations!
print("High Five!!!")
```

Analyze International Debt Statistics

Task 1: Instructions

Inspect the international debt data.

- Read the line of code provided for you, which connects you to the `international_debt` database.
- Select all of the columns from the `international_debt` table and limit the output to the first 10 rows.

Good to know

The only prerequisite to complete this project is familiarity with the contents covered in DataCamp's Intro to SQL for Data Science course.

SQL DataCamp projects are completed in Jupyter Notebooks. If you're not familiar with Jupyter Notebooks, that's okay! All you need to know is that you can execute SQL commands in the code cells provided, as long as you have `%%sql` at the top of them. If you'd like more info on Jupyter Notebooks, go here.

If you experience odd behavior you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written so be sure to save it offline first.

Helpful links:

- `SELECT`ing columns exercise from Intro to SQL for Data Science
- `LIMIT` exercise from Intro to SQL for Data Science
-

Solution

```
%%sql
postgresql:///international_debt

SELECT *
FROM international_debt
LIMIT 10;
```

Task 2: Instructions

Find the number of distinct countries.

- Use the `DISTINCT` clause and the `COUNT()` function in pair on the `country_name` column.
- Alias the resulting column as `total_distinct_countries`.

Jupyter Notebook trick: if you click the white area to the left of the output for this task's code cell, the output area will be collapsed and become scrollable.

Helpful links:

- `COUNT` and `DISTINCT` exercise from Intro to SQL for Data Science
- Aliasing exercise from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    COUNT(DISTINCT country_name) AS total_distinct_countries
FROM international_debt;
```

Task 3: Instructions

Extract the unique debt indicators in the table.

- Use the `DISTINCT` clause on the `indicator_code` column.
- Alias the resulting column as `distinct_debt_indicators`.
- Order the results by `distinct_debt_indicators`.

Helpful links:

- Exercise on `DISTINCT` from Intro to SQL for Data Science
- Aliasing exercise from Intro to SQL for Data Science
- Exercise on `ORDER BY` from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    DISTINCT indicator_code AS distinct_debt_indicators
FROM international_debt
ORDER BY distinct_debt_indicators;
```

Task 4: Instructions

Find out the total amount of debt as reflected in the table.

- Use the built-in `SUM` function on the `debt` column, then divide it by 1000000 and round the result to 2 decimal places so that the output is fathomable.
- Alias the resulting column as `total_debt`.

Helpful links:

- Exercise on *aggregate functions* from Intro to SQL for Data Science
- Aliasing exercise from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    ROUND(SUM(debt)/1000000, 2) AS total_debt
FROM international_debt;
```

Task 5: Instructions

Find out the country owing to the highest debt.

- Select the `country_name` and `debt` columns, then apply the `SUM` function on the `debt` column.
- Alias the column resulted from the summation as `total_debt`.
- `GROUP` the results `BY` `country_name` and `ORDER` them `BY` the new alias `total_debt` in a *descending* manner.
- `LIMIT` the number of rows to be *one*.

Helpful links:

- Exercise on *aggregate functions* from Intro to SQL for Data Science
- `GROUP BY` exercise from Intro to SQL for Data Science
- Exercise on `ORDER BY` from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    country_name,
    SUM(debt) AS total_debt
FROM international_debt
GROUP BY country_name
```

```
ORDER BY total_debt DESC
LIMIT 1;
```

Task 6: Instructions

Determine the average amount of debt owed across the categories.

- Select `indicator_code` aliased as `debt_indicator`, then select `indicator_name` and `debt`.
- Apply an aggregate function on the `debt` column to average out its values and alias it as `average_debt`.
- Group the results by the newly created `debt_indicator` and already present `indicator_name` columns.
- Sort the output with respect to the `average_debt` column in a descending manner and limit the results to *ten*.

Helpful links:

- Aggregate functions exercise from Intro to SQL for Data Science
- `GROUP BY` exercise from Intro to SQL for Data Science
- Exercise on `ORDER BY` from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    indicator_code AS debt_indicator,
    indicator_name,
    AVG(debt) AS average_debt
FROM international_debt
GROUP BY debt_indicator, indicator_name
ORDER BY average_debt DESC
LIMIT 10;
```

Task 7: Instructions

Find out the country with the highest amount of principal repayments.

- Select the `country_name` and `indicator_name` columns.
- Add a `WHERE` clause to filter out the maximum `debt` in `DT.AMT.DLXF.CD` category.

