Machine SurfLearning

PSTAT 131 Fall 2022 Final Project - Statistical Machine Learning
"Utilizing Machine Learning Models to Predict San Diego Surf Conditions with NOAA Historical Buoy Data"

Ezra Reyes Aguimatang

University of California, Santa Barbara

Contents

1 Introduction **3**

2 Dataset Overview **4**

2.1 Data Source 4

2.2 Observations and Variables 4

3 Research Process **5**

3.1 Response Variable 5

3.2 Expected Model 5

3.3 Expected Significant Predictors 5

3.4 Python Modules 5

4 Data Cleaning **7**

4.1 Reading Data 7

4.2 Universalizing Variables Across Datasets 9

4.3 Concatenating Data 9

4.4 Date and Time Variables 9

4.5 Dropping Variables and Handling Missing/Null Values 10

4.6 Unit Conversion 11

4.7 Final Cleaned Dataframe 11

5 Exploratory Data Analysis **13**

5.1 Checking Variable Distributions 13

5.2 Target vs. Discrete Predictors 14

5.3 Visualizing Variable Correlations 17

6 Model Building Preparation **23**

6.1 Data Splitting 23

6.2 Feature Engineering 26

6.3 What is a Modeling Pipeline? 27

7 Building Modeling Pipelines **28**

7.1 Cross Validation 29

7.2 Performance Metrics 29

8 Our Modeling Pipelines **31**

8.1 Linear Models 31

8.2 Support Vector Machines 32

8.3 K-Nearest Neighbors Methods 34

8.4 Boosted Tree Models 37

8.5 Outcomes of All Our Fitted Models 38

9 Model Awards Ceremony! **42**

9.1 "Gives a Pressing Knit Phobia" 42

9.2 "Was in the Final Submission I Guess" 42

9.3 "You Made Life so Easy" 42

9.4 "America's Next Top Model" 43

10 Who's our Best-Fitting Model? **44**

11 Final Thoughts **44**

1 Introduction

For this Machine Learning project I wanted to fit models that gave me information that I wish I had easy access to on a daily basis, something that I would use everyday and something others could potentially find useful as well. Almost every day during our Summer Breaks, at 6:30am on the dot, me and my friends routinely went for an early morning sunrise surf sessions to start our days. Whether it was projected to be amazing conditions or even if the surf was as flat as a pool, we made it an effort to at least go for an early morning drive hoping for even the possibility of some fun surf. However, on those days that one of us or all of us were too lazy to wake up to our alarms to drive 10-20 minutes and check the surf, we always had Surfline to give us a surf report and an early morning camera streaming of our break destination of the day.

Figure 1: Surfline Application Poster

For those who aren't familiar, Surfline is a popular app used among the surf community to check daily and future surf forecast and swell conditions, as well as access to live camera feeds of your local surf break of choice. However, as a subscription based service, Surfline requests a subscription fee in order for a user to access 7-day forecast reports, view camera feeds of certain popular surf breaks, and to remove a 3-time a week limit to check daily conditions. Although a great application that I still use today, sometimes it isn't the most easily accessible way to check the surf forecast 7 days in the future.

So for this Machine Learning project, I've decided to take a little inspiration from Surfline, and make my own predictions of the surf for my hometown San Diego. Although the conditions of the surf are rapidly change by the week, day, hour, and even down to the minute, I thought why not take a peek into the process behind the Surf forecasting application I've spent my early mornings checking.

2 Dataset Overview

For this project, I've decided to base my prediction of surf in San Diego on historical daily data from NOAA's National Data Buoy Center, specifically on data from the Tanner Bank Buoy that is used for predicting the conditions for San Diego...

2.1 Data Source

The source of the database used in this project will be a set of concatenated **Historical Data**. The **Historical Data Source** used will be NOAA's National Data Buoy Center.

Variables Codebook:

- YY - Year
- MM - Month
- DD - Day
- hh - Hour
- mm - Minutes
- WDIR - Wind Direction (degrees clockwise from true N)
- WSPD - Wind Speed (m/s)
- GST - Gust Speed (m/s)
- WVHT - Significant Wave Height (meters)
- DPD - Dominate Wave Period (seconds)
- APD - Average Wave Period (seconds)
- MWD - Direction of DPD (degrees clockwise from true N)
- PRES - Sea Level Pressure (hPa)
- ATMP - Air Temperature (Celsius)
- WTMP - Sea Surface Temperature (Celsius)
- DEWP - Dewpoint Temperature (Celsius)
- VIS - Station Visibility (nautical miles)
- PTDY - Pressure Tendency (plus or minus)
- TIDE - Water Level (feet)

Station ID's and Locations Used:

- 46047 - Tanner Bank (San Diego)
  - 1991, 1992, 1993, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021

2.2 Observations and Variables

From NOAA's National Data Buoy Center, I have decided to take data from Station ID 46047 which is NOAA's Tanner Banks Buoy. After scanning the station data, I found I have access to the buoy's data for the years; 1991, 1992, 1993, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, and 2021. It is important to note that in the span of roughly 3 decades, the names of variables changed, new variables were added, and some variables were comprised of mostly null values for some years or even nonexistent for other years. Based on that notion, I have decided to deselect certain variables and only read in the variables: "YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP".

3 Research Process

3.1 Response Variable

With a goal in this project of predicting the height of the surf, I have decided the response variable for this model will be `WVHT`, or "significant wave height" when referencing the data from NOAA's Historical National Data Buoy Center. Now since the goal of this project, which is to predict the next day that a specific surf break will have the "Perfect Surf", is a relative and subjective evaluation of the `WVHT` variable, I will have to decide if I want my prediction of the response to be outputted as a categorical variable (ex. "Bad", "Good", "Perfect") or numerical wave height in meters.

3.2 Expected Model

For this particular statistical machine learning project, I will be using assuming a regression model due to the presence of a majority continuous quantitative predictor variables significant to the outcome of my chosen continuous numerical response/target variable (`WVHT`).

Although the main goal of this model will be prediction, it will be a process of trial and error that requires a combination of prediction and inference. The process of inference will be very important in helping me understand my data by questioning the correlation of each predictor variable to the response `WVHT`. The early exploration of my data, testing of significance and dependence of variables, will be a crucial step in choosing a model with the most accurate prediction.

3.3 Expected Significant Predictors

Since the adequacy of surf is so specific, I do not think many of the imported variables will have neutral correlations to the response variable `WVHT`, but if I were to choose variables that I would expect to be most significant in predicting the response I would suspect:

- Date/Time Variables
- WDIR - Wind Direction (degrees clockwise from true N)
- WSPD - Wind Speed (m/s)
- DPD - Dominate Wave Period (seconds)
- APD - Average Wave Period (seconds)
- TIDE - Water Level (feet)

3.4 Python Modules

Although this project was created for UCSB's PSTAT 131 Course: Statistical Machine Learning, a class that is taught entirely in the language `R`, I have decided to try and expand my understanding of Machine Learning by completing this project in `Python`. For this project, in order to clean my data, explore my data, visualize my analysis, build and evaluate my models, and make my predictions, I have chosen to utilize the `Python` Modules below.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
import scipy as sp
import patsy as pat
import statsmodels as stmd
```

```
import random
import sklearn
```

4 Data Cleaning

Before even beginning to explore my data or predictor/target variables, I want to start with cleaning my data by looking at missing and unimportant variables/values across each year of data, concatenating/combining each year of data into one large dataframe, and possibly converting the units of some variables.

4.1 Reading Data

My first step in cleaning my data starts with reading in our `.txt` files, using `pandas.read_csv()` function, and only passing the variables "YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", and "WTMP" into our dataframes.

```
buoy_46047_1991_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_1991.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_1992_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_1992.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_1993_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_1993.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_1999_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_1999.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2000_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2000.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2001_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2001.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2002_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2002.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2003_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2003.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2004_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2004.txt", \
sep=" ", usecols=["YY", "MM", "DD", "hh", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2005_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2005.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2006_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2006.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"])

buoy_46047_2007_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2007.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2008_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2008.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])
```

```
buoy_46047_2009_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2009.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2010_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2010.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2011_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2011.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2012_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2012.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2013_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2013.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2014_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2014.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2015_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2015.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2016_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2016.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2017_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2017.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2018_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2018.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2019_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2019.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2020_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2020.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])

buoy_46047_2021_data = pd.read_csv("131_Project_Buoy_Data/46047_sandiego_2021.txt", sep=" ", \
usecols=["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "WVHT", "APD", "PRES", "ATMP", "WTMP"], \
skiprows=[1])
```

4.2 Universalizing Variables Across Datasets

After scanning the data, it's clear that for years 1991, 1992, 1993, 1999, 2000, 2001, 2002, 2003, and 2004, we do not have the variable `mm` which indicates the minute of the hour (`hh`) for that day's observation. To compensate for this we will use the function from the `pandas` module, `df.insert()`, to add a new `mm` column to the dataframe with a value of 00: assuming each observation is at the top of the hour. To verify this, we can print `df.dtypes` to make sure the column is successfully added with the correct variable type (`int`).

```
# Inserting a column for the `mm` variable
buoy_46047_1991_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_1992_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_1993_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_1999_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_2000_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_2001_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_2002_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_2003_data.insert(4, "mm", int(00), allow_duplicates=False)
buoy_46047_2004_data.insert(4, "mm", int(00), allow_duplicates=False)

# List of Dataframes for Each Year
sandiego_df_list = [buoy_46047_1991_data, buoy_46047_1992_data, \
buoy_46047_1993_data, buoy_46047_1999_data, buoy_46047_2000_data, \
buoy_46047_2001_data, buoy_46047_2002_data, buoy_46047_2003_data, \
buoy_46047_2004_data, buoy_46047_2005_data, buoy_46047_2006_data, \
buoy_46047_2007_data, buoy_46047_2008_data, buoy_46047_2009_data, \
buoy_46047_2010_data, buoy_46047_2011_data, buoy_46047_2012_data, \
buoy_46047_2013_data, buoy_46047_2014_data, buoy_46047_2015_data, \
buoy_46047_2016_data, buoy_46047_2017_data, buoy_46047_2018_data, \
buoy_46047_2019_data, buoy_46047_2020_data, buoy_46047_2021_data]
```

4.3 Concatenating Data

Now that we've successfully read in our data with the same variables for each year, I want to now create one large concatenated data set with all year of data included, using `pandas.concat()` function, in order to prepare my data for a train/test split later in my analysis.

```
# Concatenating Dataframes
sandiego_46047_concat = pd.concat(sandiego_df_list, ignore_index=True)
```

4.4 Date and Time Variables

One important thing to note in this dataset, is that we have `int` (integer) value types for the variables `YY`, `MM`, `DD`, `hh`, and `mm`, which indicate time. In order to make better use of these variables, we will insert a new column `DATE` that transforms these `int` values into a date-time format using `pandas.to_datetime()`.

```
sandiego_46047_concat['DATE'] = \
sandiego_46047_concat['MM'].apply(str) + '/' + sandiego_46047_concat['DD'].apply(str) + '/' + sandiego_

move_column = sandiego_46047_concat.pop('DATE')
sandiego_46047_concat.insert(0, 'DATE', move_column)
sandiego_46047_concat['DATE'] = pd.to_datetime(sandiego_46047_concat['DATE'], format="%m/%d/%Y")
```

4.5 Dropping Variables and Handling Missing/Null Values

Although this data set was not read in with any missing values, while scanning the data, I found that NOAA's National Data Buoy Center consistently uses values 99.0, 999.0, and 9999.0 multiple times throughout the decades of data to indicate missing values for a given observation. In order to take this into account, I have already dropped variables that were primarily comprised of these values (`GST`, `DPD`, `MWD`, `DEWP`, `VIS`, `TIDE`).

But now, since `python` cannot distinguish these numerical values as "missing" without writing some unnecessary code, we will instead simply replace them with `NaN` values using `pandas.dataframe.replace()` and `numpy.nan` to insert an `NaN` (missing/null) value.

```
# Replace Values (99.0, 999.0, 9999.0) with NaN
sandiego_46047_concat['WDIR'] = sandiego_46047_concat['WDIR'].replace(999, np.nan)
sandiego_46047_concat['WSPD'] = sandiego_46047_concat['WSPD'].replace(99.00, np.nan)
sandiego_46047_concat['WVHT'] = sandiego_46047_concat['WVHT'].replace(99.00, np.nan)
sandiego_46047_concat['APD'] = sandiego_46047_concat['APD'].replace(99.00, np.nan)
sandiego_46047_concat['PRES'] = sandiego_46047_concat['PRES'].replace(9999.00, np.nan)
sandiego_46047_concat['ATMP'] = sandiego_46047_concat['ATMP'].replace(99.00, np.nan)
sandiego_46047_concat['ATMP'] = sandiego_46047_concat['ATMP'].replace(999.00, np.nan)
sandiego_46047_concat['WTMP'] = sandiego_46047_concat['WTMP'].replace(99.00, np.nan)
sandiego_46047_concat['WTMP'] = sandiego_46047_concat['WTMP'].replace(999.00, np.nan)
```

Now since `WVHT` is our response/target variable, we want to remove any observations where `WVHT` has a missing value due to the fact that none of these observations will be very helpful in our final prediction. Therefore, we will use `pandas` function `dataframe.dropna(axis=0)` to drop any row where `WVHT` is missing or null from the dataframe. After dropping these rows, we will reset the row indices with `dataframe.reset_index()`.

