West Nile Virus Prediction

Predicting the likelihood of West Nile virus in the mosquitoes of Chicago

Final Report

Paul Marten
June 2021

I. Introduction

1. The Problem

West Nile virus (WNV) is the leading cause of mosquito-borne disease in the continental United States. It was discovered in 1927 in the West Nile sub-region of Uganda. It is most commonly spread to people through bites from infected mosquitoes. The first serious outbreaks of WNV occurred in the mid-1990s in Algeria and Romania. The virus was introduced in the United States in 1999, with the first case being identified in New York City. About 1 in 5 people who are infected develop a fever and other symptoms. About 1 out of 150 infected people develop a serious, sometimes fatal, illness.

West Nile virus was first detected in humans Chicago in 2002. A year later, the city faced an epidemic in which 225 people were infected, and 22 of these cases were fatal (this is approximately a 10% mortality rate). By 2004 the City of Chicago and the Chicago Department of Public Health (CDPH) had established a comprehensive surveillance and control program that is still in effect today. Every week from late spring through the fall, mosquitos in traps across the city are tested for the virus. The results of these tests influence when and where the city will spray airborne pesticides to control adult mosquito populations.

The goal of this project is to determine when and where mosquitos will be most likely to test positive for West Nile virus. Chicago needs a more accurate method of predicting outbreaks of the virus in order to more effectively and efficiently allocate resources to help stop the transmission of it. My analysis will initially focus on determining which features of weather and location will be most effective in making these predictions. After having figured this out, I must determine what the most effective modeling technique is for the predictions.

II. Data Sourcing, Loading, and Transformation

1. Data Overview (Data found at https://www.kaggle.com/c/predict-west-nile-virus/data)

Weather Data: This dataset, found as weather.csv, contains 2944 records from 2007 to 2014 and has 21 features. These features consist of a multitude of weather conditions, and includes but is not limited to, sunrise/sunset, average temperature, total precipitation, and wind speed and direction.

Main Data: This dataset, found as train.csv, contains 10506 records from 2007, 2009, 2011, and 2013 and has 11 features. These features consist of address, species, trap type, the number of mosquitoes caught, and the target variable, 'WnvPresent', which denotes whether or not the virus was found.

2. Data Cleaning and Preparation

(Both datasets were read into the notebook using pandas.read_csv( ), with the arguments: index_col='Date', parse_dates=True so as to use the date column as the index in chronological order)

Weather Data:

Before

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2944 entries, 2007-05-01 to 2014-10-31
Data columns (total 21 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Station      2944 non-null   int64
 1   Tmax         2944 non-null   int64
 2   Tmin         2944 non-null   int64
 3   Tavg         2944 non-null   object
 4   Depart       2944 non-null   object
 5   DewPoint     2944 non-null   int64
 6   WetBulb      2944 non-null   object
 7   Heat         2944 non-null   object
 8   Cool         2944 non-null   object
 9   Sunrise      2944 non-null   object
 10  Sunset       2944 non-null   object
 11  CodeSum      2944 non-null   object
 12  Depth        2944 non-null   object
 13  Water1       2944 non-null   object
 14  SnowFall     2944 non-null   object
 15  PrecipTotal  2944 non-null   object
 16  StnPressure  2944 non-null   object
 17  SeaLevel     2944 non-null   object
 18  ResultSpeed  2944 non-null   float64
 19  ResultDir    2944 non-null   int64
 20  AvgSpeed     2944 non-null   object
dtypes: float64(1), int64(5), object(15)
memory usage: 506.0+ KB
```

After

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1460 entries, 2007-05-01 to 2014-10
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Tmax         1460 non-null   float64
 1   Tmin         1460 non-null   float64
 2   Tavg         1460 non-null   float64
 3   DewPoint     1460 non-null   float64
 4   WetBulb      1460 non-null   float64
 5   Heat         1460 non-null   float64
 6   Cool         1460 non-null   float64
 7   Depth        1460 non-null   float64
 8   SnowFall     1460 non-null   float64
 9   PrecipTotal  1460 non-null   float64
 10  StnPressure  1460 non-null   float64
 11  SeaLevel     1460 non-null   float64
 12  ResultSpeed  1460 non-null   float64
 13  ResultDir    1460 non-null   float64
 14  AvgSpeed     1460 non-null   float64
dtypes: float64(15)
memory usage: 182.5 KB
```

- At first glance, it wasn't apparent that there were as many missing values as there were. This was due to the fact that some were denoted as "M" for missing, "T" for trace, and "-", as opposed to the traditional NaN entry. Because of this, I replaced all of these entries with NaNs for easier analysis. After doing so, I checked the amount of null entries.

```
In [11]: weather.isnull().sum(axis = 0)

Out[11]: Station            0
         Tmax               0
         Tmin               0
         Tavg              11
         Depart          1472
         DewPoint           0
         WetBulb            4
         Heat              11
         Cool              11
         Sunrise         1472
         Sunset          1472
         CodeSum            0
         Depth           1472
         Water1          2944
         SnowFall        1472
         PrecipTotal        2
         StnPressure        4
         SeaLevel           9
         ResultSpeed        0
         ResultDir          0
         AvgSpeed           3
         dtype: int64
```

- Water1 was entirely NaN, Depart was half empty, and being as that there were 2 rows per day, half of sunrise and sunset were NaN. I decided to drop these rows entirely, seeing as that they were not entirely interesting to the problem at hand.

- SnowFall, Depth, and PrecipTotal had a lot of 0's, so for the time being I imputed the missing values with 0's.

- Investigating the categorical variables, CodeSum and Station, CodeSum had a ton of unique values. This would be an issue for dummy variables, later on, so I dropped this variable all together. The Station variable merely let us know the weather conditions per station, so I dropped

```
array([' ', 'BR', 'BR HZ', 'HZ', 'RA', 'RA BR', 'TSRA RA BR', 'RA VCTS',
       'TSRA RA', 'RA HZ', 'TSRA RA BR HZ', 'TSRA BR HZ', 'RA BR HZ VCTS',
       'TSRA RA HZ', 'TSRA BR HZ VCTS', 'TSRA', 'TSRA BR HZ FU',
       'TSRA RA HZ FU', 'BR HZ FU', 'TSRA RA VCTS', 'HZ VCTS', 'TSRA HZ',
       'VCTS', 'RA BR VCTS', 'TSRA RA BR VCTS', 'TS TSRA RA BR HZ VCTS',
       'DZ BR', 'TS TSRA RA BR HZ', 'TS TSRA BR HZ', 'RA BR HZ',
       'TSRA RA DZ BR HZ', 'TS TSRA RA BR', 'TS RA BR', 'TS TSRA RA',
       'TS TSRA RA BR VCTS', 'TS TSRA BR', 'TS RA', 'RA BCFG BR',
       'TSRA BR', 'RA DZ FG+ BCFG BR', 'RA FG+ MIFG BR', 'RA DZ',
       'RA DZ BR', 'TS TSRA RA HZ', 'TSRA RA FG+ FG BR',
       'TSRA DZ FG+ FG BR HZ', 'TS BR', 'RA BR SQ', 'TS TSRA',
       'TSRA RA BR HZ VCTS', 'BR VCTS', 'TS', 'FG+ BR HZ', 'RA SN',
       'TSRA RA DZ BR', 'DZ BR HZ', 'RA BR FU', 'TS BR HZ', 'DZ',
       'FG+ BR', 'FG+ FG BR', 'FG+ MIFG BR', 'TSRA RA FG BR',
       'TSRA FG+ BR', 'RA DZ BR HZ', 'RA DZ SN', 'FG+ FG BR HZ',
       'TS TSRA RA FG BR', 'BR HZ VCFG', 'TS RA FG+ FG BR',
       'TSRA RA FG+ BR', 'RA DZ FG+ FG BR', 'TS TSRA RA VCTS', 'FU',
       'TS TSRA VCFG', 'TS TSRA HZ', 'TS TSRA GR RA BR', 'RA FG BR',
       'HZ FU', 'RA BR HZ FU', 'FG+ BCFG BR', 'TSRA RA FG+ FG BR HZ',
       'FG+', 'TSRA BR SQ', 'TSRA DZ BR HZ', 'RA BR HZ VCFG', 'RA FG+ BR',
       'FG BR HZ', 'TS HZ', 'TS TSRA RA FG BR HZ', 'RA DZ FG+ BR',
       'RA DZ FG+ BR HZ', 'TSRA FG+ BR HZ', 'RA BR VCFG', 'TS RA BR HZ',
       'BCFG BR', 'RA SN BR'], dtype=object)
```

merely let us know the weather conditions per station, so I dropped

this column and averaged the two measurements per day into one to get a general understanding of the weather across the city on any particular day.