Helpful links:

- `WHERE` exercise from Intro to SQL for Data Science
- Aggregate functions exercise from Intro to SQL for Data Science
- Tutorial on writing subqueries in PostgreSQL

-

Solution

```
%%sql
SELECT
    country_name,
    indicator_name
FROM international_debt
WHERE debt = (SELECT
                 MAX(debt)
              FROM international_debt
              WHERE indicator_code='DT.AMT.DLXF.CD');
```

Task 8: Instructions

Find out the debt indicator that appears most frequently.

- Select the `indicator_code` column, then separately apply an aggregate function to count its values. Alias the column resulting from the counting as `indicator_count`.
- Group the results by `indicator_code` and order them first by the newly created `indicator_count` column then the `indicator_code` column, both in a descending manner.
- Limit the resulting number of rows to 20.

Helpful links:

- Exercise on *aggregate functions* from Intro to SQL for Data Science
- `GROUP BY` exercise from Intro to SQL for Data Science
- Exercise on `ORDER BY` from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    indicator_code,
    COUNT(indicator_code) AS indicator_count
FROM international_debt
GROUP BY indicator_code
ORDER BY indicator_count DESC, indicator_code DESC
LIMIT 20;
```

Task 9: Instructions

Find out the debt indicators for which a country owes its highest debt.

- Select the `country_name`, `indicator_code` and `debt` columns, and apply an aggregate function to take the maximum of `debt`. Alias the result as `maximum_debt`.
- Group the results by `country_name` and `indicator_code`.
- Order the results by `maximum_debt` in a descending manner.
- Limit the output to 10.

Helpful links:

- Exercise on *aggregate functions* from Intro to SQL for Data Science
- `GROUP BY` exercise from Intro to SQL for Data Science
- Exercise on `ORDER BY` from Intro to SQL for Data Science

-

Solution

```
%%sql
SELECT
    country_name,
    indicator_code,
    MAX(debt) AS maximum_debt
FROM international_debt
GROUP BY country_name, indicator_code
ORDER BY maximum_debt DESC
LIMIT 10;
```

Text Mining America's Toughest Game Show

Task 1: Instructions

Load the packages and read in the data.

- Load in the `readr`, `dplyr`, `tm`, and `wordcloud` packages.
- Use `read_csv()` to read in the dataset, `datasets/jeopardy.csv`, and assign it to `jeopardy`.

Good to know

Note: this project is soft-launched, which means you may experience bugs. Please click "Report an Issue" in the top-right corner of the screen to provide feedback.

This project lets you apply the skills from Text Mining: Bag of Words. We recommend that you are familiar with the content in that course and its prerequisites before starting this project.

Helpful links:

- `read_csv()` function documentation

If you experience odd behavior, you can reset the project by clicking the circular arrow in the bottom-right corner of the screen. Resetting the project will discard all code you have written, so be sure to save it offline first.

-

Solution

```
# Load packages
library(readr)
library(dplyr)
library(tm)
library(wordcloud)

# Load the dataset
jeopardy <- read_csv("datasets/jeopardy.csv")
```

Task 2: Instructions

Inspect the data and display the first six rows.

- Use `glimpse()` on the `jeopardy` dataset.

- Display the first six rows of `jeopardy`.

Helpful links:

- `glimpse()` documentation
- `head()` documentation

Your code output should look something like this:

```
Observations: 116,837
Variables: 7
$ show_number <dbl> 4031, 4031, 4031, 4031, 4031, 4031, 4031, 4031, 4031, 4...
$ air_date    <chr> "2/25/2002", "2/25/2002", "2/25/2002", "2/25/2002", "2/...
$ round       <chr> "Jeopardy!", "Jeopardy!", "Jeopardy!", "Jeopardy!", "Je...
$ category    <chr> "AMERICAN HISTORY", "FIREFIGHTING", "GEOGRAPH\"E\"", "G...
$ value       <chr> "$200", "$200", "$200", "$200", "$200", "$200", "$400",...
$ question    <chr> "In 1805 this territory was created from the Indiana on...
$ answer      <chr> "Michigan", "the Hall of Flame", "Etna", "Gary Burghoff...
```

A tibble: 6 x 7

show_number	air_date	round	category	value	
<dbl>	<chr>	<chr>	<chr>	<chr>	
4031	2/25/2002	Jeopardy!	AMERICAN HISTORY	$200	In 1805 this territory was created from the with all or parts of the lower & uppe
4031	2/25/2002	Jeopardy!	FIREFIGHTING	$200	The firefighting museum in Phoenix, Arizon the Hall of Fame, but t
4031	2/25/2002	Jeopardy!	GEOGRAPH"E"	$200	Sicilians call this active volcan
4031	2/25/2002	Jeopardy!	GIVE THE ROLE TO GARY	$200	TV, 1972-1979: Walter "Ra
4031	2/25/2002	Jeopardy!	WED TO THE IDEA	$200	The sacrament of marria
4031	2/25/2002	Jeopardy!	CRIMINAL CONVERSATION	$200	Ice can refer to diamonds; chill can me maybe with a lawnmower (a r

-

Solution