```
# Drop Observations/Rows where a Variable has an NaN Value
sandiego_46047_df = sandiego_46047_concat.dropna(axis=0)

# Reset the Index Values after dropping observations
sandiego_46047_df = sandiego_46047_df.reset_index(drop=True)

# Check if any null values are still present
sandiego_46047_df.isnull().sum()
```

```
## DATE    0
## YY      0
## MM      0
## DD      0
## hh      0
## mm      0
## WDIR    0
## WSPD    0
## WVHT    0
## APD     0
## PRES    0
## ATMP    0
## WTMP    0
## dtype: int64
```

4.6 Unit Conversion

Lastly, before finalizing our dataframe before our Exploratory Data Analysis, I've decided that I want to convert the units of my response/target variable `WVHT` from meters to feet, since most surfers in San Diego look at surf height in terms of feet.

```
# Convert 'WVHT' from Meters to Feet
sandiego_46047_df.loc[:, 'WVHT'] *= 3.28084
sandiego_46047_df['WVHT'] = sandiego_46047_df['WVHT'].round(2)
```

4.7 Final Cleaned Dataframe

Now that we've finished concatenating and cleaning our data sets, let's see our final dataframe and variable object types.

Looking below at the dimensions of our final dataframe, we have ended up with 13 total variables (1 Response Variable: `WVHT`, and 12 Predictor Variables). Over the span of 3 decades of data from a singular buoy, after cleaning our data, we are left with a relatively large dataset comprised of 136,376 non-null rows/observations.

```
print(sandiego_46047_df)
```

```
##              DATE    YY  MM  DD  hh  mm  ...  WSPD   WVHT   APD    PRES  ATMP  WTMP
## 0      1991-12-04  1991  12   4  18   0  ...   5.5   4.27  7.30  1017.8  14.9  15.3
## 1      1991-12-04  1991  12   4  19   0  ...   5.9   3.61  7.00  1016.7  14.9  15.3
## 2      1991-12-04  1991  12   4  20   0  ...   5.2   3.61  7.20  1016.2  15.1  15.4
## 3      1991-12-04  1991  12   4  21   0  ...   4.8   3.61  7.10  1015.4  15.1  15.4
## 4      1991-12-04  1991  12   4  22   0  ...   4.9   3.28  7.30  1015.2  15.1  15.4
## ...           ...   ...  ..  ..  ..  ..  ...   ...    ...   ...     ...   ...   ...
## 136371 2021-12-31  2021  12  31  19  40  ...  11.4   8.63  6.24  1009.1  13.3  15.0
## 136372 2021-12-31  2021  12  31  20  40  ...  11.5   8.92  6.06  1008.7  13.0  15.0
## 136373 2021-12-31  2021  12  31  21  40  ...  12.2   8.76  6.06  1008.3  13.3  15.0
## 136374 2021-12-31  2021  12  31  22  40  ...  12.4   9.88  6.20  1008.4  13.3  14.9
## 136375 2021-12-31  2021  12  31  23  40  ...  12.9  10.30  6.26  1008.7  13.1  14.8
##
## [136376 rows x 13 columns]
```

Now in terms of datatypes, we can see from the output below, we have nearly all numerical variables data types where:

- `DATE` is of type `datetime64[ns]`
- `YY`, `MM`, `DD`, `hh`, `mm` are integer/whole number values or `int64` data types
- `WDIR`, `WSPD`, `WVHT`, `APD`, `PRES`, `ATMP`, `WTMP` are all float/decimal values or `float64` data types

```
print(sandiego_46047_df.dtypes)
```

```
## DATE    datetime64[ns]
## YY               int64
## MM               int64
## DD               int64
## hh               int64
## mm               int64
```

```
## WDIR           float64
## WSPD           float64
## WVHT           float64
## APD            float64
## PRES           float64
## ATMP           float64
## WTMP           float64
## dtype: object
```

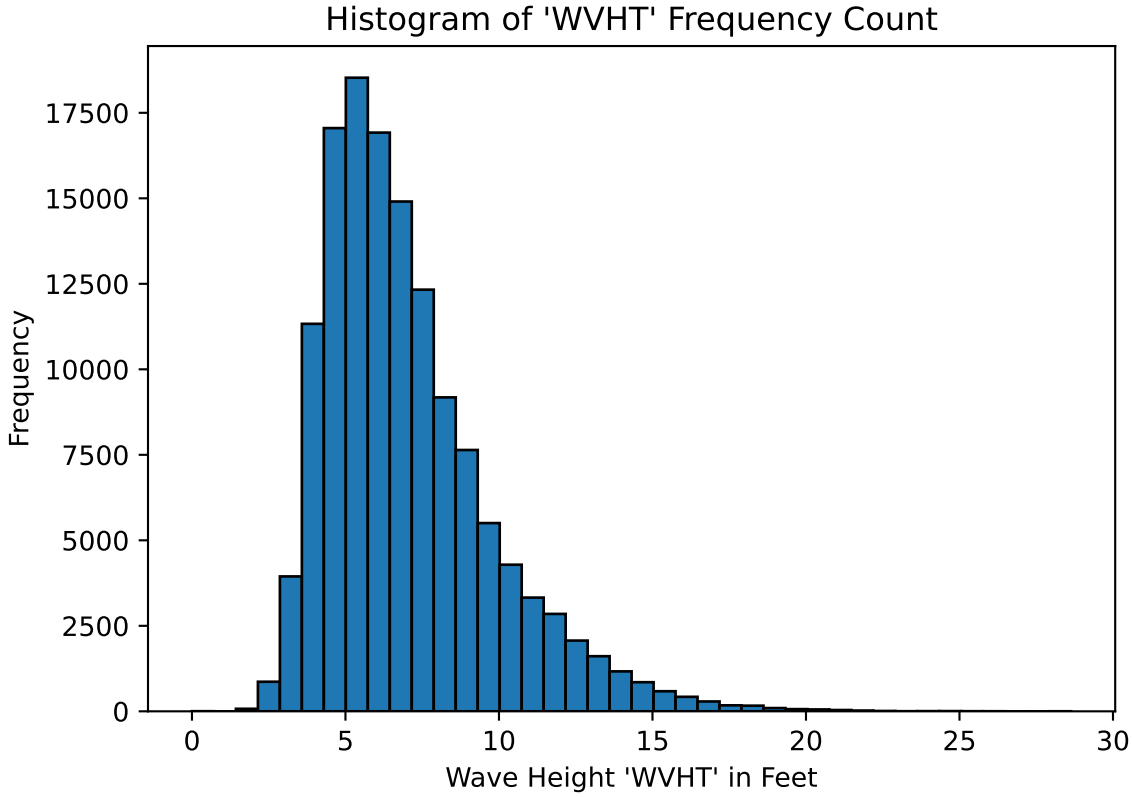
5 Exploratory Data Analysis

Now that we have our cleaned data, before we start splitting our dataframe for training and testing, let's first take some time to learn more about our data. In order to avoid any later errors in our predictions, we want to verify our understanding of how our response variable `WVHT` is influenced by our chosen predictor variables, and how our predictor variables influence one another.

5.1 Checking Variable Distributions

Firstly, to understand our response variable `WVHT` a little more, we will start by looking at its distribution using a histogram visualization.

```
plt.hist(sandiego_46047_df["WVHT"], bins=40, edgecolor="black")
plt.title("Histogram of 'WVHT' Frequency Count")
plt.xlabel("Wave Height 'WVHT' in Feet")
plt.ylabel("Frequency")
plt.show()
```



Visually, we can see we have a very strong right-skewness of `WVHT` which we must note as a possible cause of concern, since a non-normally distributed response variable can create some less reliable predictions later down the road.

To verify the visual interpretation of this distribution, we will use `python`'s `pandas` module to compute the skewness test for normality with the `df.skew()` function for `WVHT` as well as all predictor variables.

```
sandiego_46047_df.skew(axis=0, numeric_only=True)
```

```
## YY     -0.042559
## MM     -0.091213
## DD      0.007739
## hh     -0.005662
## mm      0.376884
## WDIR   -2.585497
## WSPD    0.273188
## WVHT    1.264860
## APD     1.010199
## PRES    0.293388
## ATMP    0.326886
## WTMP    0.391420
## dtype: float64
```

With this function, values that are not close to zero are not normally distributed. As can be seen, we have very skewed variables. `WVHT` outputs a value of 1.241757 which indicates a very strong right skewed distribution, and `WDIR` outputs a value of -2.622121, indicating a very strong left skew. `APD` also presents a fairly moderate/severe right skew with a value of 0.992836, while all other predictor variables appear to be approximately symmetric.

5.2 Target vs. Discrete Predictors

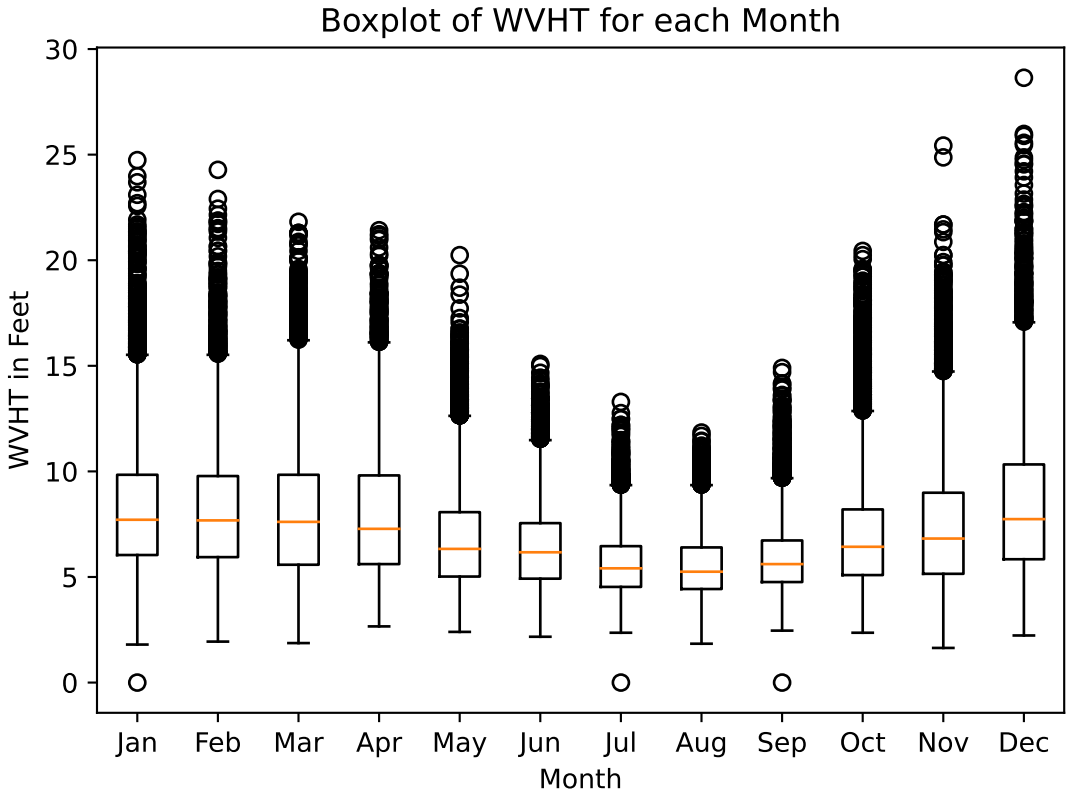
Now that we have found clear evidence of a non-normally distributed response variable, we take another step towards understanding `WVHT` by utilizing box-plots to view `WVHT` responding to our `int` data type predictors, a.k.a. whole number/integer values.

Knowing that seasons and the time of year are well known indicators of surf, we will start with a box plot of wave height (`WVHT`) for each month (`MM`) over the course of the 3 decades recorded in the data.