- Because of the way the file denoted the missing values as characters, a lot of features were objects that should have been numeric. For all the should be remaining numeric columns, I converted them to numeric ones and the data was ready for the joining.

Main Data:

Before

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 10506 entries, 2007-05-29 to 2013-09-26
Data columns (total 11 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Address                 10506 non-null  object
 1   Species                 10506 non-null  object
 2   Block                   10506 non-null  int64
 3   Street                  10506 non-null  object
 4   Trap                    10506 non-null  object
 5   AddressNumberAndStreet  10506 non-null  object
 6   Latitude                10506 non-null  float64
 7   Longitude               10506 non-null  float64
 8   AddressAccuracy         10506 non-null  int64
 9   NumMosquitos            10506 non-null  int64
 10  WnvPresent              10506 non-null  int64
dtypes: float64(2), int64(4), object(5)
memory usage: 984.9+ KB
```

After

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 10506 entries, 2007-05-29 to 2013-09-26
Data columns (total 14 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Address                         10506 non-null  object
 1   Street                          10506 non-null  object
 2   Trap                            10506 non-null  object
 3   Latitude                        10506 non-null  float64
 4   Longitude                       10506 non-null  float64
 5   NumMosquitos                    10506 non-null  int64
 6   WnvPresent                      10506 non-null  int64
 7   Species_CULEX ERRATICUS         10506 non-null  uint8
 8   Species_CULEX PIPIENS           10506 non-null  uint8
 9   Species_CULEX PIPIENS/RESTUANS  10506 non-null  uint8
 10  Species_CULEX RESTUANS          10506 non-null  uint8
 11  Species_CULEX SALINARIUS        10506 non-null  uint8
 12  Species_CULEX TARSALIS          10506 non-null  uint8
 13  Species_CULEX TERRITANS         10506 non-null  uint8
dtypes: float64(2), int64(2), object(3), uint8(7)
memory usage: 728.4+ KB
```

- Magically, there were no missing values nor duplicates so this dataset was already off to a better start.

- There were multiple features for the address, so I narrowed it down to just Address, Longitude, and Latitude to reduce redundancy.

- To work with the species column, I used a one hot encoding method that resulted in new features denoting each type of species, with a 0 or 1 indicating if that was the species for the row (0 is no, 1 is yes).

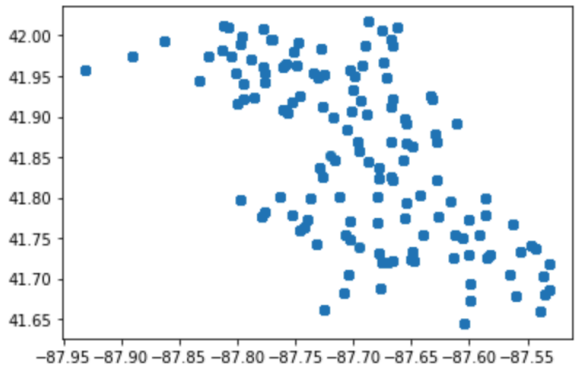
Joined Dataframe

- I performed an inner join on the two datasets using the index, so that they were joined on common dates. I used an inner join specifically so that every row was full as well as every feature, minimizing the missing values and ensuring that every record had as much information as possible.

- Following the join, I used the longitude and latitude columns to split the data by region, so as to have a more descriptive way of describing the locations. I binned the different locations into 9 regions, and adjusting some of the values to receive the values of 1-9 for the added region column.

```
plt.scatter(df['Longitude'], df['Latitude'])
```

```
<matplotlib.collections.PathCollection at 0x11f313190>
```




```
# Divide Longitude and Latitude into regions

bin_labels = [1, 2, 3]
df['Lat_Region'] = pd.qcut(df['Latitude'], q=3, labels=bin_labels)
df['Long_Region'] = pd.qcut(df['Longitude'], q=3, labels=bin_labels)
df.head()
```

```
# The distribution of the regions looks good, turn both columns into one Region and
# drop the others. Turn this column into values 1-9.

df['Lat_Region'] = pd.to_numeric(df['Lat_Region'])
df['Long_Region'] = pd.to_numeric(df['Long_Region'])
df['Region'] = df.Lat_Region.astype(str) + df.Long_Region.astype(str)
df.drop(['Lat_Region', 'Long_Region'], axis=1, inplace=True)
df['Region'].replace({'11':1, '12':2, '13':3, '21':4, '22':5, '23':6, '31':7, '32':8,
                      '33':9}, inplace=True)
df.head()
```

- The final addition I made before the exploratory analysis was creating lagged variables for each numeric weather condition. I used lag 1, 7, 14, and 21 for this. The purpose of this was to see if the weather in the previous day,

previous week, 2 weeks prior, or 3 weeks prior had an affect on whether or not the virus was found. I would later go on to determine the correlation between each lagged instance and the target variable.

```
# DataFrame is almost ready for EDA. Time to create lag variables for all of the weather columns.

lags = [1, 7, 14, 21]
df_lagged_weather = pd.DataFrame(data=df[['Tmax', 'Tmin', 'Tavg', 'DewPoint',
       'WetBulb', 'Heat', 'Cool', 'Depth', 'SnowFall', 'PrecipTotal',
       'StnPressure', 'SeaLevel', 'ResultSpeed', 'ResultDir', 'AvgSpeed']])
for column in df_lagged_weather.columns:
    for lag in lags:
        df_lagged_weather[str(column) + '_lag_' + str(lag)] = df_lagged_weather[column].shift(lag)
```

- After this, the data frame had 90 features and was ready for some exploratory analysis.

III. Exploratory Data Analysis

- There were three main things I needed to focus on to accomplish the goal of this project:

    1. Which species of mosquito carries the virus?
    2. Which areas of the city is the virus mostly found?
    3. Do the weather patters on a particular day indicate any correlation with the virus being present?