```
# Glimpse the dataset
glimpse(jeopardy)

# Display the first six rows
head(jeopardy, n = 6)
```

Task 3: Instructions

Create a corpus of *Jeopardy!* categories from the first round of each game.

- In the `jeopardy` dataset, filter for any row from the "Jeopardy!" round and select the `category` column. Assign it to `categories`.
- Create a vector source from `categories`, and call it `categories_source`.
- Create a volatile corpus from `categories_source` and call it `categories_corp`.

`select()` and `filter()`, are included in the `dplyr` package.

Helpful links:

- `VectorSource()` documentation
- `VCorpus()` documentation

-

Solution

```
# Create the categories variable
categories <- jeopardy %>%
  filter(round == "Jeopardy!") %>%
  select(category)

# Create a vector source
categories_source <- VectorSource(categories)

# Create a corpus from the vector source
categories_corp <- VCorpus(categories_source)
```

Task 4: Instructions

Create a clean term-document matrix from `categories`.

- Create a clean categories corpus, named `clean_corp`, with `tm_map()` by:
  - transforming all text to lowercase
  - removing punctuation
  - stripping whitespace
  - removing English stopwords
- Create a term-document matrix from `clean_corp`. Name it `categories_tdm`.

Remember, only certain transformations are compatible with `tm_map()`. `tolower()`, a base R function, cannot be passed as an argument to `tm_map()` by itself. The `content_transformer()` wrapper must be used around `tolower()` first.

Helpful links:

- `tm_map()` function documentation
- Transformations which can be used with `tm_map()`

-

Solution

```
# Clean the corpus
clean_corp <- tm_map(categories_corp, content_transformer(tolower))
clean_corp <- tm_map(clean_corp, removePunctuation)
clean_corp <- tm_map(clean_corp, stripWhitespace)
clean_corp <- tm_map(clean_corp, removeWords, stopwords("en"))

# Create a TDM from the clean corpus
categories_tdm <- TermDocumentMatrix(clean_corp)
```

Task 5: Instructions

Retrieve the frequencies of each word in the *Jeopardy!* category names, from highest to lowest.

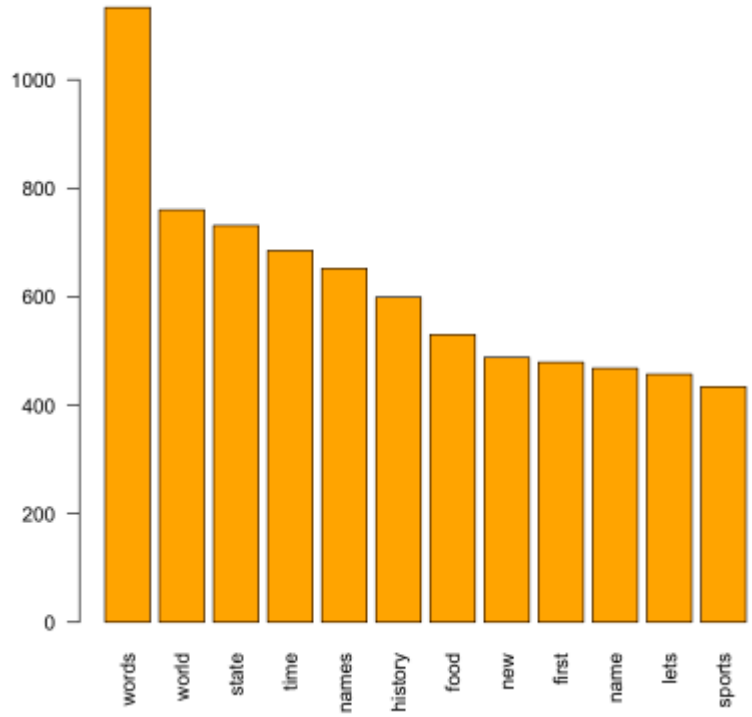
- Create a word-frequency matrix from `categories_tdm` called `categories_m`.
- Sum the values in each row using `rowSums()` and sort the sums in decreasing order. Store the sorted word sums in `term_frequency`.
- Use `barplot()` to create a visualization of the **twelve most frequent words**. Be sure to set the `las =` argument equal to `2` to rotate the x-axis labels.

Calling `rowSums()` on the matrix will return a double vector containing all of the words and their respective frequencies. Sorting the values in decreasing order will allow us to easily see which words have the highest frequency.

Helpful links:

- `rowSums()` documentation
- `barplot()` documentation

Your code output should look something like this:



-

Solution