```
SD_jan_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 1]
SD_feb_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 2]
SD_mar_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 3]
SD_apr_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 4]
SD_may_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 5]
SD_jun_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 6]
SD_jul_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 7]
SD_aug_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 8]
SD_sep_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 9]
SD_oct_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 10]
SD_nov_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 11]
SD_dec_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['MM'] == 12]

plt.boxplot([SD_jan_wvht, SD_feb_wvht, SD_mar_wvht, SD_apr_wvht, SD_may_wvht, \
SD_jun_wvht, SD_jul_wvht, SD_aug_wvht, SD_sep_wvht, SD_oct_wvht, SD_nov_wvht, \
SD_dec_wvht])
```

```
plt.xticks(list(range(1, 13)), ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", \
"Aug", "Sep", "Oct", "Nov", "Dec"])
```

```
plt.title("Boxplot of WVHT for each Month")
plt.xlabel("Month")
plt.ylabel("WVHT in Feet")
plt.show()
```



As expected, it seems like `WVHT`'s distribution for the mean, the interquartile range, and maximum/minimum values are not consistent for each month of data. With this we can assume that significant wave height `WVHT` is most likely dependent on the month `MM` of the year.

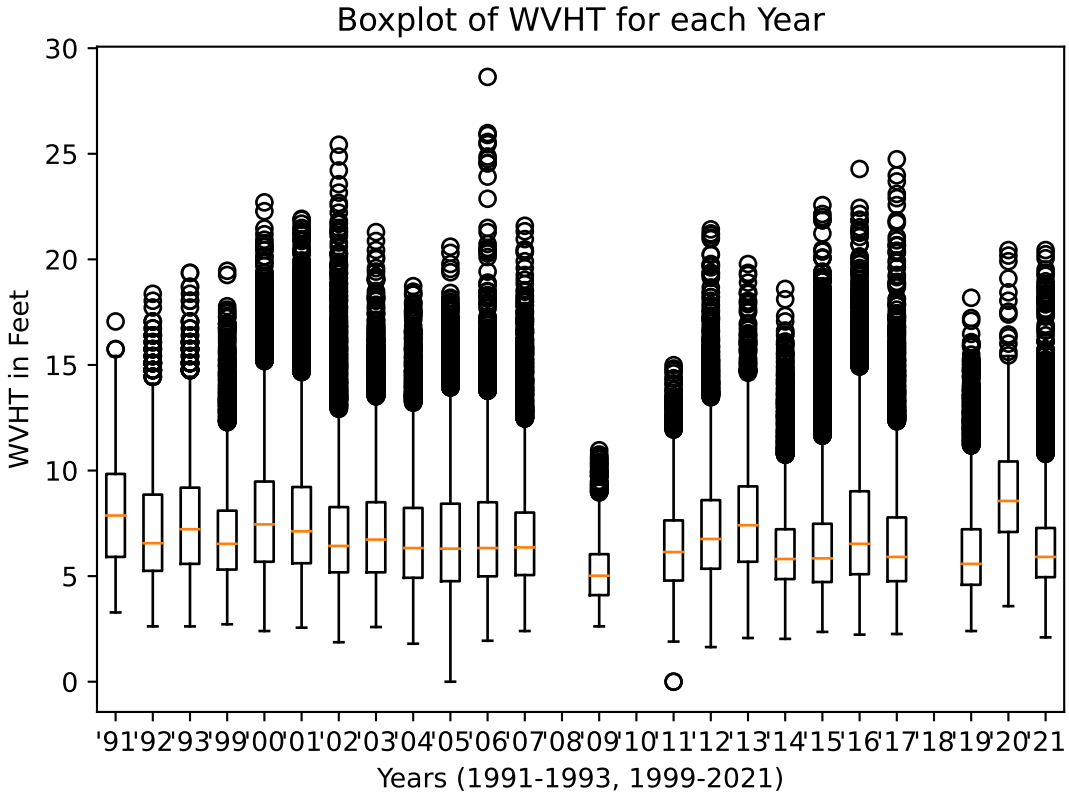
Following this result, let's create another box-plot to check and see if we have any clear visible trends in our significant wave height over the course of the 3 decades of data by visualizing `WVHT` per `YY`.

```
SD_1991_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 1991]
SD_1992_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 1992]
SD_1993_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 1993]
SD_1999_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 1999]
SD_2000_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2000]
SD_2001_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2001]
SD_2002_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2002]
SD_2003_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2003]
SD_2004_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2004]
SD_2005_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2005]
SD_2006_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2006]
SD_2007_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2007]
SD_2008_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2008]
SD_2009_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2009]
```

```
SD_2010_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2010]
SD_2011_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2011]
SD_2012_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2012]
SD_2013_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2013]
SD_2014_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2014]
SD_2015_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2015]
SD_2016_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2016]
SD_2017_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2017]
SD_2018_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2018]
SD_2019_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2019]
SD_2020_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2020]
SD_2021_wvht = sandiego_46047_df["WVHT"].loc[sandiego_46047_df['YY'] == 2021]

plt.boxplot([SD_1991_wvht, SD_1992_wvht, SD_1993_wvht, SD_1999_wvht, SD_2000_wvht, \
SD_2001_wvht, SD_2002_wvht, SD_2003_wvht, SD_2004_wvht, SD_2005_wvht, SD_2006_wvht, \
SD_2007_wvht, SD_2008_wvht, SD_2009_wvht, SD_2010_wvht, SD_2011_wvht, SD_2012_wvht, \
SD_2013_wvht, SD_2014_wvht, SD_2015_wvht, SD_2016_wvht, SD_2017_wvht, SD_2018_wvht, \
SD_2019_wvht, SD_2020_wvht, SD_2021_wvht])
```

```
plt.xticks(list(range(1, 27)), ["'91", "'92", "'93", "'99", "'00", "'01", "'02", \
"'03", "'04", "'05", "'06", "'07", "'08", "'09", "'10", "'11", "'12", "'13", "'14", \
"'15", "'16", "'17", "'18", "'19", "'20", "'21"])
```

```
plt.title("Boxplot of WVHT for each Year")
plt.xlabel("Years (1991-1993, 1999-2021)")
plt.ylabel("WVHT in Feet")
plt.show()
```



From a visual standpoint, in comparison to our box-plot of `WVHT` each `MM`, there does not seem to be a notable trend in `WVHT` each `YY` from 1991 - 2021. From this box-plot alone, we can make a general assumption that our response variable `WVHT` is most likely dependent of `YY` in near future years.

5.3 Visualizing Variable Correlations

Now that we've taken some visual interpretations of `WVHT` responding to predictors, we'll next look at some numerical approaches to evaluating the interaction of our target variable and its predictors.

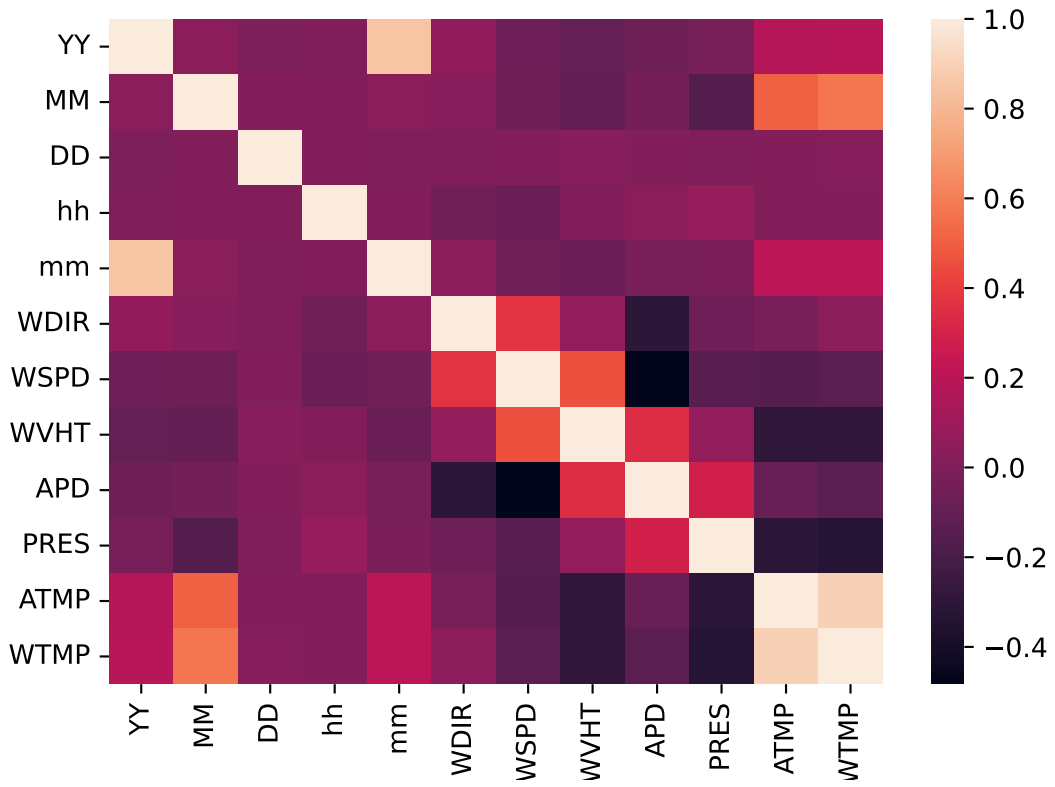
Firstly, let us take a look at the correlation between our variables using a correlation matrix and correlation heat map plot.

```
sandiego_46047_df.corr()
```

```
##             YY        MM        DD  ...      PRES      ATMP      WTMP
## YY    1.000000  0.034458 -0.008364  ... -0.036881  0.178512  0.183727
## MM    0.034458  1.000000  0.009075  ... -0.159161  0.506643  0.571917
## DD   -0.008364  0.009075  1.000000  ... -0.006277  0.014276  0.015546
## hh   -0.004173  0.002211 -0.000308  ...  0.077469 -0.000728  0.004882
## mm    0.847007  0.030399 -0.003074  ... -0.016789  0.203762  0.205610
## WDIR  0.057336  0.017613 -0.004659  ... -0.061663 -0.030407  0.040658
## WSPD -0.061029 -0.066858  0.007826  ... -0.139959 -0.154017 -0.131019
## WVHT -0.099283 -0.101492  0.021910  ...  0.062353 -0.294825 -0.294693
## APD  -0.071221 -0.038713  0.008739  ...  0.285582 -0.087847 -0.130362
## PRES -0.036881 -0.159161 -0.006277  ...  1.000000 -0.313032 -0.333352
```

```
## ATMP  0.178512  0.506643  0.014276  ... -0.313032  1.000000  0.893585
## WTMP  0.183727  0.571917  0.015546  ... -0.333352  0.893585  1.000000
##
## [12 rows x 12 columns]
```

```
corrplot = sb.heatmap(sandiego_46047_df.corr())
plt.show()
```



Noting that, a correlation coefficient in the range:

- $(-1, -0.5)$ indicates a strong negative correlation,
- $(-0.5, 0.5)$ indicates a non-existent or weak negative/positive correlation,
- $(0.5, 1)$ indicates a strong positive correlation,

we take a look at our color coded correlation heat map above to see the correlation between our variables.

When focusing on the response variable `WVHT`, we can see that a majority of the correlation plots with predictors take on a purple/pink hue which indicate essentially zero significant correlation. However, the most notable correlations between `WVHT` and predictors can be seen with `ATMP` and `WTMP` which take on a dark purple/black hue (indicating a slight negative correlation), and the correlations of `WVHT` with `WSPD` and `APD` which display an orange/red hue (indicating a slight positive correlation).

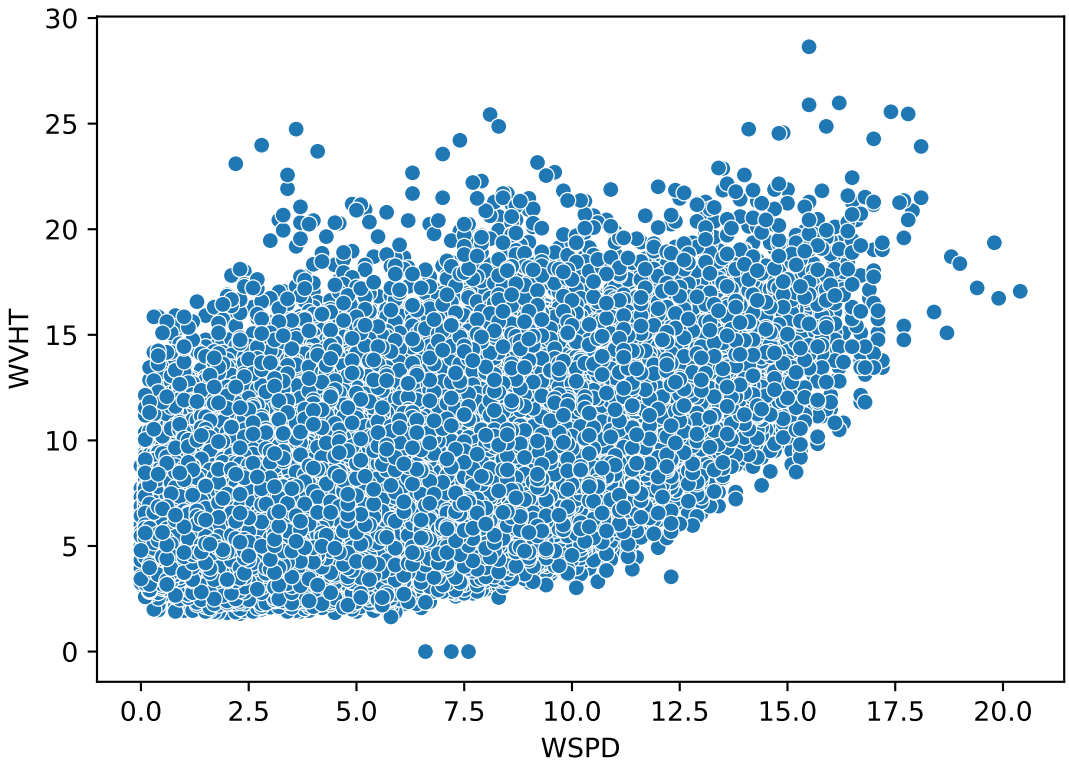
Taking a closer look at these correlations, we'll look at the correlation coefficient values for `WVHT` with each of these 4 predictor variables, as well as their scatter plots to get a better visual representation of their spread, bias, and correlations.

For WSPD we have a correlation coefficient of ≈ 0.4609707914850592, a trend that can be corroborated by the scatter plot below.