- First, I subset the dataframe ‘df’ on the ‘WnvPresent’ feature, indicated as ‘df_WnvPresent’, so as to have all the information for instances in which the virus was found. There were 551 entires in which

```
# Which species of mosquito carries the virus?
# Here I will subset the data on WnvPresent and investigate the species.

df_WnvPresent = df.loc[df['WnvPresent'] == 1]
df_WnvPresent.shape
```

```
(551, 90)
```

```
# There are 551 cases of the virus found. Let's see which species are involved.

df_WnvPresent[['Species_CULEX ERRATICUS', 'Species_CULEX PIPIENS',
       'Species_CULEX PIPIENS/RESTUANS', 'Species_CULEX RESTUANS',
       'Species_CULEX SALINARIUS', 'Species_CULEX TARSALIS',
       'Species_CULEX TERRITANS']].sum(axis=0)
```

```
Species_CULEX ERRATICUS            0
Species_CULEX PIPIENS            240
Species_CULEX PIPIENS/RESTUANS   262
Species_CULEX RESTUANS            49
Species_CULEX SALINARIUS           0
Species_CULEX TARSALIS             0
Species_CULEX TERRITANS            0
dtype: int64
```

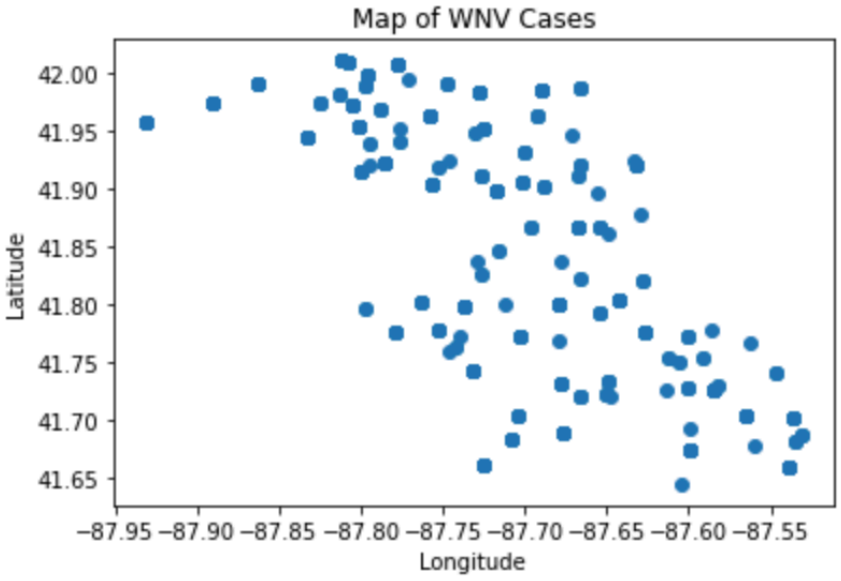
the virus was found, which was about 5% of the data.

- Being as that only 3 species were found to have the virus, I subsetted the data frame yet again into one containing just the species that had the virus. Plotting the data helped show the distribution of the virus amongst the species.

- I followed this by using df_WnvPresent to plot the regions in which the virus is found, as well as their value counts to better understand the plot.

```
# Based on this information, let's see what region of the city contains the most of these
# species.

df_WnvPresent['Region'].value_counts()
```

```
7    200
3    118
5     60
4     44
6     39
2     38
8     37
1     14
9      1
Name: Region, dtype: int64
```



- I then looked at answering my third question, regarding the weather. It was believed by Chicago Department of Public Health that warmer and wetter weather led to the virus being spread easier. I looked at statistics for days where the virus was found versus it not being found, and found that this assumption had some logical basis.

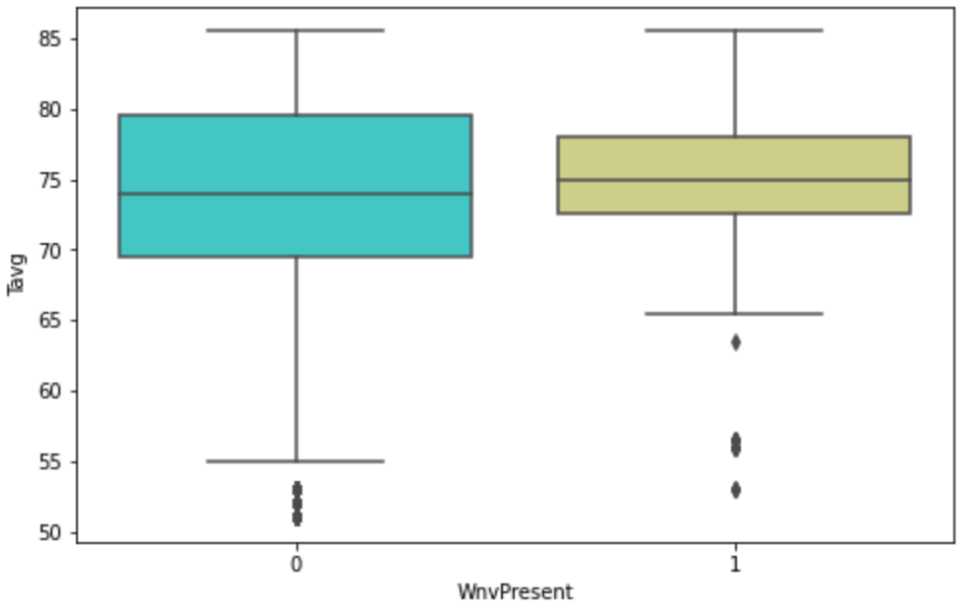
First 7 features of when the virus was found…

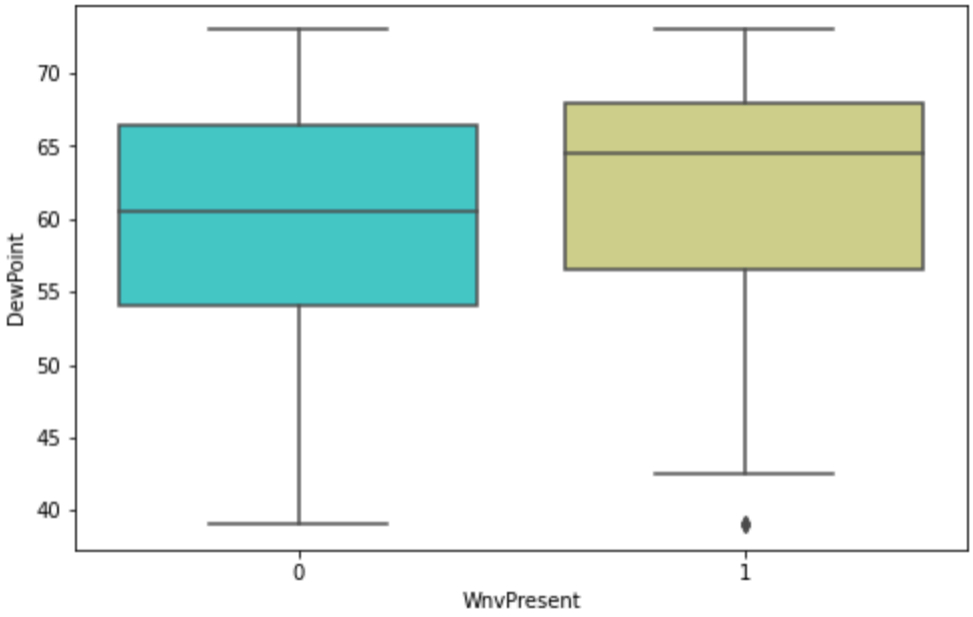
	Tmax	Tmin	Tavg	DewPoint	WetBulb	Heat	Cool
count	551.000000	551.000000	551.000000	551.000000	551.000000	551.000000	551.000000
mean	83.558076	65.995463	75.034483	62.530853	67.080762	0.243194	10.277677
std	6.161042	5.606123	5.302288	6.700472	5.312619	1.506900	4.564590
min	58.500000	46.000000	53.000000	39.000000	47.000000	0.000000	0.000000
25%	80.500000	62.500000	72.500000	56.500000	63.500000	0.000000	7.500000
50%	83.000000	67.500000	75.000000	64.500000	69.000000	0.000000	10.000000
75%	88.000000	69.500000	78.000000	68.000000	70.500000	0.000000	13.000000
max	96.500000	77.500000	85.500000	73.000000	76.000000	12.000000	20.500000