```
# Create a matrix from the TDM
categories_m <- as.matrix(categories_tdm)

# Sum the values in each row and sort them in decreasing order
term_frequency <- sort(rowSums(categories_m), decreasing = TRUE)

# Barplot of the twelve most frequent words
barplot(term_frequency[1:12], col = "orange", las = 2)
```

Task 6: Instructions

Remove unhelpful words from the corpus and re-plot the most frequent terms.

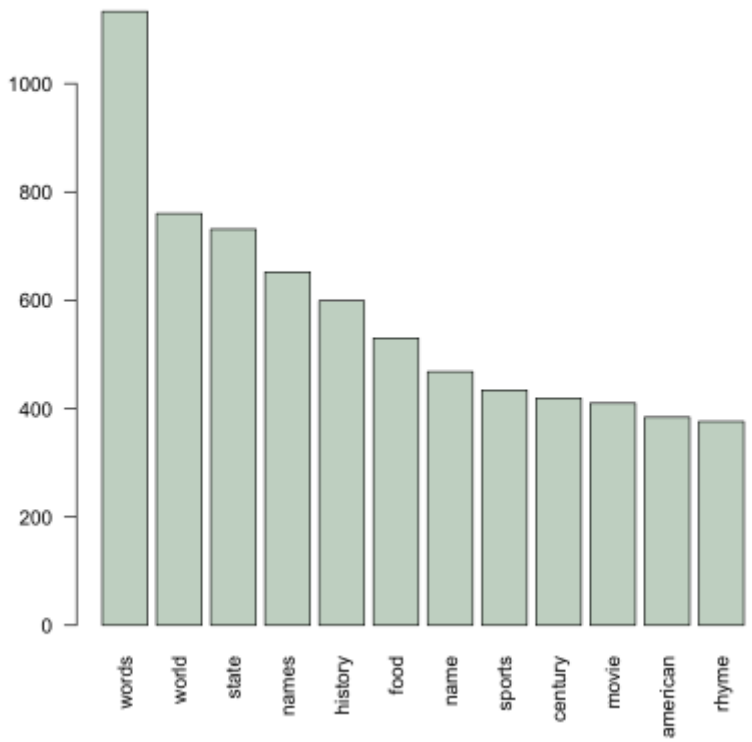
- Modify `clean_corp` to exclude the words **"time," "new," "first,"** and **"lets"** in addition to the English stopwords. Rename it `cleaner_corp`.
- Create a term-document matrix named `cleaner_tdm` from the new `cleaner_corp`.
- Refer to your code from Task 5 to create a word-frequency barplot from `cleaner_tdm`. Remember to set the `las =` argument equal to `2`.

You do not need to call `tm_map()` more than once; `removeWords()` can accept a character vector as its argument. You can use `c()` to create a character vector containing strings of all the words to be removed.

Helpful links:

- `stopwords()` documentation

Your code output should look something like this:



-

Solution

```
# Remove additional words from the corpus
cleaner_corp <- tm_map(clean_corp, removeWords,
                       c(stopwords("en"), "time", "new", "first", "lets"))
# Create a TDM
cleaner_tdm <- TermDocumentMatrix(cleaner_corp)

# Copy your code from Task 5 (change barplot colors if you want)
categories_m <- as.matrix(cleaner_tdm)
term_frequency <- sort(rowSums(categories_m), decreasing = TRUE)
barplot(term_frequency[1:12], col = "honeydew3", las = 2)
```

Task 7: Instructions

Create a single function that incorporates the cleaning functions you used earlier.

- Use `function()` to create a new function, `speed_clean()`, which cleans a corpus by transforming all text to lowercase, removes punctuation, strips whitespaces, and removes English stopwords.

Helpful links:

- `function()` function [documentation]

-

Solution