```
sandiego_46047_df['WSPD'].corr(sandiego_46047_df['WVHT'])
```

```
## 0.4609707914850592
```

```
wspd_scatter = sb.scatterplot(x = sandiego_46047_df['WSPD'], y = sandiego_46047_df['WVHT'])
wspd_scatter
```

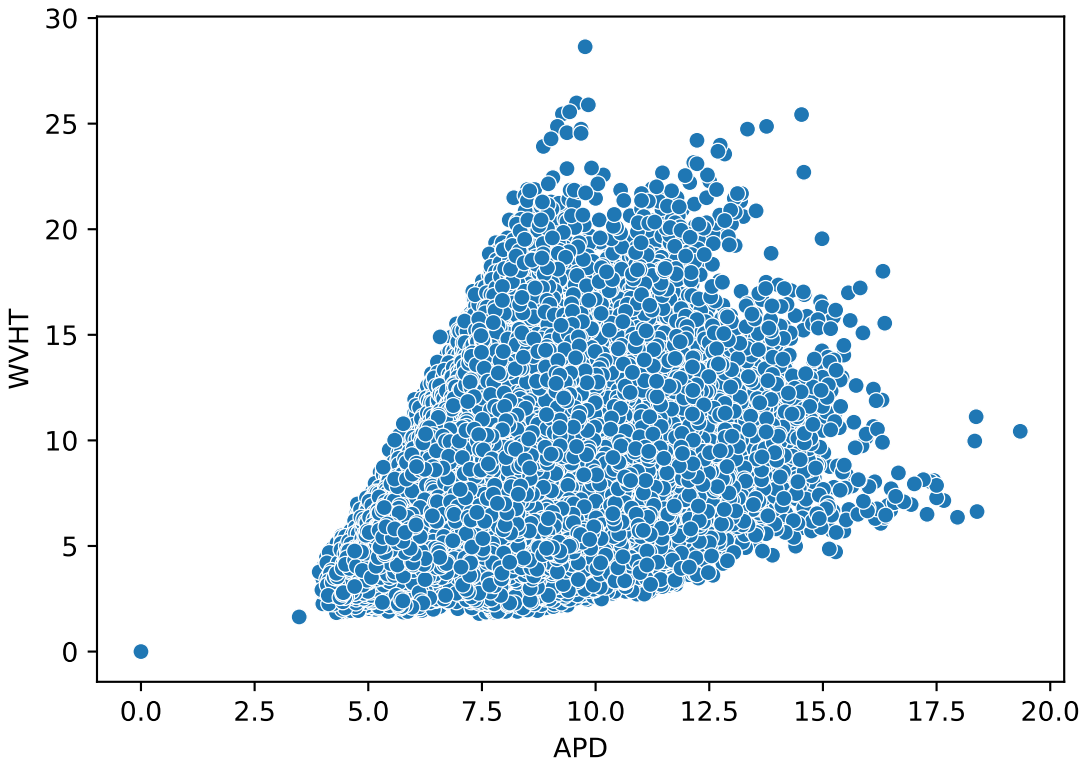


For `APD` we have a correlation coefficient of $\approx 0.3401179783730226$, a trend that can be corroborated by the scatter plot below.

```
sandiego_46047_df['APD'].corr(sandiego_46047_df['WVHT'])
```

```
## 0.3401179783730226
```

```
apd_scatter = sb.scatterplot(x = sandiego_46047_df['APD'], y = sandiego_46047_df['WVHT'])
apd_scatter
```

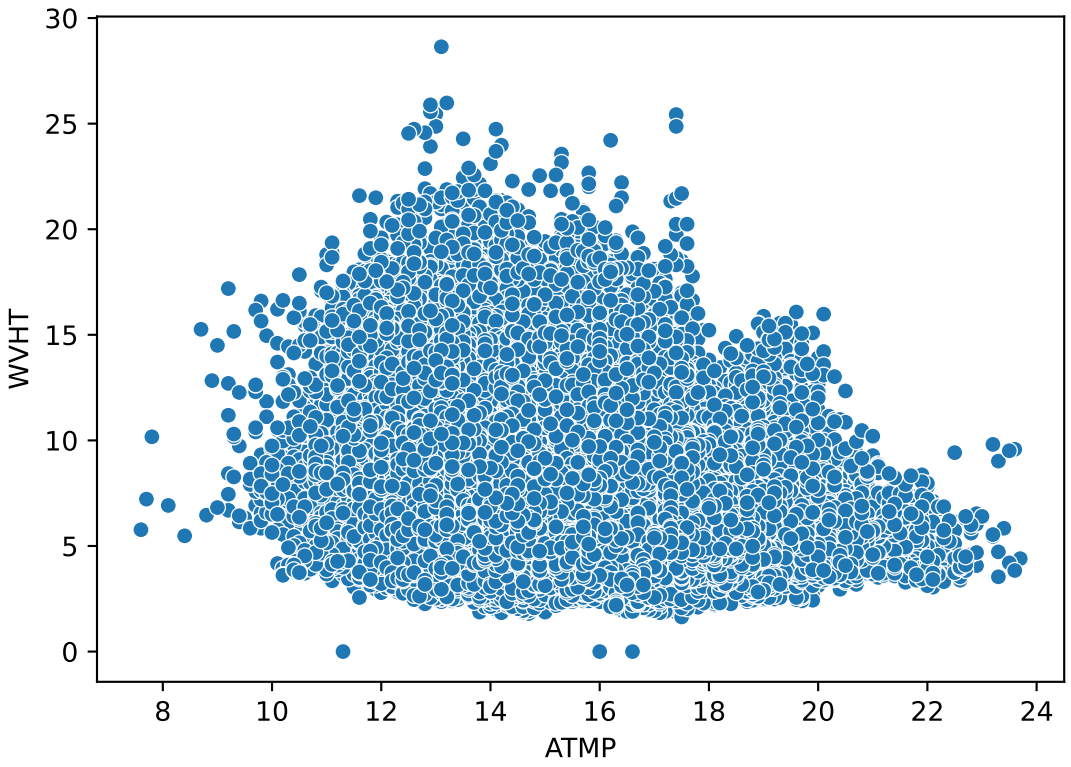


For ATMP we have a correlation coefficient of $\approx -0.2948253489768376$, a trend that can be corroborated by the scatter plot below.

```
sandiego_46047_df['ATMP'].corr(sandiego_46047_df['WVHT'])
```

```
## -0.2948253489768376
```

```
atmp_scatter = sb.scatterplot(x = sandiego_46047_df['ATMP'], y = sandiego_46047_df['WVHT'])
atmp_scatter
```

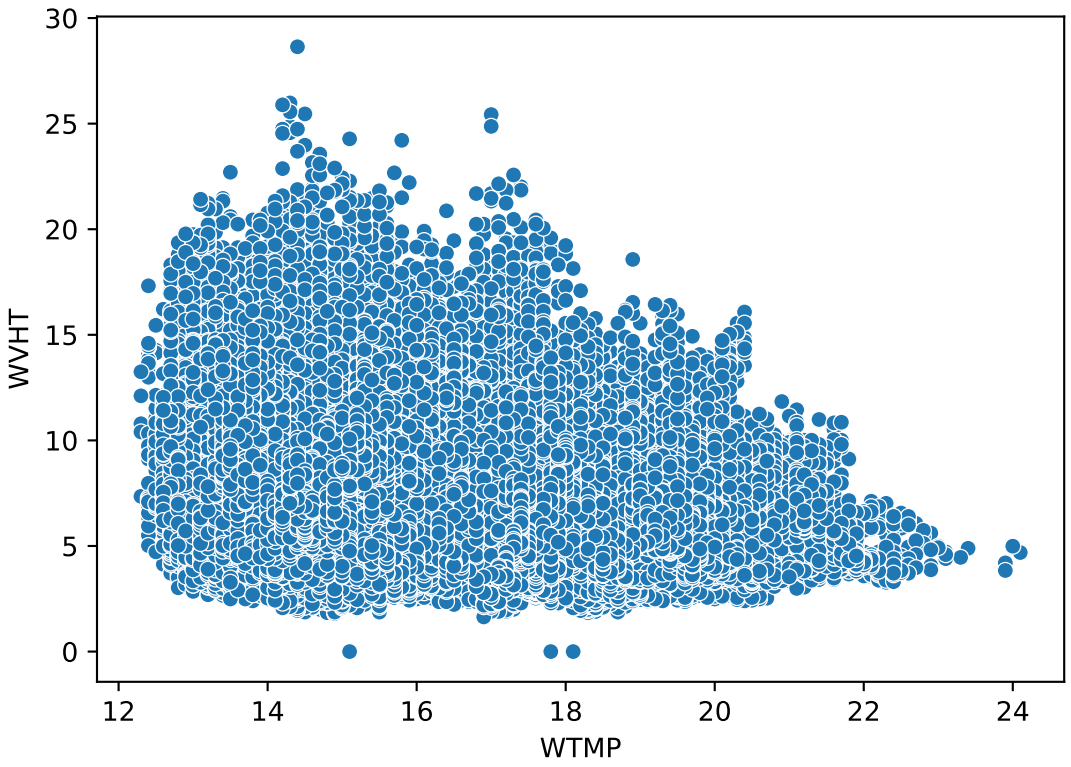


For WTMP we have a correlation coefficient of $\approx -0.2946934397448771$, a trend that can be corroborated by the scatter plot below.