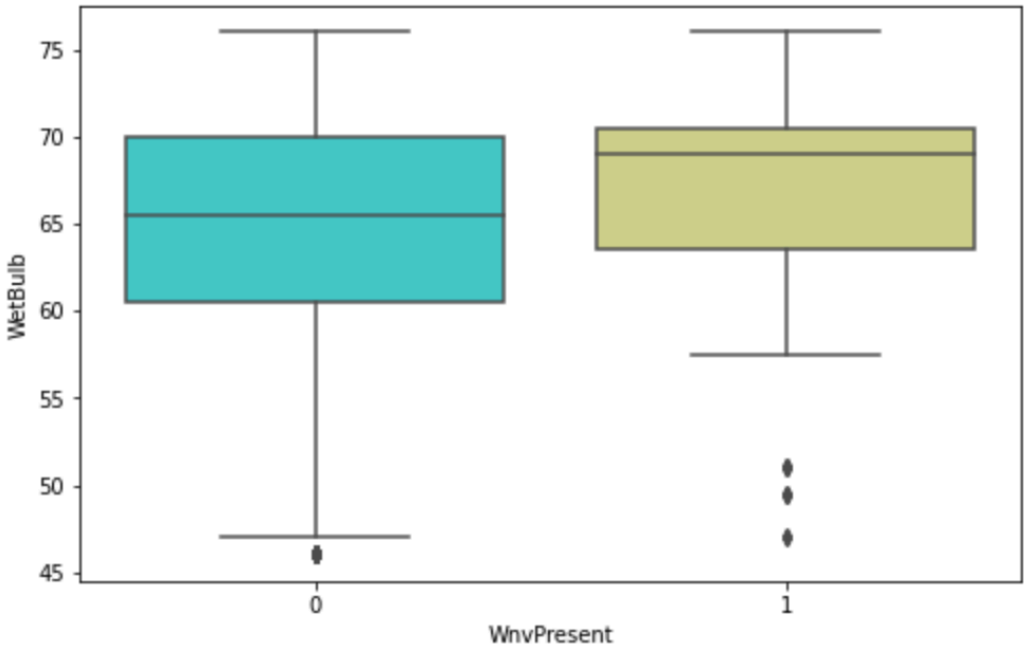
…versus when the virus was not found.

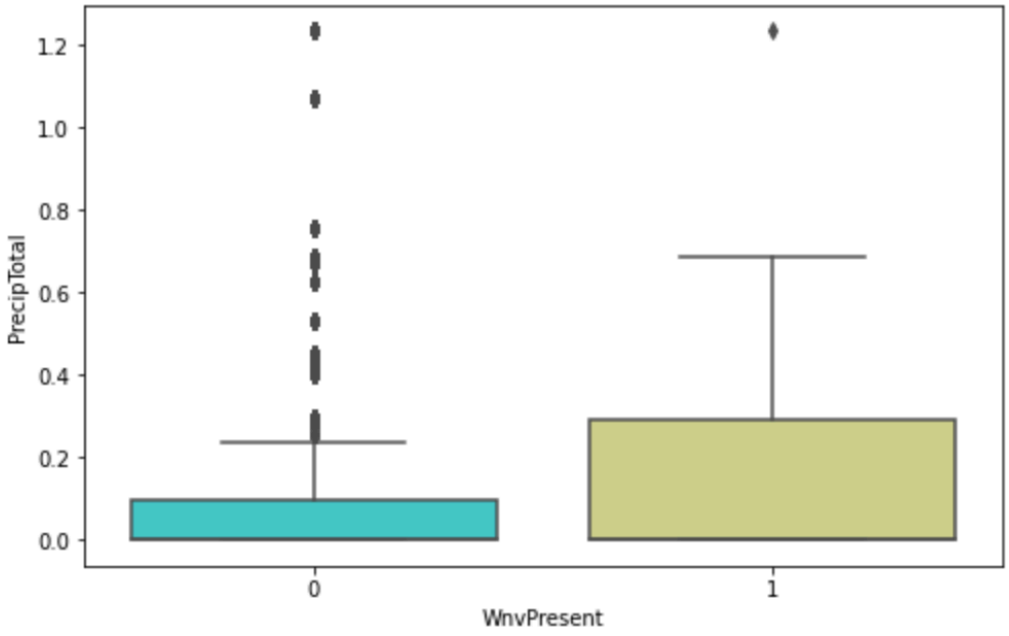
	Tmax	Tmin	Tavg	DewPoint	WetBulb	Heat	Cool
count	9862.000000	9862.000000	9862.000000	9862.000000	9862.000000	9862.000000	9862.000000
mean	81.832640	63.528950	72.935409	59.463141	64.653671	0.901693	8.837102
std	8.316824	7.558852	7.535232	7.762816	6.724382	2.708274	5.790087
min	57.000000	44.500000	51.000000	39.000000	46.000000	0.000000	0.000000
25%	78.000000	59.000000	69.500000	54.000000	60.500000	0.000000	4.500000
50%	83.000000	65.500000	74.000000	60.500000	65.500000	0.000000	9.000000
75%	88.000000	69.500000	79.500000	66.500000	70.000000	0.000000	14.500000
max	96.500000	77.500000	85.500000	73.000000	76.000000	14.000000	20.500000

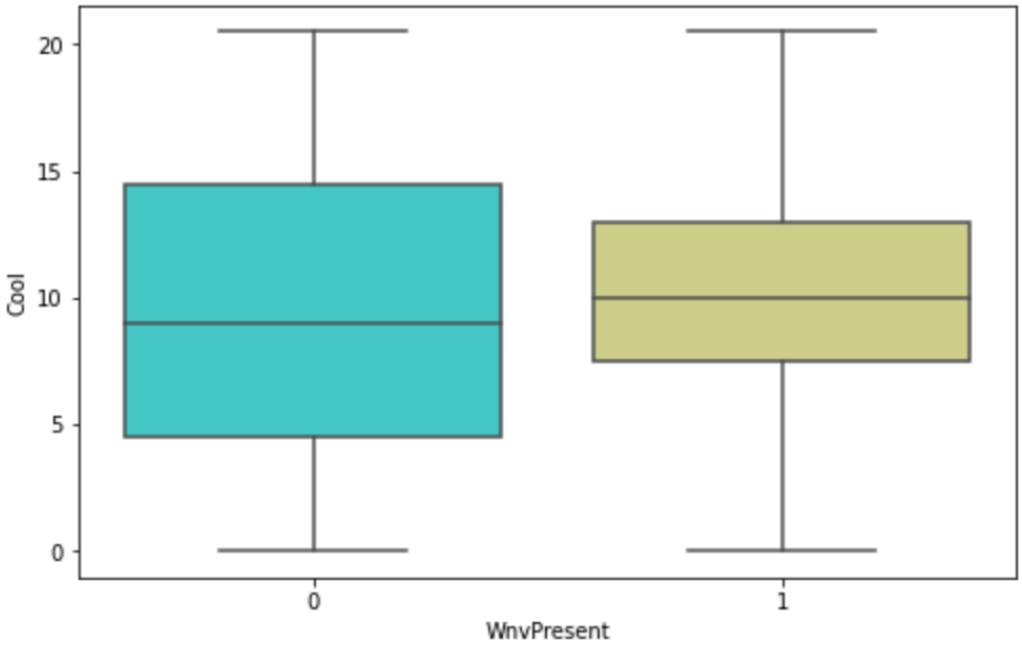
- To further investigate this phenomenon, I plotted the distributions of the main weather variables, comparing the instances when the virus was present versus when it was not. What I found was that it is definitely true that hotter warmer weather is more associated with the presence of the virus.