```
# Create a cleaning function
speed_clean <- function(corpus){
  corpus <- tm_map(corpus, content_transformer(tolower))
  corpus <- tm_map(corpus, stripWhitespace)
  corpus <- tm_map(corpus, removePunctuation)
  corpus <- tm_map(corpus, removeWords, stopwords("en"))
  return(corpus)
}
```

Task 8: Instructions

Create a single function that returns the word frequencies of a list in descending order.

- Create a new function `freq_terms()`, which extracts the word frequencies (in descending order) from a list. Incorporate `speed_clean()` in `freq_terms()` rather than using each cleaning function individually.

This function essentially condenses Task 3 through Task 5 into one step, but without creating the barplots.

-

Solution

```
# Create freq_terms function
freq_terms <- function(list) {
  source <- VectorSource(list)
  corpus <- VCorpus(source)
  clean_corpus <- speed_clean(corpus)
  tdm <- TermDocumentMatrix(clean_corpus)
  matrix <- as.matrix(tdm)
  term_frequency <- sort(rowSums(matrix), decreasing = TRUE)
  return(term_frequency)
}
```

Task 9: Instructions

Create a wordcloud of the most frequent words in **Final Jeopardy** answers.

- Filter `jeopardy` for any row from the "Final Jeopardy!" round and select the `answer` column. Assign it to `answers`.
- Retrieve the word-frequency vector from `answers` using the newly-created `freq_terms()`. Assign it to `ans_frequency`.
- Use `names()` to retrieve the names of `ans_frequency`. Store them in `ans_names`.
- Create a wordcloud of the most frequent words in *Jeopardy!* answers. Set the `max.words` argument to `40`.

Helpful links:

- `wordcloud()` function [documentation]

Your code output should look something like this:

-

Solution

```
# Create the answers variable
answers <- jeopardy %>%
  filter(round == "Final Jeopardy!") %>%
  select(answer)
```

```
# Retrieve word frequency
ans_frequency <- freq_terms(answers)

# Retrieve names
ans_names <- names(ans_frequency)

# Create wordcloud
wordcloud(ans_names, ans_frequency, max.words = 40,
          colors = c("wheat", "plum", "salmon"))
```

Task 10: Instructions

Answer the following question: Which textbook might be most useful when studying for *Jeopardy*?

- a. "Geography of Indonesia"
- b. "Chemistry 101"
- c. "U.S. History"
- d. "Introduction to Text Mining in R"
- Print the lowercase character containing the best answer to the question.

Congratulations on reaching the end of the project! If you'd like to continue building your R skills, all of DataCamp's R courses are listed here.

-

Solution

```
# Print the letter corresponding to the answer
print("c")
```

Exploring the Evolution of Lego (Unguided)

Welcome to the Python project Exploring the Evolution of Lego!

It may not be widely known, but Lego has had its share of ups and downs since its inception in the early 20th century. This includes a particularly rough period in the late 90s. As described in this article, Lego was only able to survive due to a successful internal brand (Bionicle) and the introduction of its first licensed series: Star Wars.

You are a Data Analyst at Lego working with the Sales/Customer Success teams. The Account Executive responsible for the Star Wars partnership has asked for specific information in preparation for their meeting with the Star Wars team. Although Star Wars was critical to the survival of the brand, Lego has since introduced a wide variety of licensed sets over subsequent years.

Your two questions are as follows:

1. What percentage of all licensed sets ever released were Star Wars themed? Save your answer as a variable `the_force` in the form of an integer (e.g. 25).

2. In which year was Star Wars not the most popular licensed theme (in terms of number of sets released that year)? Save your answer as a variable `new_era` in the form of an integer (e.g. 2012).

The method through which you approach this question is up to you, but one thing to keep in mind is that the **dataset is not necessarily clean**, and may require the removal rows where there are values missing from *critical* columns.

Hint

- You don't need to drop missing values from every column.
- You can use the method `.isin()` to filter a categorical variable using a sequence (such as a pandas Series that you can read in from `'datasets/parent_themes.csv'`).
- It will help if you create a new DataFrame that is subset to only include sets that are licensed.
- For the second question, a pivot table may be of use.

Solution

```
# Task 1: Import pandas and read in the DataFrame, and inspect it
import pandas as pd
lego_sets = pd.read_csv('datasets/lego_sets.csv')
lego_sets.info()