```
sandiego_46047_df['WTMP'].corr(sandiego_46047_df['WVHT'])
```

```
## -0.2946934397448771
```

```
wtmp_scatter = sb.scatterplot(x = sandiego_46047_df['WTMP'], y = sandiego_46047_df['WVHT'])
wtmp_scatter
```



6 Model Building Preparation

For the purposes of translating my work in terms of this course, it should be noted that there are many differences in the procedure of preparing our data for model building and selection when using `Python` versus when using `R`. In `Python`, many of the processes handled by `R`, with functions such as `recipe()` and piped `step_()` operators, are not so easily replicable and neatly written with just a few lines of code. Hence, this step of our project is slightly longer than would be in `R`. In this project, data pre-processing and model building will be done using the methods within the `scikitlearn`/`sklearn` module.

Although having many similarities to `R`, I will be following the general `Python` workflow of model building preparation below:

1. Data Splitting
    - Splitting the Dataframe into Training/Testing Sets
    - (where the process of stratified splitting differs from `R`)
2. Feature Engineering
    - Defining Data Preprocessing and Data Transformation Techniques for Each Model
    - (similar to `recipe()` in `R` except much more tedious step by step process due to a lack of a `Python` equivalent)
3. Modeling Pipeline
    - Stratified K-Folds of our Training Data for Parameter Tuning
    - (rather than folding first and specifying model fitting expectations with a workflow and a recipe like in `R`, in `Python` we can more simply do all the above steps simultaneously with `sklearn.pipeline` with just a few lines of code)

6.1 Data Splitting

First off, let's clarify the reason for splitting our data prior to Pre-processing and Data Transformations. When setting up **Supervised** Machine Learning workflows to build and test data on multiple models, it is important we separate data to avoid **over-fitting**: building a prediction model that falsely assumes future unknown values will look exactly like the data we already know. The root of this problem of over-fitting, comes from workflow setup mistakes that don't take measures to prevent **data leakage**. Hence, the first step in Model Building Workflow Preparation to avoid data leakage, is properly splitting our data into a training subset (to train/build/learn our prediction model) and testing subset (to treat as unknown data to test our prediction accuracy).

Now let's outline the steps we will take to properly split our data in this project. But first, as stated previously, it must be noted that the process of properly splitting our data by stratifying our response variable is slightly different than done in `R`:

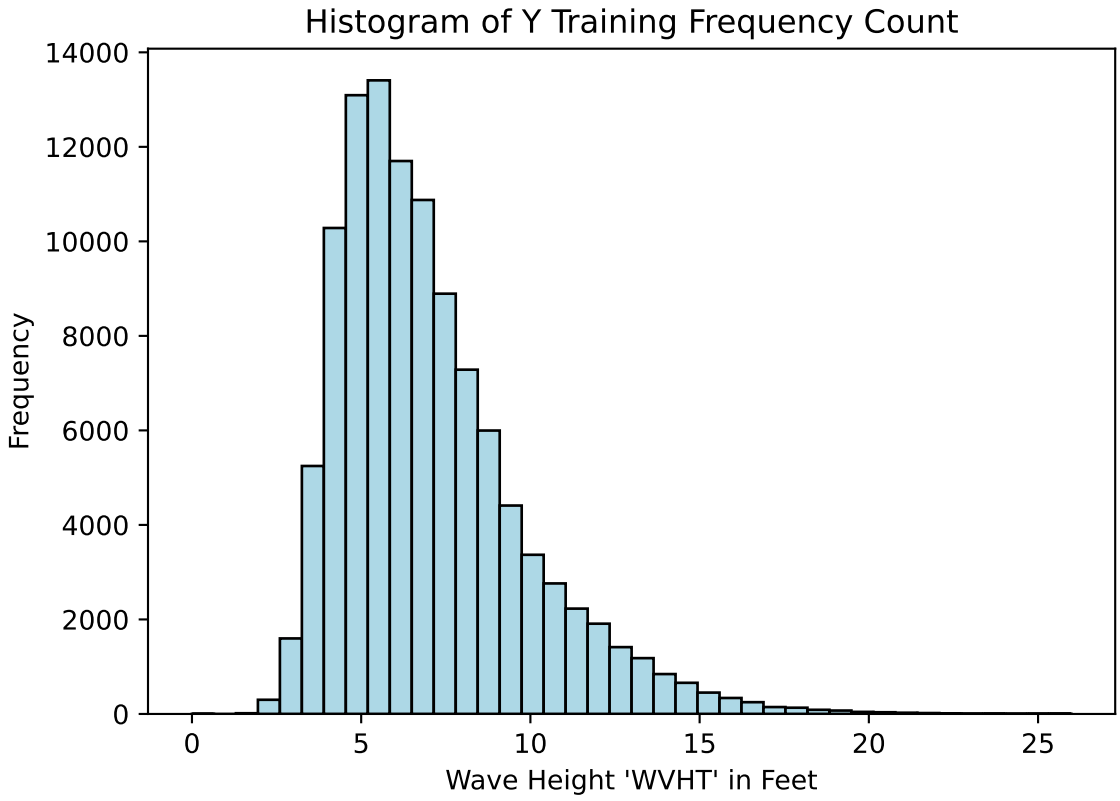
1. Firstly, similar to `R`, we need to set our random seed value to allow us to have a fixed randomized sample value to call back and reproduce the same random sample of our data across runs of our model(s), which will ultimately increase the stability and reliability of our model accuracy results.

2. Now to split our data, we have an equivalent to `R`'s `initial_split(df, prop=proportion, strata="response_variable")` using `sklearn.train_test_split(X, y, test_size=proportion, stratify=y)`. However, rather than being able to easily stratify our split by the continuous response/target variable `WVHT`, a functionality that `sklearn` does not currently have, we have to trick the function into seeing a categorical variable by binning our continuous response variable (commonly called discretization).

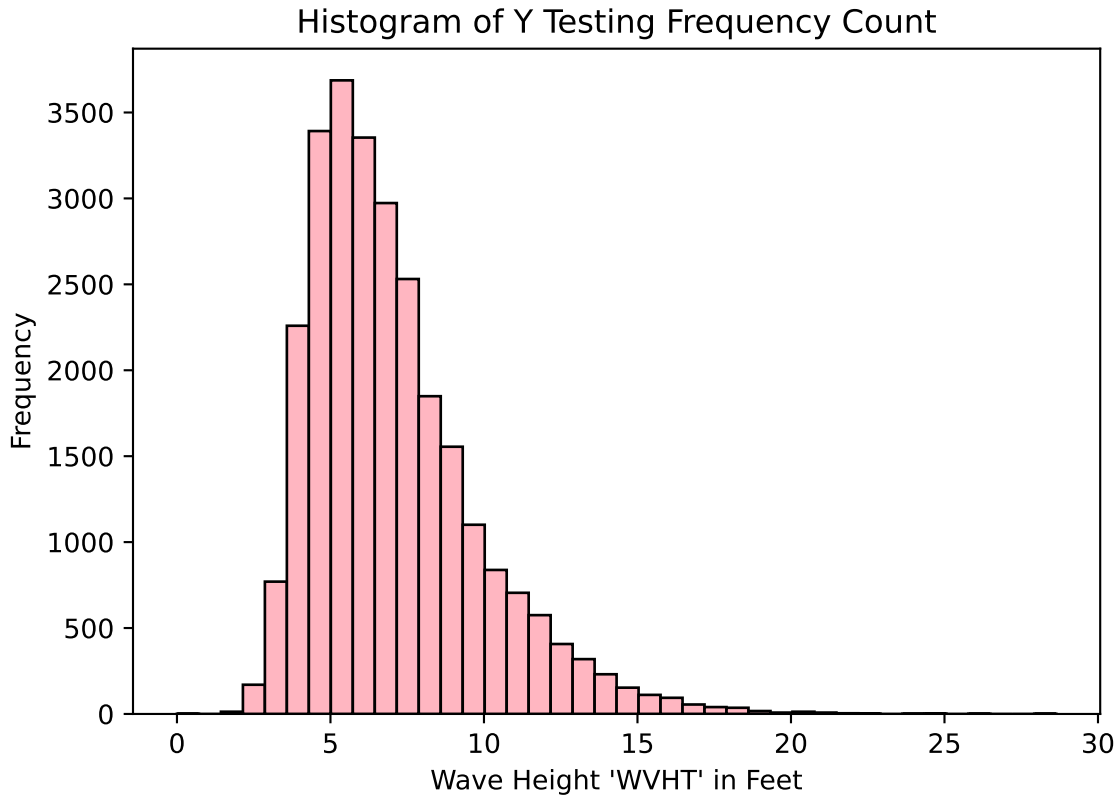
```
from sklearn.model_selection import train_test_split

# X = sandiego_46047_df[["YY", "MM", "DD", "hh", "mm", "WDIR", "WSPD", "APD", "PRES", "ATMP", "WTMP"]]
X = sandiego_46047_df.drop(['WVHT', 'DATE'], axis=1, inplace=False)
y = sandiego_46047_df["WVHT"]

# stratified_bins = np.linspace(0, 136376, 4)
# wvht_binned = np.digitize(y, stratified_bins)

seed_val = 42
random.seed(seed_val)
np.random.seed(seed_val)

X_sdsurf_train, X_sdsurf_test, y_sdsurf_train, y_sdsurf_test = \
train_test_split(X, y, test_size=0.2, random_state=42)
```

Now let's make sure that we have a stratified split of our target/response variable `WVHT`, the same proportion of wave heights in both sets, which we can verify by checking the distribution of both the training and testing splits with a histogram. If they have the same shape/distribution, we have successfully created a stratified split.

```
plt.hist(y_sdsurf_train, bins=40, color="lightblue", edgecolor="black")
plt.title("Histogram of Y Training Frequency Count")
plt.xlabel("Wave Height 'WVHT' in Feet")
plt.ylabel("Frequency")
plt.show()
```




```
plt.hist(y_sdsurf_test, bins=40, color="lightpink", edgecolor="black")
plt.title("Histogram of Y Testing Frequency Count")
plt.xlabel("Wave Height 'WVHT' in Feet")
plt.ylabel("Frequency")
plt.show()
```



After successfully splitting our data into a training and testing set, as seen above, we will check to make sure our sets of training and testing data have an accurate percentage of observations in each.

```
print("Dimensions of our Predictors Training Split Dataframe: ", \
X_sdsurf_train.shape, "\nDimensions of our Predictors Testing Split Dataframe: ", \
X_sdsurf_test.shape, "\nDimensions of our Response Training Split Dataframe: ", \
y_sdsurf_train.shape, "\nDimensions of our Response Testing Split Dataframe: ", \
y_sdsurf_test.shape)
```

```
## Dimensions of our Predictors Training Split Dataframe:  (109100, 11)
## Dimensions of our Predictors Testing Split Dataframe:  (27276, 11)
## Dimensions of our Response Training Split Dataframe:  (109100,)
## Dimensions of our Response Testing Split Dataframe:  (27276,)
```

As we can see above, we have successfully created a split of our 136,376 observations of data using 80% for our training data (109,100 observations) and 20% for our testing data (27,276 observations). By using 80% of our data for training, we allow ourselves to still use a large amount of our data to create an accurate and reliable model, without the concern of over-fitting that can happen using 100% of our data for training a model.

6.2 Feature Engineering

After data splitting, the next this step of our model building preparation involves **Feature Engineering** (or feature extraction). A very important step that is susceptible to data leakage, feature engineering refers to manipulating the features of our data using **Data Pre-Processing** and **Data Transformation** techniques.

Thinking of our data prior to feature engineering as a rough draft of an essay, we want to revise or change our data so that we may have a final draft of our data set that is complete and has variables with correct values and distributions. Although the steps that are applied vary depending on a Machine Learning Project's Data, the general order of techniques that should be considered for any problem are as follows:

- Data Pre-Processing and Data transformations:
    1. ***Choosing Important Predictors***: deciding which predictors to continue with before model building

    2. ***Missing Value Handling***: deciding whether or not to keep or drop our previously explored missing values, or to use **imputation** to fill in kept missing values

    3. ***Transformations***
        - *Polynomial Features*: creating **interactions** for features/predictors that have non-linear or dependent relationships with the target/response variable or other predictors
        - *Continuous Features*: deciding if continuous variables need to categorized into numerical splits using **discretization** or **binarization**

    4. ***Scaling***
        - *Standardization*:
        - *Normalization*:

As I've briefly mentioned in a previous section (Model Building Preparation), we can define these processes neatly using a **Modeling Pipeline**, which will be explained in the next section.

6.3 What is a Modeling Pipeline?

Dissimilar to `R`, `Python` does not have a `workflow()` function, but it does have a similar method of **Pipelines**. A pipeline can be explained as a sort of guidelines that are written as a sequence of rules for data preprocessing, modeling building, transformations, and prediction metrics. Multiple pipelines can be made with different "rules" depending on the nature of the training data, model being fit, and more.

The pipeline created from this process is what `sklearn` calls a **Modeling Pipeline**. In `R`, after splitting and pre-processing our data, one normally follows a step by step procedure that folds their training data, sets up a model and workflow, creates a parameter grid, tunes a model, and selects parameters based on performance metrics. However in `Python`, the `sklearn` module allows us to do this all simultaneously and a little quicker and cleaner with the `sklearn.pipeline` class's functionality. Within each pipeline, created using `sklearn.pipeline.Pipeline()` or `sklearn.pipeline.make_pipeline()`, we not only can we apply the aforementioned procedures, but we can also evaluate our pipeline using a chosen **Re-sampling** and/or **Cross-Validation** procedure to again address the persisting issue of data leakage and reproducibility.

The advantage of using this pipeline workflow is that as a user of Machine Learning, we can evaluate the pipelines directly. This means are able to focus solely on the outcome and accuracy of a pipeline as opposed to; what specific "rules" `Pipeline()` chooses to use for data pre-processing and transformation, and/or which configuration of hyperparameters from a grid search it chooses.

7 Building Modeling Pipelines

Now with our understanding of Modeling Pipeline workflows, lets build our pipelines! We'll be using the following modules and classes from the `sklearn`