WetBulb
75
70
65
60
55
50
45
0
1
WnvPresent


PrecipTotal
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1
WnvPresent


Cool
20
15
10
5
0
0
1
WnvPresent

IV. Feature Engineering

- Looking at the data frame, I realized there were still some useless variables. Those were the address and street columns, due to me already having one variable for the location, that being region. I needed my dataframe to be entirely numeric for the modeling process. I also dropped the trap column because it was similar to the address in the sense that each address had unique trap, so it was also unnecessary with the presence of the region variable.

- Because of the nature of lagged variables, there will be rows with NaN values, referring to days in which there was no data at the beginning of the dataframe. In other words, the data started on 5/29/2007, so for the lag 1, 7, 14, and 21 variables on that day, they were entirely NaN because there was no data for the previous weeks. Because of this, I dropped the rows with the remaining NaN values so as to not interfere with the weather data through imputation.

- I had a final working dataframe that I used for my train and test split, opting for an 8:2 ratio for the train and test split respectively.

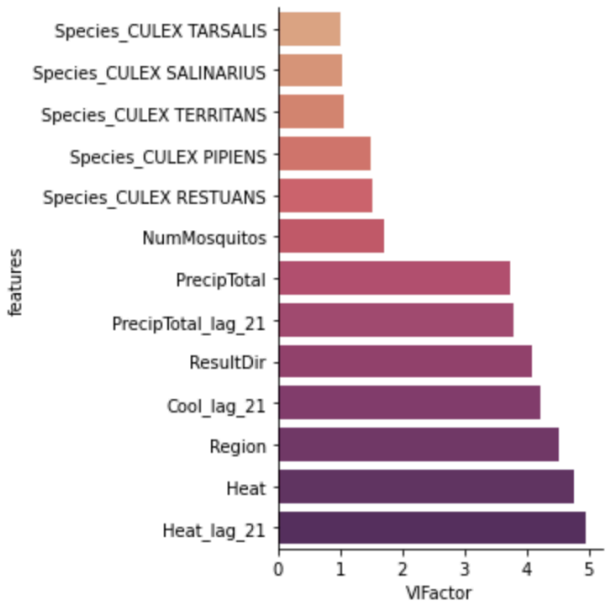
```
# Split data into train and test sets

y = df['WnvPresent']
X = df.drop('WnvPresent', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state=42, stratify=y)
```

- For the sake of dimensionality reduction, I performed information value analysis through means of the variance inflation factor (VIF). VIF is used to weed out multicollinearity in a regression analysis. This basically means that the predictor variables are correlated with one another, which can impede the performance of the model. The VIF estimates how much of the variance of a model is inflated due to this multicollinearity. I imported the following function, it compares the variables against one another and checks their VIF threshold.

If the value is 1, they are not correlated. If it is between 1 and 5, they are moderately correlated. If it is greater than 5, they are highly correlated, so anything above 5 was removed from the data.

```
# Information Value Analysis via the Variance Inlfation Factor.

def iterate_vif(df, vif_threshold=5, max_vif=6):
    count = 0
    while max_vif > vif_threshold:
        count += 1
        print("Iteration # "+str(count))
        vif = pd.DataFrame()
        vif["VIFactor"] = [variance_inflation_factor(df.values, i) for i in range(df.shape[1])]
        vif["features"] = df.columns

        if vif['VIFactor'].max() > vif_threshold:
            print('Removing %s with VIF of %f' % (vif[vif['VIFactor'] ==
                vif['VIFactor'].max()]['features'].values[0], vif['VIFactor'].max()))
            df = df.drop(vif[vif['VIFactor'] == vif['VIFactor'].max()]['features'].values[0], axis=1)
            max_vif = vif['VIFactor'].max()
        else:
            print('Complete')
            break
    return df, vif.sort_values('VIFactor')

final_df, final_vif = iterate_vif(X_train)
```

- This function iterated 64 times, leaving me with final_df (the data frame with only important variables left in it) and final_vif (a dataframe with each important feature and its VIF score). I sorted final_vif by the VIFactor, and as you can see the species features are the least correlated with the predictor variable. There were some NaN values in the dataframe, particularly pertaining to the depth and snowfall features, so I removed those entirely and was left with 12 features for modeling.



	VIFactor	features
0	1.007717	Species_CULEX TARSALIS
1	1.020086	Species_CULEX SALINARIUS
2	1.058026	Species_CULEX TERRITANS
3	1.500469	Species_CULEX PIPIENS
4	1.512638	Species_CULEX RESTUANS
5	1.705421	NumMosquitos
6	3.735686	PrecipTotal
7	3.794456	PrecipTotal_lag_21
8	4.090379	ResultDir
9	4.209629	Cool_lag_21
10	4.530008	Region
11	4.749748	Heat
12	4.963788	Heat_lag_21

- Being as that I needed to predefine the train test sets for the information value analysis, I subset X_train and X_test on final_df, to ensure only the important, uncorrelated features were used for modeling.

```
# As the final pre-processing step, I will subset X_train and X_test on final_df, which
# shows which columns are of use.

X_train = X_train[final_df.columns]
X_train.columns
```

```
Index(['NumMosquitos', 'Species_CULEX PIPIENS', 'Species_CULEX RESTUANS',
       'Species_CULEX SALINARIUS', 'Species_CULEX TARSALIS',
       'Species_CULEX TERRITANS', 'Heat', 'PrecipTotal', 'ResultDir', 'Region',
       'Heat_lag_21', 'Cool_lag_21', 'PrecipTotal_lag_21'],
      dtype='object')
```

```
X_test = X_test[final_df.columns]
X_test.columns
```

```
Index(['NumMosquitos', 'Species_CULEX PIPIENS', 'Species_CULEX RESTUANS',
       'Species_CULEX SALINARIUS', 'Species_CULEX TARSALIS',
       'Species_CULEX TERRITANS', 'Heat', 'PrecipTotal', 'ResultDir', 'Region',
       'Heat_lag_21', 'Cool_lag_21', 'PrecipTotal_lag_21'],
      dtype='object')
```

V. Modeling

- For the modeling process, I have chosen to use four different models. These models consist of a random forrest model, two logistic regression models, and an XG boost model. To implement and evaluate the metrics, I have used the following function. The purpose of the function is to implement the model using a pipeline that scales the data with StandardScaler, does a 10-fold randomized cross validation search to optimize the parameters, fits and predicts the model and data, produces a classification report and confusion matrix, and finally, plots the receiver operating characteristic (ROC), and determines the area under the curve (AUC), which is my main metric given that this is a binary classification problem.

```
def model_evaluation(model,params,avg):
    '''Creates model, classification report, and plots the roc curve of the model.'''
    pipe = make_pipeline(StandardScaler(),model)
    model_ran = RandomizedSearchCV(pipe,params,cv=10, n_jobs=-1, scoring = 'roc_auc',random_state = 42)
    model_ran = model_ran.fit(X_train,y_train)
    y_pred = model_ran.predict(X_test)
    y_pred_proba = model_ran.predict_proba(X_test)[:,1]
    f1 = f1_score(y_test, y_pred, average= avg)
    cm = confusion_matrix(y_test, y_pred)
    model_roc_auc = roc_auc_score(y_test, y_pred_proba)

    print('F1-score: ' , round(f1,4))
    print("Best Score: " , round(model_ran.best_score_,4))
    print("Test ROC AUC:", round(model_roc_auc, 4), '\n')
    print("Best Parameters: " , model_ran.best_params_)
    print("Confusion Matrix: " ,'\n', cm, '\n')
    print("Classsification Report: ",'\n', classification_report(y_test, y_pred))

    y_pred_proba=model_ran.predict_proba(X_test)[:,1]
    fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

    plt.figure()
    plt.plot(fpr, tpr, label= 'Model (area = %0.2f)' % model_roc_auc)
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic')
    plt.legend(loc="lower right")
```