# Task 2: Drop relevant missing rows
lego_sets_clean = lego_sets.dropna(subset=['set_num', 'name',
'theme_name'])
lego_sets_clean.info()
```

```
# Task 3: Get list of licensed sets
parent_themes = pd.read_csv('datasets/parent_themes.csv')
licensed_themes = parent_themes[parent_themes['is_licensed']]['name']
licensed_themes.head()

# Task 4: Subset for licensed sets
licensed = lego_sets_clean['parent_theme'].isin(licensed_themes)
licensed_sets = lego_sets_clean[licensed]
licensed_sets.head()

# Task 5: Calculate the percentage of licensed sets that are Star Wars
themed
all_sets = len(licensed_sets)
star_wars_sets = licensed_sets.groupby('parent_theme').count()['set_num']
['Star Wars']
ratio = star_wars_sets/all_sets
the_force = int(ratio*100)
print(the_force)

# Task 7: Create a pivot table of sets released by theme per year
licensed_pivot = licensed_sets.pivot_table(index='year',
columns='parent_theme', values='set_num', aggfunc='count')

# Task 8: Find the year when Star Wars was not the top theme
licensed_pivot[licensed_pivot['Star Wars'] <
licensed_pivot.max(axis='columns')]
new_era = 2017
print(new_era)
```

Analyzing Password Strength in Python

Instructions

You are a data analyst working with the IT team at your company. After a recent data breach, the IT team has decided to strengthen password requirements. They've asked you to write a script to analyze the company's employees logins and identify which employees need to update their password. This will require you to use your string manipulation and regular expression skills.

Your two questions are as follows:

1. **What percentage of users have invalid passwords?** Save your answer as a variable, `bad_pass`, in the form of a float rounded up to two decimals (e.g., 0.18).
2. **Which users need to change their passwords?** Save your answer as a `pandas` Series consisting of the `usernames` *in alphabetically descending order* called `email_list`. This will be used to automate email notifications to employees.

•

Solution

```
# Importing the pandas module
import pandas as pd

# Loading in datasets/users.csv
logins = pd.read_csv("datasets/logins.csv")

# Rule 1: Not too short
# Create a boolean variable
length_check = logins['password'].str.len() >= 10
# Separate using boolean indexing
valid_pws = logins[length_check]
bad_pws = logins[~length_check]

# Rule 2: All the types of characters
# Let's create a boolean index for each character requirement
# [ ] is used to indicate a set of characters
# e.g. [abc] will match 'a', 'b', or 'c'.
# We can use a-z to represent all lowercase chars between a and Z
lcase = valid_pws['password'].str.contains('[a-z]')
ucase = valid_pws['password'].str.contains('[A-Z]')
special = valid_pws['password'].str.contains('[~!@#$%^&*()-+={}
[]|;:<>,./?]')
# /d matches any decimal digit; this is equivalent to doing [0-9]
numeric = valid_pws['password'].str.contains('\d')
# A password needs to have all these as true
# If any of these are false, we need it to return false
# In other words, all of these have to be true to return true
# We can use the & (and) operator
char_check = lcase & ucase & numeric & special
bad_pws = bad_pws.append(valid_pws[~char_check],ignore_index=True)
valid_pws = valid_pws[char_check]
```

```
# Rule 3: Must not contain the phrase password (case insensitive)
banned_phrases = valid_pws['password'].str.contains('password', case=False)
bad_pws = bad_pws.append(valid_pws[banned_phrases],ignore_index=True)
valid_pws = valid_pws[~banned_phrases]

# Rule 4: Must not contain the user's first or last name
# Extracting first and last names into their own columns
valid_pws['first_name'] = valid_pws['username'].str.extract('(^\w+)',
expand = False)
valid_pws['last_name'] = valid_pws['username'].str.extract('(\w+$)', expand
= False)
# Iterate over DataFrame rows
for i, row in valid_pws.iterrows():
    if row.first_name in row.password.lower() or row.last_name in
row.password.lower():
        valid_pws = valid_pws.drop(index=i)
        bad_pws = bad_pws.append(row,ignore_index=True)
# Note this could be done more efficiently with a lambda function

# Answering the questions
bad_pass = round(bad_pws.shape[0] / logins.shape[0], 2)
print("Percentage of users with invalid passwords", bad_pass)
email_list = bad_pws['username'].sort_values()
print(email_list)
```