```
from numpy import (
  mean,
  absolute,
  sqrt,
  logspace
)

from sklearn.model_selection import (
  train_test_split,
  KFold,
  RepeatedKFold,
  RepeatedStratifiedKFold,
  cross_val_score,
  GridSearchCV
  )
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest
from sklearn.preprocessing import (
  MinMaxScaler,
  KBinsDiscretizer,
  StandardScaler
  )
from sklearn.linear_model import (
  LinearRegression,
  Ridge,
  RidgeCV,
  Lasso,
  ElasticNet
  )
from sklearn.svm import (
  LinearSVR,
  NuSVR
  )
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import (
  HistGradientBoostingRegressor,
  RandomForestRegressor
  )
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import (
  mean_absolute_error,
  r2_score,
  mean_squared_error
)
from sklearn.pipeline import (
  Pipeline,
  FeatureUnion
  )
```

7.1 Cross Validation

For this project, of our models will mostly be using a Repeated Stratified K-Fold Cross-Validation with 5 folds of the training data and 5 repeats.

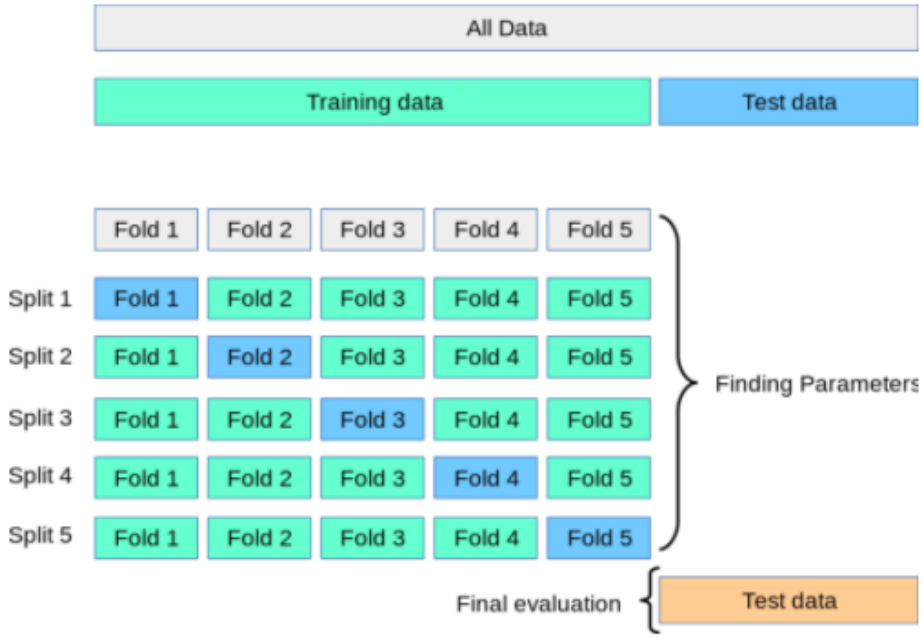


Figure 2: K-Fold Visual Explanation

According to `sklearn`, Repeated Stratified K-Fold Cross-Validation is not recommended for large data sets where observations exceed 10,000 observations, which is the case for this project's data set, due to the fact that this amount of data are more than sufficiently cross validated with the simpler automated methods such as Leave-One-Out or regular K-Fold. On another note, unlike classification problems that require the use of the `RepeatedStratifiedKFold()` method for repeated stratified k-folds, regression problems (where the target/response variable is a continuous data type) are automatically create repeated stratified k-folds with the `RepeatedKFold()` method. Depending on if the model is linear, strictly for regression, is fitting a large data set (observations $>$ 10,000), has an automated preferred CV strategy; `sklearn` will usually default to supporting a simple `KFold()` cross-validation.

7.2 Performance Metrics

Before discussing the metrics we will be using in this project, we'll first discuss the importance of these metrics and the difference in metrics between Regression and Classification problems. **Performance Metrics** are very important values that we use to numerically represent how effectively our assumed models fit our training data and how our tuned models predict the held out test data points of our response/target.

Between Classification problems and Regression problems, we use very different metrics to evaluate our models' effectiveness and accuracy:

1. Classification Metrics:
    - **Confusion Matrices**
        - a table of the combinations of predicted and actual values (from the test data set) where the response/target variable is binary or multi-class

- **Log-Loss**
  - a log representation of the probability estimate

- **Accuracy**
  - a percentage indicating how often a classification model correctly makes predictions

- **AUC-ROC**
  - the Area Under the Curve (AUC) Receiver Operator Characteristic (ROC) is essentially the area under the probability curve plotting True Positive Rate (TPR) against the False Positive Rate (FPR)

2. Regression Metrics:

- **Mean Absolute Error** (MAE)
  - the mean of the absolute difference of the actual and predicted values, which can be useful since its in the same units as our predictor/target variable

- **Mean Square Error** (MSE)
  - a metric similar to MAE but represents a magnitude of error rather than an absolute difference

- **Root Mean Square Error** (RMSE)
  - reverses the MSE back to the units of Mean Error

- **R^2**
  - known as the coefficient of determination, a metric value ranging from 0 to 1 that measures the goodness of fit between actual and predicted values

For this project, since `WVHT` is a continuous variable, we will be using MAE, MSE, $RMSE$, and R^2 to evaluate the performance of most of our models. To easily output each of these outcomes for each model, we will define the general function below `def eval_performance_metrics(model_estimate)`, where the object passed to the `model_estimate` argument is a trained model having an object type of the returned output for the `model.fit(X_sdsurf_train, y_sdsurf_train)` attribute, to more easily call the values for each of these performance metrics:

```
def eval_performance_metrics(model_estimate):
  y_pred = model_estimate.predict(X_sdsurf_test)
  print("R^2  Prediction Error Score: ", r2_score(y_sdsurf_test, y_pred), \
  "\nMAE  Prediction Error Score: ", mean_absolute_error(y_sdsurf_test, y_pred), \
  "\nMSE  Prediction Error Score: ", mean_squared_error(y_sdsurf_test, y_pred), \
  "\nRMSE Prediction Error Score: ", sqrt(mean_squared_error(y_sdsurf_test, y_pred)))
```

8 Our Modeling Pipelines

For this project we will be fitting, tuning, and evaluating various types of **Built-In `Python` Regression Estimators** to build and select my Final Model that best predicts my target variable in this project:

1. Linear Models:
    - Ridge Regression
2. Support Vector Machines:
    - Support Vector Regression
3. K-Nearest Neighbors Methods:
    - K-Nearest Neighbors Regression
4. Boosted Tree Models:
    - Histogram-Based Boosting Tree Regressor

8.1 Linear Models

Out of the many different variations of linear models, we will be using a Ridge Regression to attempt and fit our model. A **Ridge Regression Model** is a variation of the Linear Regression model which works to minimize the residual sum of squares, which can be a down fall in some problems due to its coefficient estimate parameters that rely on the independence of the features. Ridge Regression however, addresses Linear Regression's issue of using Ordinary Least Squares (OLS), by using a penalty parameter that works to minimize the size of coefficients (penalized residual sum of squares).

8.1.1 Ridge Regression

In the `sklearn` module, there are various estimator objects for Ridge Regression. For this problem we will be using the `RidgeCV()` model, which conveniently builds a Ridge Regression with a built in cross-validation feature of the alpha parameter that works identically to a cross-validated grid search to tune the hyperparameters for our final model. Although this model in `sklearn` prefers Leave-One-Out Cross-Validation, I have overrode the CV process to be a Repeated Stratified K-Fold with 5 folds and 5 repeats.

```
# Defining Cross-Validation Procedure
rkf = RepeatedKFold(n_splits=5, n_repeats=5, random_state=seed_val)
# Defining Alpha Values to Grid Search
ridge_alphas = np.logspace(0, 5, 100)

# Defining our Ridge Regression Model with Built-In CV
ridgecv_model = RidgeCV(alphas=ridge_alphas, cv=rkf, scoring='neg_mean_absolute_error')

# Training our Ridge Regression Model with Built-In CV
ridgecv_model = ridgecv_model.fit(X_sdsurf_train, y_sdsurf_train)

# Defining a Function to Output Attributes Specific to RidgeCV()
def ridgecv_model_other_metrics():
  ridgecv_model_bestMSE = absolute(ridgecv_model.best_score_)
  ridgecv_model_nfeatures = ridgecv_model.n_features_in_
  ridgecv_model_bestalpha = ridgecv_model.alpha_
  ridgecv_model_weights = ridgecv_model.coef_
```

```
  print("Best Estimated MSE Score:    ", ridgecv_model_bestMSE, \
  "\nBest Estimated RMSE Score:   ", sqrt(ridgecv_model_bestMSE), \
  "\nRegularization Parameter:    ", ridgecv_model_bestalpha, \
  "\nEstimator's Num of Features: ", ridgecv_model_weights, \
  "\nEstimated Weight Vectors:    ", ridgecv_model_weights, end='')
```

Now that we've finished training our Ridge Regression model on the training data and calculated our performance metric scores, along with other metrics of our estimator, let's see how well our model did!

```
ridgecv_model_other_metrics()
eval_performance_metrics(ridgecv_model)
```

```
## R^2  Prediction Error Score:  0.6491233448758329
## MAE  Prediction Error Score:  1.198893973561779
## MSE  Prediction Error Score:  2.6320537070147103
## RMSE Prediction Error Score:  1.6223605354589683
```

As we can see above we used the following process to evaluate our model:

1. First we defined our cross validation process of a Repeated Stratified K-Fold with 5 folds and 5 repeats,
2. We defined our base assumption model with `RidgeCV()`'s built in functionality to also define a range 100 of alpha values between 0 and 5 to grid search,
3. Using this `RidgeCV()` object, we train the model on the training data using the `RidgeCV.fit()` method,
4. We then access the model's `RidgeCV.fit.alpha_` attribute to see our model's chosen estimated regularization parameter of alpha (which we discover is $\alpha = 1.0$), as well as `RidgeCV.fit.best_score_` attribute to access the MSE score with the best value of alpha (which we see is approximately $MSE = 1.206$),
5. Next we begin the process of evaluating our model using `predict()` to use our test set of X to make predictions of Y,
6. Finally to make an evaluation of our final Ridge Regression model, we compare our predicted values of Y to the true values of Y in our `y_sdsurf_test` data set

```
ridgecv_model_estimators = []
ridgecv_model_estimators.append(('standardize', StandardScaler()))
ridgecv_model_estimators.append(('RidgeCV', RidgeCV()))
ridge_model = Pipeline(ridgecv_model_estimators)
ridge_model
```

```
## Pipeline(steps=[('standardize', StandardScaler()), ('RidgeCV', RidgeCV())])
```

8.2 Support Vector Machines

Before we begin building this model, we'll first give a brief introduction to **Support Vector Machines (SVMs)**. This particular type of supervised learning methods has the great functionality to thrive in high dimensional spaces (even when the number of dimensions > the number of samples) and to utilize memory

efficiently by using a subset of the training data in the decision function (also known as **support vectors**). However, SVMs have trouble with over-fitting (when the number of features (far greater) > the number of samples) and SVMs cannot output estimates of probability explicitly.

Within this class of supervised learning methods we call Support Vector Machines (SVMs), we will be using a **Support Vector Regression** to attempt and fit our model for this project. A Support Vector Regression model is a method of SVM that only depends on the training set due to its inherent functionality that ignores samples who have predictions close to the target.

8.2.1 Nu SVR

In the `sklearn.svm` class, there are various estimator model objects for Support Vector Regression: `SVR()`, `NuSVR()`, and `LinearSVR()`. For this problem we will be using the `NuSVR()` model, which uses a slightly different formula in comparison to the traditional SVR and Linear SVR models, where our parameter `nu=` replaces the parameter epsilon and indicates an upper bound ("on the fraction of training errors") and a lower bound ("of the fraction of support vectors"). With a default of `nu=0.5`, we are able to set `nu` to be any value in the interval $(0, 1]$.

```
# Defining Cross-Validation Procedure
rkf = RepeatedKFold(n_splits=5, n_repeats=5, random_state=seed_val)

# Defining our Nu-Support Vector Regressor Model
NuSVR_model = NuSVR()

# Training our Nu-Support Vector Regressor Model
NuSVR_model = NuSVR_model.fit(X_sdsurf_train, y_sdsurf_train)

# Defining a Function to Output Attributes Specific to NuSVR()
def NuSVR_model_other_metrics():
  NuSVR_model_fitparams = NuSVR_model.get_params
  print("Estimated Parameters:        ", NuSVR_model_fitparams)
```

Now that we've finished training our Nu-Support Vector Regression model on the training data and calculated our performance metric scores, along with other metrics of our estimator, let's see how well our model did!