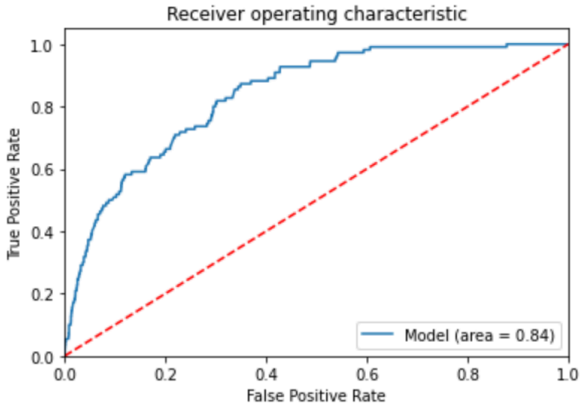
Random Forrest

```
params= {'randomforestclassifier__n_estimators':[500, 600, 800, 1000, 1500, 1800, 2000],
             'randomforestclassifier__max_features':['auto', 'sqrt', 'log2'],
             'randomforestclassifier__max_depth':np.arange(4,20),
             'randomforestclassifier__criterion':['gini', 'entropy'],
            'randomforestclassifier__min_samples_split' : [2, 3, 4] }
avg= 'binary'

model_evaluation(RandomForestClassifier(), params, avg)
```

```
F1-score:  0.0
Best Score:  0.85
Test ROC AUC: 0.8372

Best Parameters:  {'randomforestclassifier__n_estimators': 2000, 'randomforestclassifier__min
_samples_split': 3, 'randomforestclassifier__max_features': 'sqrt', 'randomforestclassifier__
max_depth': 8, 'randomforestclassifier__criterion': 'entropy'}
Confusion Matrix:
 [[1969    0]
 [ 110    0]]
```

```
Classsification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97      1969
           1       0.00      0.00      0.00       110

    accuracy                           0.95      2079
   macro avg       0.47      0.50      0.49      2079
weighted avg       0.90      0.95      0.92      2079
```

```
/opt/anaconda3/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1221: Undefined
MetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no p
redicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```



Logistic Regression

Model 1:

```
# First logistic regression model

logit1_params = {'logisticregression__penalty':['l1','l2','elasticnet'],
                 'logisticregression__C':np.arange(0.5, 100.0, 0.5) }

logit1 = LogisticRegression()
model_evaluation(logit1, logit1_params, avg)
```

```
F1-score:  0.0
Best Score:  0.7623
Test ROC AUC: 0.7624

Best Parameters:  {'logisticregression__penalty': 'l2', 'logisticregression__C': 2.0}
Confusion Matrix:
 [[1969    0]
 [ 110    0]]
```

```
Classsification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97      1969
           1       0.00      0.00      0.00       110

    accuracy                           0.95      2079
   macro avg       0.47      0.50      0.49      2079
weighted avg       0.90      0.95      0.92      2079
```

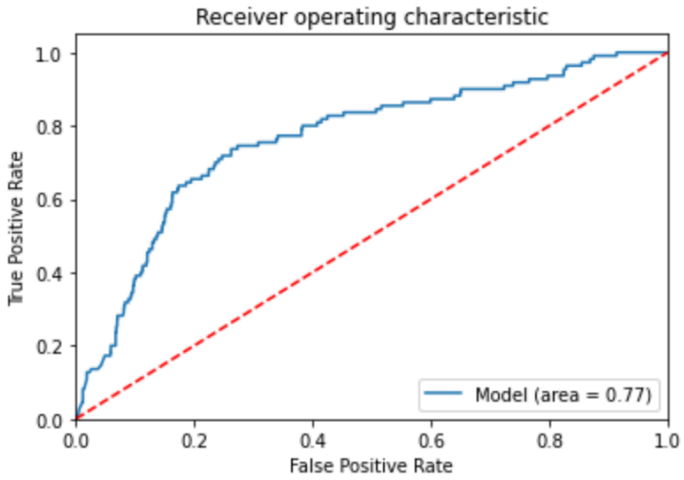
```
/opt/anaconda3/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1221: Undefined
MetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no p
redicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Model 2:

```
# Second logistic regression model

logit2_params = {'logisticregression__penalty':['l1','l2','elasticnet'],
                 'logisticregression__class_weight':['balanced'],
                'logisticregression__C':np.arange(0.5, 100.0, 0.5)}
logit2 = LogisticRegression()

model_evaluation(logit2, logit2_params, avg)
```

```
F1-score:  0.2293
Best Score:  0.769
Test ROC AUC: 0.7651

Best Parameters:  {'logisticregression__penalty': 'l2', 'logisticregression__class_weight':
'balanced', 'logisticregression__C': 2.0}
Confusion Matrix:
 [[1469  500]
 [  31   79]]
```

```
Classsification Report:
              precision    recall  f1-score   support

           0       0.98      0.75      0.85      1969
           1       0.14      0.72      0.23       110

    accuracy                           0.74      2079
   macro avg       0.56      0.73      0.54      2079
weighted avg       0.93      0.74      0.81      2079
```



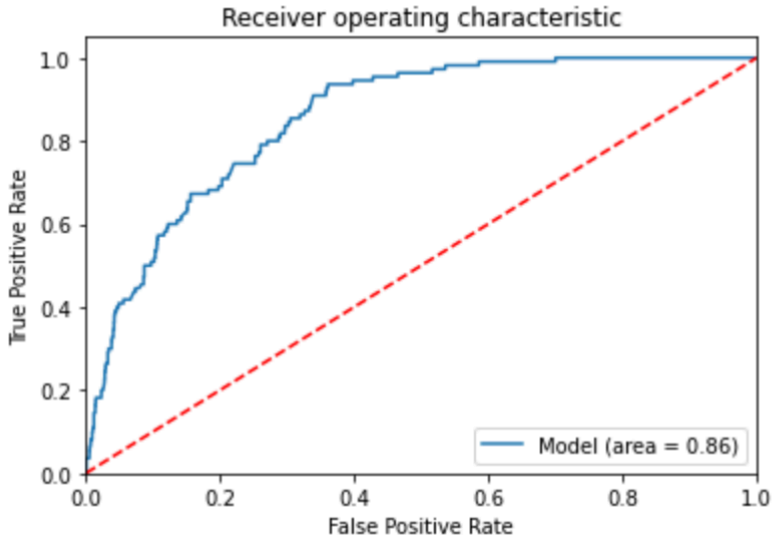
XG Boost

```python
from xgboost import XGBClassifier


xg_params= {'xgbclassifier__nthread':[10], # when use hyperthread, xgboost may become slower
              'xgbclassifier__objective':['binary:logistic'],
              'xgbclassifier__learning_rate':[0.05], # so called `eta` value
              'xgbclassifier__max_depth':[2],
              'xgbclassifier__min_child_weight':[1],
              'xgbclassifier__silent':[1],
              'xgbclassifier__subsample':[.89,.91,.895],
              'xgbclassifier__colsample_bytree':[.58,.585],
              'xgbclassifier__n_estimators':[1000],
             'xgbclassifier__seed':[27]} # number of trees

xgb = XGBClassifier()

model_evaluation(XGBClassifier(), xg_params, avg)
```

```
[15:59:30] WARNING: /private/var/folders/rg/vb03dfw167593y158dqrs8200000gn/T/pip-install-h2an
l8uv/xgboost/build/temp.macosx-10.9-x86_64-3.8/xgboost/src/learner.cc:1095: Starting in XGBoo
st 1.3.0, the default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the old behavi
or.
F1-score:  0.0678
Best Score:  0.8621
Test ROC AUC: 0.8554

Best Parameters:  {'xgbclassifier__subsample': 0.89, 'xgbclassifier__silent': 1, 'xgbclassifi
er__seed': 27, 'xgbclassifier__objective': 'binary:logistic', 'xgbclassifier__nthread': 10,
'xgbclassifier__n_estimators': 1000, 'xgbclassifier__min_child_weight': 1, 'xgbclassifier__ma
x_depth': 2, 'xgbclassifier__learning_rate': 0.05, 'xgbclassifier__colsample_bytree': 0.58}
Confusion Matrix:
 [[1965    4]
 [ 106    4]]
```