```
NuSVR_model_other_metrics()
eval_performance_metrics(NuSVR_model)
```

```
## R^2  Prediction Error Score:  0.064451871720105 36
## MAE  Prediction Error Score:  1.9674437575462775
## MSE  Prediction Error Score:  7.017887577212 51
## RMSE Prediction Error Score:  2.649129588603115
```

As we can see above we used the following process to evaluate our model:

1. First we defined our cross validation process of a Repeated Stratified K-Fold with 5 folds and 5 repeats,
2. We defined our base assumption model with `NuSVR()`,
3. Using this `NuSVR()` object, we train the model on the training data using the `NuSVR.fit()` method,

4. We then access the model's `NuSVR_model.get_params` attribute to see our model's chosen estimated parameter of epsilon,

5. Next we begin the process of evaluating our model using our `eval_performance_metrics()` user-defined function to use our test set of X to make predictions of Y,

6. Finally to make an evaluations of our final Nu-Support Vector Regression model, we compare our predicted values of Y to the true values of Y in our `y_sdsurf_test` data set

```
NuSVR_model_estimators = []
NuSVR_model_estimators.append(('standardize', StandardScaler()))
NuSVR_model_estimators.append(('NuSVR', NuSVR()))
NuSVR_model = Pipeline(NuSVR_model_estimators)
NuSVR_model
```

```
## Pipeline(steps=[('standardize', StandardScaler()), ('NuSVR', NuSVR())])
```

8.3 K-Nearest Neighbors Methods

Before building our model, let's talk about what **K-Nearest Neighbors** model entails. The idea is that with a user-defined number for `k`, our model uses the training data to find `k` neighboring points that are closest to the "new" point it has assigned a label to, also known as a **query point**. According to the `sklearn` documentation, K-Nearest Neighbors Methods are considered **non-generalizing machine learning methods**, largely due to the concept that these methods " 'remember' all of their training data".

8.3.1 K-Nearest Neighbors Regression Model

In the `sklearn.neighbors` class, we are offered a variety of K-Nearest Neighbors Methods. For this particular data set, we will be using a **K-Nearest Neighbors Regression** with the `KNeighborsRegressor()` model, which can be used for data that is comprised of continuous data labels as opposed to discrete variables. This method computes each label that is assigned to a query point by calculating the mean of the labels of the `k` amount of nearest neighboring points.

Besides defining the value of `k`, there are a couple other ways that the model "chooses" its next query point to create its regression estimate. Within the `sklearn.neighbors` class, there are two weighting parameter options offered for a KNN Regressor: `weights = 'uniform'` or `weights = 'distance'`. The default value for the `weights` parameter, `'uniform'`, assigns equal weight to all points, while `'distance'` considers proportional weights that are "inverse of the distance from the query point". Yes this concept can be a little abstract and difficult to picture, so I've included a figure below from `sklearn`'s official documentation to give a visual representation for further explanation:

As we can see, although we have a very accurate prediction when `weights = 'distance'`, we have a very clear case of over-fitting that would be extremely detrimental in our final evaluation of our model's performance. Therefore for this project, we will be using the default of the `KNeighborsRegressor()` model, and set `weights = 'uniform'` to build a more generalized regression estimate to make a prediction of "new/unknown" data that we can evaluate with our testing data set.

Another important parameter of our model we should discuss prior, is our choice of a nearest neighbor algorithm. The `KNeighborsRegressor()` model still depends on one of the chosen algorithms passed to the `algorithm=` parameter to compute the nearest neighbors: **Brute Force**, **K-D Tree**, or a **Ball Tree**. The optimal choice of an algorithm parameter depends on a multitude of factors that varies per data set: number of samples N and dimensionality D, intrinsic dimensionality and/or sparsity of the data, the amount of `k` neighbors set for a query point, and number of requested query points. For this project, my choice of a K-D

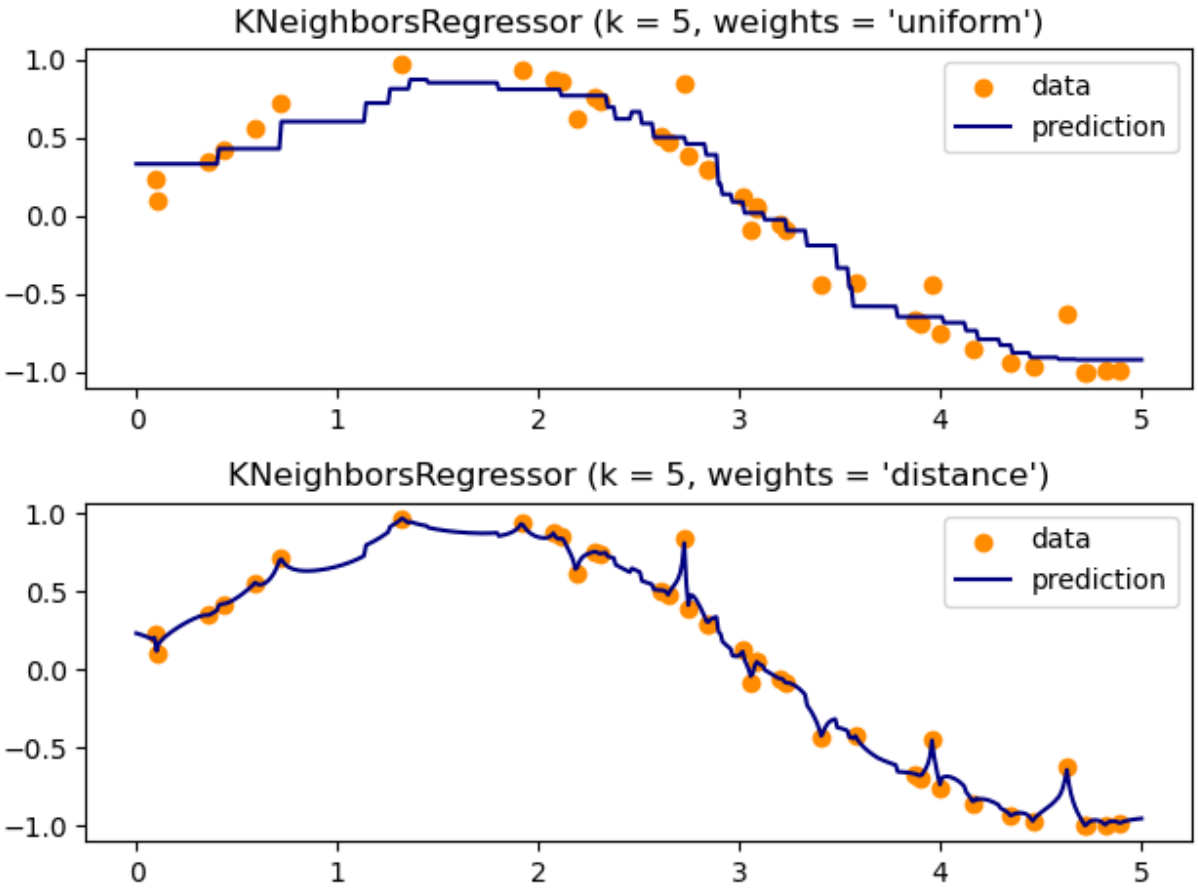


Figure 3: K-Nearest Neighbors Regression Explanation

Tree algorithm was largely due to the size of my data set. A **K-D Tree** algorithm is a method that allows a user to decrease the `leaf_size=` parameter to improve model construction time. Although the Brute Force algorithm is generally considered more efficient than tree-based algorithms, in this project's case it would be much slower as its query time increases as the number of samples N and dimensionality D grows: $O(DN)$. However, rather than a usual **Linearithmic** run-time of $O(DlogN)$ with K-D Trees, with a large data set, K-D Trees improve run-time to roughly become closer to a **Quadratic** time of $O(DN)$.

```
# Defining Cross-Validation Procedure
rkf = RepeatedKFold(n_splits=5, n_repeats=5, random_state=seed_val)

# Defining our K-Nearest Neighbors Regressor Model
KNN_model = KNeighborsRegressor(algorithm='kd_tree', leaf_size=55000)

# Training our K-Nearest Neighbors Regressor Model
KNN_model = KNN_model.fit(X_sdsurf_train, y_sdsurf_train)

# Defining a Function to Output Attributes Specific to KNeighborsRegressor()
def KNN_model_other_metrics():
  KNN_model_fitparams = KNN_model.get_params
  KNN_model_graph = KNN_model.kneighbors_graph(X_sdsurf_train)
  print("Estimated Parameters:        ", KNN_model_fitparams)
```

Now that we've finished training our K-Nearest Neighbors Regression model on the training data and calculated our performance metric scores, along with other metrics of our estimator, let's see how well our model did!

```
KNN_model_other_metrics()
eval_performance_metrics(KNN_model)
```

```
## R^2  Prediction Error Score:  0.656779427108612
## MAE  Prediction Error Score:  1.1528233611966563
## MSE  Prediction Error Score:  2.5746226430561667
## RMSE Prediction Error Score:  1.6045630692048745
```

As we can see above we used the following process to evaluate our model:

1. First we defined our cross validation process of a Repeated Stratified K-Fold with 5 folds and 5 repeats,

2. We defined our base assumption model with `KNeighborsRegressor(algorithm='kd_tree', leaf_size=55000)`,

3. Using this `KNeighborsRegressor()` object, we train the model on the training data using the `KNeighborsRegressor.fit()` method,

4. We then access the model's `NuSVR_model.get_params` attribute to return a dictionary object of the model's chosen estimated parameters,

5. Next we begin the process of evaluating our model using our `eval_performance_metrics()` user-defined function to use our test set of X to make predictions of Y,

6. Finally to make an evaluations of our final K-Nearest Neighbors Regression model, we compare our predicted values of Y to the true values of Y in our `y_sdsurf_test` data set

```
KNN_model_estimators = []
KNN_model_estimators.append(('standardize', StandardScaler()))
KNN_model_estimators.append(('KNN Regressor', KNeighborsRegressor()))
KNN_model = Pipeline(KNN_model_estimators)
KNN_model
```

```
## Pipeline(steps=[('standardize', StandardScaler()),
##                 ('KNN Regressor', KNeighborsRegressor())])
```

8.4 Boosted Tree Models

In the class of **Boosted Tree Models**, `sklearn` provides us with a plethora of options to consider. For this project we will be using a **Histogram-Based Gradient Boosting Tree Regressor** to attempt and fit our model. Similar to the common **Gradient Boosting Tree Regressor**, which builds an additive model in a forward stage-wise fashion optimizing arbitrary differentiable loss functions, this model estimator we will use is encourages by `sklearn` "when the number of samples is larger than tens of thousands of samples" due to its functionality that is "**orders of magnitude faster**". By using bin sizes to reduce the number of points to split on, we are able to train a model on our very large data set of continuous values.

8.4.1 Histogram-Based Gradient Boosting Tree Regression Model

According to the official documentation from `sklearn`'s website, the process of the `HistGradientBoostingRegressor()` method is as follows. As a sub-method of the `ensemble` class, it works similarly to a `GradientBoostingRegressor()` with a bottleneck procedure of building decision trees. However, as opposed to the tradition Gradient Boosting Regressor Tree procedure, the Histogram Based Gradient Boosting Tree Regressor does not require sorting the values of features but rather uses the histogram data-structure that "implicitly" orders the samples. Due to this, we have a much faster model building procedure with a O-notation run-time of $O(n_{\text{features}} \times n)$ as opposed to `GradientBoostingRegressor()`'s much more lengthy complex run-time of $O(n_{\text{features}} \times nlog(n))$.

```
# Defining Cross-Validation Procedure
rkf = RepeatedKFold(n_splits=5, n_repeats=5, random_state=seed_val)

# Defining our Histogram-Based Gradient Boosting Tree Regressor Model
HGBoost_model = HistGradientBoostingRegressor(loss='squared_error', random_state=seed_val)

# Training our Histogram-Based Gradient Boosting Tree Regressor Model
HGBoost_model = HGBoost_model.fit(X_sdsurf_train, y_sdsurf_train)

# Defining a Function to Output Attributes Specific to HistGradientBoostingRegressor()
def HGBoost_model_other_metrics():
  # number of iterations of the boosting process
  HGBoost_num_iter = HGBoost_model.n_iter_
  # number of tree that are built at each iteration
  HGBoost_tree_per_iter = HGBoost_model.n_trees_per_iteration_
  # scores at each iteration on the training data
  HGBoost_train_score = HGBoost_model.train_score_
  # scores at each iteration on the held-out validation data
  HGBoost_val_score = HGBoost_model.validation_score_
  HGBoost_model_fitparams = HGBoost_model.get_params
  print("Number of Iterations of the Boosting Process:    ", HGBoost_num_iter, \
```

```
  "\nNumber of Trees Built at each Iteration:         ", HGBoost_tree_per_iter, \
  "\nScores at each Iteration on the Training Data:   ", HGBoost_train_score, \
  "\nScores at each Iteration on the Validation Data: ", HGBoost_val_score)
```

Now that we've finished training our Histogram-Based Gradient Boosting Tree Regression model on the training data and calculated our performance metric scores, along with other metrics of our estimator, let's see how well our model did!