Classsification Report:

	precision	recall	f1-score	support
0	0.95	1.00	0.97	1969
1	0.50	0.04	0.07	110
accuracy			0.95	2079
macro avg	0.72	0.52	0.52	2079
weighted avg	0.93	0.95	0.92	2079



- Looking at the receiver operating characteristics of each model, the corresponding area under the curve (denoted in the bottom right corner of the graph) is the highest for the XG boost model. The closer the area is to 1, the more accurate the model. This is the case because the goal is to maximize true positive rates. Looking at the precision, we see that when the model predicts that the virus is present, it is correct 95% of the time. Based on the

recall, we see that the model correctly identifies every instance (100%) of the virus being present. The F1 score is 0.97, which is very close to being perfect, and this tells us that we have low false positives and low false negatives, so we're correctly identifying real threats and not worrying too much about the false alarms. This is apparent when looking at the confusion matrix right above the classification report.

- Being as that the model was fit inside of a function, I then needed to fit it outside of the function to establish a global instance of the model for further evaluation. To do this, I used the contents of the model_evaluation function with the best found parameters for the model.

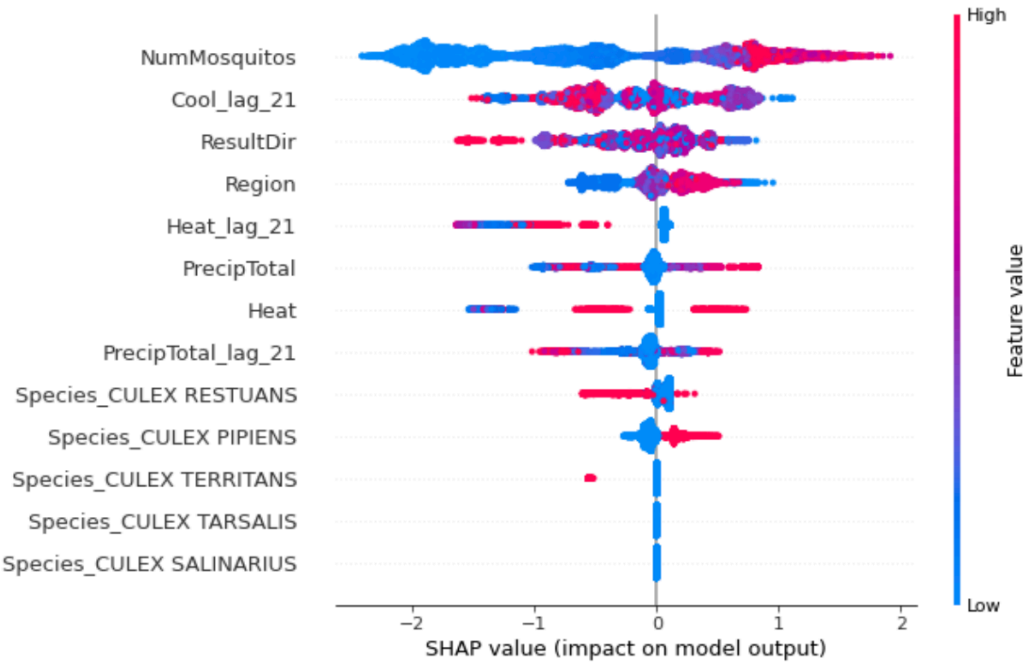
```
xgb = XGBClassifier(subsample=0.89,
                    silent=1,
                    seed=27,
                    objective= 'binary:logistic',
                    nthread=10,
                    n_estimators=1000,
                    min_child_weight=1,
                    max_depth=2,
                    learning_rate=0.05,
                    colsample_bytree=0.58)

xgb.fit(X_train,y_train)
y_pred = xgb.predict(X_test)
y_pred_proba = xgb.predict_proba(X_test)[:,1]

f1 = f1_score(y_test, y_pred, average= 'binary')
cm = confusion_matrix(y_test, y_pred)
model_roc_auc = roc_auc_score(y_test, y_pred_proba)
```

```
print('F1-score: ' , round(f1,4))
print("ROC AUC:", round(model_roc_auc, 4), '\n')
print("Confusion Matrix: " ,'\n', cm, '\n')
print("Classsification Report: ",'\n', classification_report(y_test, y_pred))
y_pred_proba=xgb.predict_proba(X_test)[:,1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

plt.figure()
plt.plot(fpr, tpr, label= 'Model (area = %0.2f)' % model_roc_auc)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
```

VI. SHAP Analysis

- **SHAP** stands for **SH**apley **A**dditive ex**P**lanations. It is used to interpret the impact of the features by analyzing how the model performs with and without the feature present.

Summary Plot

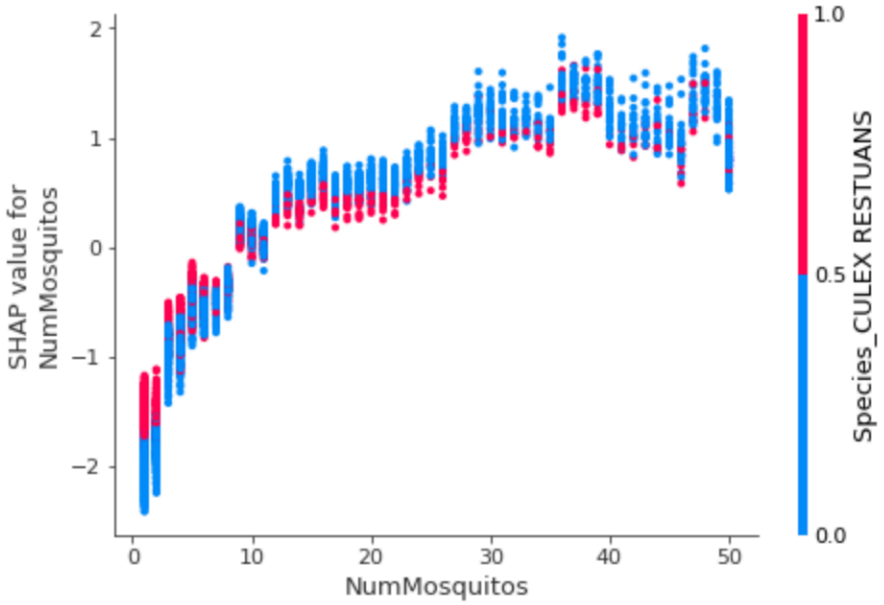


- A lot can be interpreted from this plot. Firstly, is the feature importance, which shows the features ranked in descending order. Here we can see that NumMosquitos is ranked the highest, while the species are ranked the lowest. Second, we are given the impact of the feature by the horizontal location of the bubbles. Given the x-values along the bottom axis, we can see if the impact is negative or positive. This ties into the next observation, which is the color. The red color indicates a high impact, and the blue color indicates a low one. This ties into the horizontal location because when we look at NumMosquitos, for example, we see that it has a very low negative impact with a fairly high positive impact.