```
HGBoost_model_other_metrics()
eval_performance_metrics(HGBoost_model)
```

```
## R^2  Prediction Error Score:  0.7927232133069582
## MAE  Prediction Error Score:  0.9096908421897236
## MSE  Prediction Error Score:  1.554858742598466
## RMSE Prediction Error Score:  1.2469397509897846
```

As we can see above we used the following process to evaluate our model:

1. First we defined our cross validation process of a Repeated Stratified K-Fold with 5 folds and 5 repeats,
2. We defined our base assumption model with `HistGradientBoostingRegressor(loss='squared_error', random_state=seed_val)`,
3. Using this `HistGradientBoostingRegressor()` object, we train the model on the training data using the `HistGradientBoostingRegressor.fit()` method,
4. We then access the model's `HGBoost_model.n_iter_`, `HGBoost_model.n_trees_per_iteration_`, `HGBoost_model.train_score_`, and `HGBoost_model.validation_score_` attributes,
5. Next we begin the process of evaluating our model using our `eval_performance_metrics()` user-defined function to use our test set of X to make predictions of Y,
6. Finally to make an evaluations of our final Histogram-Based Gradient Boosting Tree Regression model, we compare our predicted values of Y to the true values of Y in our `y_sdsurf_test` data set

```
HGBoost_model_estimators = []
HGBoost_model_estimators.append(('standardize', StandardScaler()))
HGBoost_model_estimators.append(('HG Boosted Tree', HistGradientBoostingRegressor()))
HGBoost_model = Pipeline(HGBoost_model_estimators)
HGBoost_model
```

```
## Pipeline(steps=[('standardize', StandardScaler()),
##                 ('HG Boosted Tree', HistGradientBoostingRegressor())])
```

8.5 Outcomes of All Our Fitted Models

Sadly, it's a little difficult to make pretty looking tables when using `library(knitr)` with `python` in `rmarkdown`. So instead I'll present the results of our models with a printed code-chunk:

```
RidgeCV_list = \
[["MSE Estimate", 1.2065245693118951], \
["RMSE Estimate", 1.0984191227905198], \
["R^2 Prediction Error", 0.6491233448758329], \
["MAE Prediction Error", 1.198893973561779], \
["MSE Prediction Error", 2.6320537070147103], \
["RMSE Prediction Error", 1.6223605354589683]]
# Ridge Regression Model
## Run-Time: 6.75 MINUTES
##
## Best Estimated MSE Score:     1.2065245693118951
## Best Estimated RMSE Score:    1.0984191227905198
## Regularization Parameter:     1.0
## Estimator's Num of Features:
###[ 1.50720947e-02  4.45326729e-02  2.91260273e-03  1.99231367e-02  ...
####-2.08471023e-03 -3.48544892e-04  7.25502860e-01  1.17954178e+00  ...
####-6.31578249e-02 -1.77825914e-01 -7.03314115e-02]
## Estimated Weight Vectors:
###[ 1.50720947e-02  4.45326729e-02  2.91260273e-03  1.99231367e-02  ...
####-2.08471023e-03 -3.48544892e-04  7.25502860e-01  1.17954178e+00  ...
####-6.31578249e-02 -1.77825914e-01 -7.03314115e-02]
##
## R^2  Prediction Error Score:  0.6491233448758329
## MAE  Prediction Error Score:  1.198893973561779
## MSE  Prediction Error Score:  2.6320537070147103
## RMSE Prediction Error Score:  1.6223605354589683

NuSVR_list = \
[["Estimated Parameters", "<bound method BaseEstimator.get_params of NuSVR()>"], \
["R^2 Prediction Error", 0.06445187172010536], \
["MSE Prediction Error", 7.01788757721251], \
["RMSE Prediction Error", 2.649129588603115]]
# NU Support Vector Regressor Model
## Run-Time: 42 MINUTES
##
## Estimated Parameters:  <bound method BaseEstimator.get_params of NuSVR()>
## Best Estimated R^2 Error Score:  0.06445187172010536
##
## R^2  Prediction Error Score:  0.06445187172010536
## MAE  Prediction Error Score:  1.9674437575462775
## MSE  Prediction Error Score:  7.01788757721251
## RMSE Prediction Error Score:  2.649129588603115

KNNRegression_list = \
[["Estimated Parameters", "<bound method BaseEstimator.get_params of KNeighborsRegressor(algorithm='kd_
["R^2 Prediction Error", 0.656779427108612], \
["MAE Prediction Error", 1.1528233611966563], \
["MSE Prediction Error", 2.5746226430561667], \
["RMSE Prediction Error", 1.6045630692048745]]
# K-Nearest Neighbors Regressor Model
## Run-Time: 13.32 MINUTES
##
## Estimated Parameters:  <bound method BaseEstimator.get_params of ...
```

```
####KNeighborsRegressor(algorithm='kd_tree', leaf_size=55000)>
##
## R^2  Prediction Error Score:  0.6567794271086l2
## MAE  Prediction Error Score:  1.1528233611966563
## MSE  Prediction Error Score:  2.5746226430561667
## RMSE Prediction Error Score:  1.6045630692048745

HGBoostingTree_list = \
[["R^2 Prediction Error", 0.7927232133069582], \
["MAE Prediction Error", 0.9096908421897236], \
["MSE Prediction Error", 1.554858742598466], \
["RMSE Prediction Error", 1.2469397509897846]]
# Histogram Based Gradient Boosted Tree Regressor Model
## Run-Time: 11.31 MINUTES
##
## Number of Iterations of the Boosting Process:     100
## Number of Trees Built at each Iteration:          1
## Scores at each Iteration on the Training Data:
###[-3.75021145 -3.28277944 -2.89955505 -2.58402141 -2.32314826 -2.10764126...
####-1.9292971  -1.78060077 -1.6575275  -1.55523214 -1.4703614  -1.39811711...
####-1.33642892 -1.28496098 -1.24134504 -1.20293853 -1.16920909 -1.13999025...
####-1.1140026  -1.09093587 -1.07068296 -1.0532467  -1.03784021 -1.02254291...
####-1.0099285  -0.99695129 -0.985819   -0.97486085 -0.96439656 -0.95535099...
####-0.94636323 -0.93777208 -0.93010544 -0.92335734 -0.91699201 -0.9113778...
####-0.90511156 -0.89993353 -0.89556786 -0.89073488 -0.88578256 -0.88042414...
####-0.87659351 -0.87176424 -0.86751813 -0.86304791 -0.85894774 -0.85525728...
####-0.85038543 -0.84640595 -0.8418726  -0.8378544  -0.83493557 -0.8318117...
####-0.82907227 -0.82451846 -0.81935704 -0.81568832 -0.81188354 -0.80927026...
####-0.80594829 -0.80245589 -0.80029438 -0.79754957 -0.79288478 -0.78960567...
####-0.7876911  -0.78538664 -0.78325049 -0.78146614 -0.77857077 -0.77509238...
####-0.77305504 -0.77050881 -0.7687818  -0.76697535 -0.76534875 -0.76378855...
####-0.76248402 -0.7603682  -0.75765686 -0.75505326 -0.75339461 -0.75116306...
####-0.74767242 -0.74654452 -0.74440524 -0.74271636 -0.74144227 -0.73970856...
####-0.73771028 -0.73597812 -0.73455302 -0.73314028 -0.73152769 -0.7303079...
####-0.72863856 -0.72521117 -0.72406323 -0.72260599 -0.72046553]
## Scores at each Iteration on the Validation Data:
###[-3.8951256  -3.40740423 -3.00471518 -2.67424108 -2.39787818 -2.17034601...
####-1.9811154  -1.82635875 -1.69716831 -1.5900641  -1.50332734 -1.42738968...
####-1.36304692 -1.30871296 -1.26414092 -1.22460024 -1.19018197 -1.16047135...
####-1.1337788  -1.10977959 -1.08968747 -1.07225621 -1.05667384 -1.04152844...
####-1.02935247 -1.01663528 -1.00527375 -0.99504775 -0.9845851l -0.97573178...
####-0.9676125  -0.95951315 -0.95247316 -0.94538026 -0.93861594 -0.93321987...
####-0.92694518 -0.92213752 -0.91786171 -0.91350649 -0.90821865 -0.90361976...
####-0.90009793 -0.89442805 -0.8908212  -0.8864947  -0.88254944 -0.87894295...
####-0.87458778 -0.87050373 -0.86753138 -0.86402059 -0.86153502 -0.8588291...
####-0.85628172 -0.8516012  -0.84696414 -0.84427563 -0.84043574 -0.83833873...
####-0.83589592 -0.83293754 -0.83123918 -0.82894951 -0.82413826 -0.82076385...
####-0.81934561 -0.81726783 -0.81542093 -0.81438989 -0.81177292 -0.80831806...
####-0.8064438  -0.8042818  -0.80290851 -0.8005627  -0.79956915 -0.79804982...
####-0.79716241 -0.7951368  -0.79317753 -0.790255   -0.78889544 -0.78683292...
####-0.78317951 -0.78277919 -0.78058299 -0.77960802 -0.77850707 -0.7770815...
####-0.77588508 -0.77431917 -0.77320405 -0.77211576 -0.77077699 -0.76995714...
####-0.76832425 -0.76566851 -0.76505363 -0.76354498 -0.76173909]
```

```
##
## R^2  Prediction Error Score:  0.7927232133069582
## MAE  Prediction Error Score:  0.9096908421897236
## MSE  Prediction Error Score:  1.554858742598466
## RMSE Prediction Error Score:  1.2469397509897846
```

As we can see for each of our 4 models, we have our final prediction error scores using the metrics MAE, MSE, $RMSE$, and R^2 to evaluate the performance of most of our models. So let's see who our best-performing model is...

9 Model Awards Ceremony!

Looking at my terrible makeshift `python in rmarkdown` table in the figure below, we can see a chart of the run-times and final prediction error score metrics (R^2, MAE, MSE, and $RMSE$) for each of our 4 Regression Models: Ridge Regression, Nu-Support Vector Regression, K-Nearest Neighbors Regression, and Histogram-Based Gradient Boosting Tree Regression.

However, before announcing the Best-Performing Model and its parameters, so that none of my models feel left out, I'm going to give out some of what I like to call **Model Performance Superlative Awards**!

```
#  ____________________________________________________________________________
# | Regression Model    | Run-Time  | R^2 Score | MAE Score | MSE Score | RMSE Score |
# |---------------------|-----------|-----------|-----------|-----------|------------|
# | Ridge Regression    | 6.75 MIN  | 0.6491    | 1.1989    | 2.6321    | 1.6224     |
# | NU Support Vector   | 42.0 MIN  | 0.0645    | 1.9674    | 7.0179    | 2.6491     |
# | K-Nearest Neighbors | 13.32 MIN | 0.6568    | 1.1528    | 2.5746    | 1.6046     |
# | HB Gradient Boosted | 11.31 MIN | 0.7927    | 0.9097    | 1.5549    | 1.2469     |
# |_____________________|___________|___________|___________|___________|____________|
```

9.1 "Gives a Pressing Knit Phobia"

Thanks to `rmarkdown`'s chunk option `cache=TRUE`, this model was somewhat less of a hassle to run as time went on. Yet, despite this seemingly fool proof loop hole, this model still found a way to bring my HTML knitting run-time into the hours.

With a run-time averaging approximately 42 Minutes per HTML Knit, I proudly present this years recipient of the "Gives a Pressing Knit Phobia" Award: the **Nu-Support Vector Regression Model**!!!

9.2 "Was in the Final Submission I Guess"

"No model is a bad model", is what you would think most would say to you to keep your model building spirits up, but sometimes as much as we love 'em, some models are just not meant to impress the audience or steal the show.

With a whopping, crowd-astonishing, ground-breaking R^2 Score of 0.0645, or better put as an approximate 6.45% prediction accuracy score, I am pleased to announce that this year the prestigious "Was in the Final Submission I Guess" Award (a.k.a "The Glorified Participation Award" Award) goes to yet again tonight's fan favorite: the **Nu-Support Vector Regression Model**!!!

9.3 "You Made Life so Easy"

Moving onto our last award before announcing our most important award, the "America's Next Top Model" Award, we will be giving out the "You Made Life so Easy" Award. This award goes to everyone's favorite model that they wished finished #1 this year: tall, beautiful, fit, efficient, and built-in hyperparameter tuning.

With a built-in functionality for hyperparameter searching and tuning, a fairly decent R^2 Score of 0.6491, and lastly a quick efficient run-time of 6.75 minutes per HTML knit to finish training, tuning-up, and evaluating; the winner of the 2022 "You Made Life so Easy" Award is the **Ridge CV Regression Model**!!!

9.4 "America's Next Top Model"

Moving onto the main event, the model we crown the model of all models (for this project), the model who needs no introduction...

...the recipient of this year's "America's Next Top Model" Award...

Figure 4: :o

...can be found in the next section after the commercial break :(

10 Who's our Best-Fitting Model?

Finally! Our recipient for the 2022 "America's Next Top Model" Award! Without further ado,

With a speedy run-time of 11.31 minutes, an $MAE = 0.9097$, $MSE = 1.5549$, $RMSE = 1.2469$, and most importantly an $R^2 = 0.7927$, better interpreted as a 79.27% prediction accuracy. We can confidently say that when using 100 iterations of each of the boosting processes with 1 tree built at each iteration, we can conclude our best performing model and this year's crowned "America's Next Top Model" is... the **Histogram-Based Gradient Boosting Tree Regression Model**!!!

11 Final Thoughts

As cliche as it is, I really have had a lot of fun with this project. Overall, I'd say I learned a lot having to translate the material I've learned throughout the 11 weeks of this course from `R` to `Python`. This process was most definitely a recursive loop of trial and errors and hours of knitting. Despite the somewhat occasionally tedious processes, I've grown a much deeper understanding of Machine Learning in a way that I could have never gotten from a tutorial video or the chapter of a textbook. Getting the chance to research Machine Learning theory, methods, and applications through the lens of one of my favorite pastimes: surfing! I've gained knowledge in such a fun and exploratory manner, I know it'll take me a step closer to finding what kind of a future I want and where my niche is in the world of Data Science and Software Engineering.

Although I haven't created the next Surfline, I think my model will serve its short term purpose at the moment. Hopefully, after some additional research, considerations of other learning methods, some tinkering, and maybe the inclusion of some web-scraped data from other my projects; I could produce a better model to make some slightly less sup-par predictions.

New updates are hopefully soon to come!

To conclude this project, here's one of the very few pictures that exist of myself in the water. Yes I know I probably should have gone left.

Figure 5: Black's Beach, San Diego (Winter 2020)