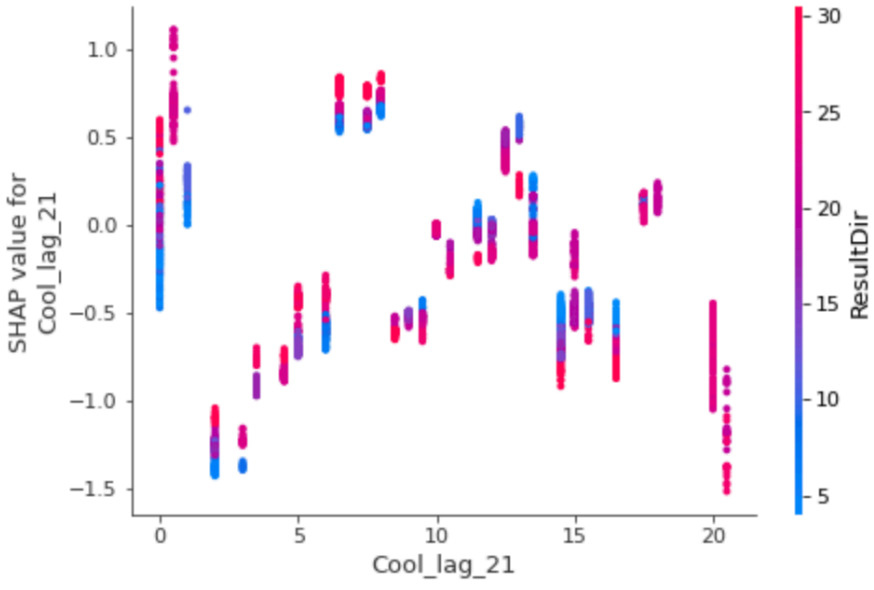
Dependence Plots

- The dependence plot shows the marginal effect one or two features have on the predicted outcome of a machine learning model. It tells whether the relationship between the target and a feature is linear, monotonic, or more complex. Using the dependence plot function automatically includes another variable that the chosen variable interacts with the most.

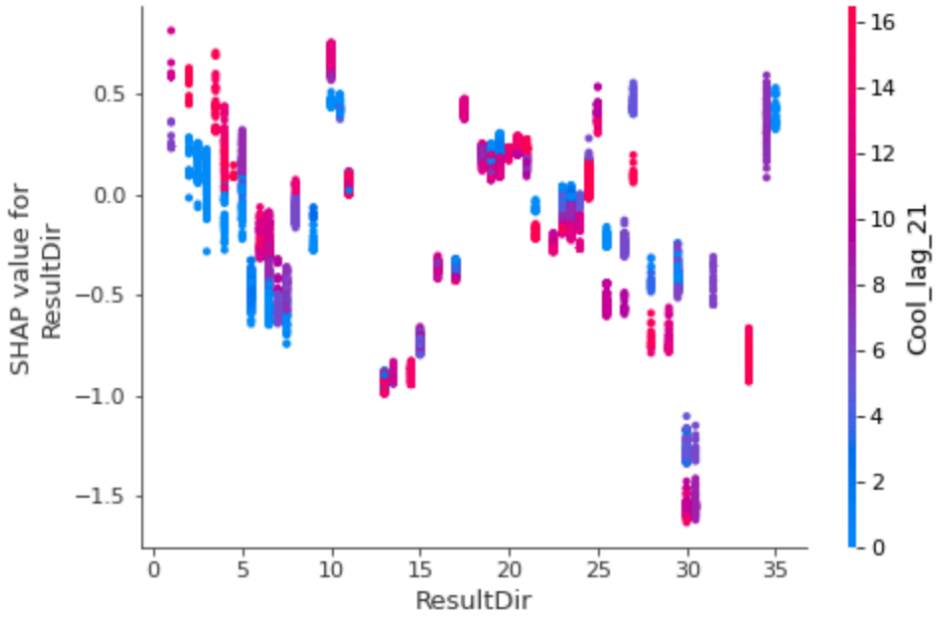
NumMosquitos:



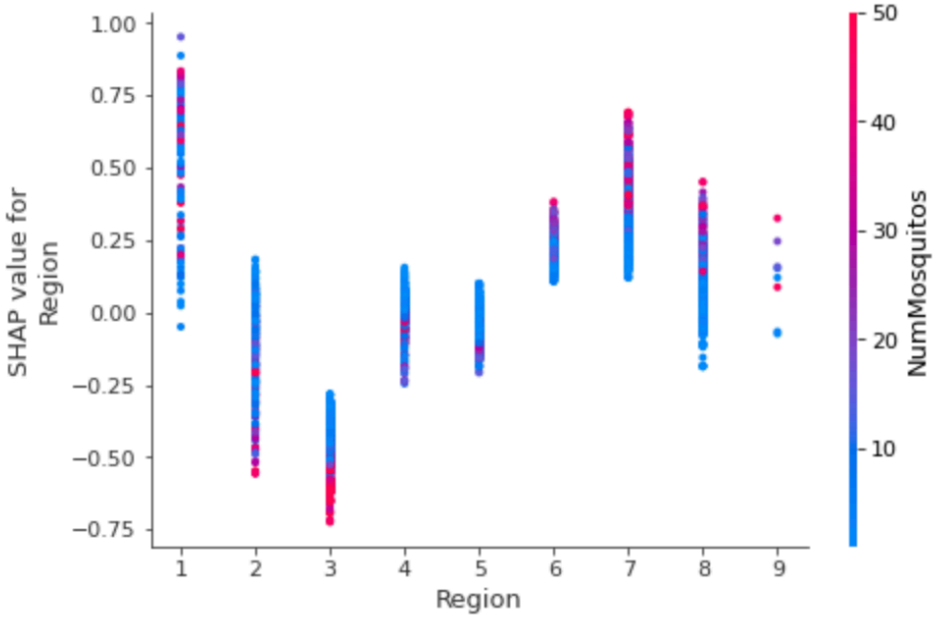
Cool_lag_21:



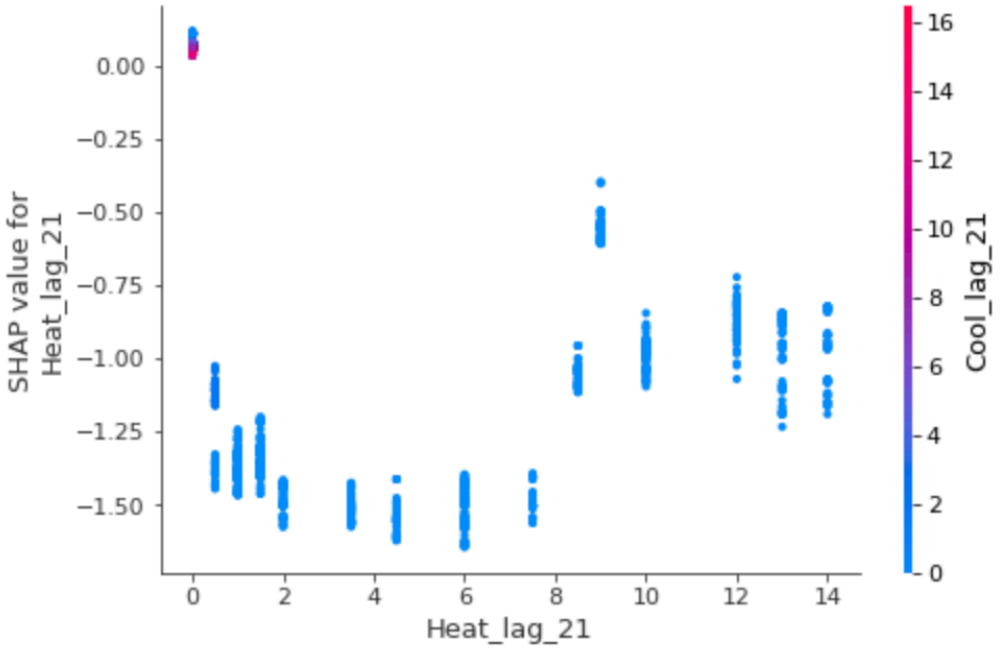
ResultDir:



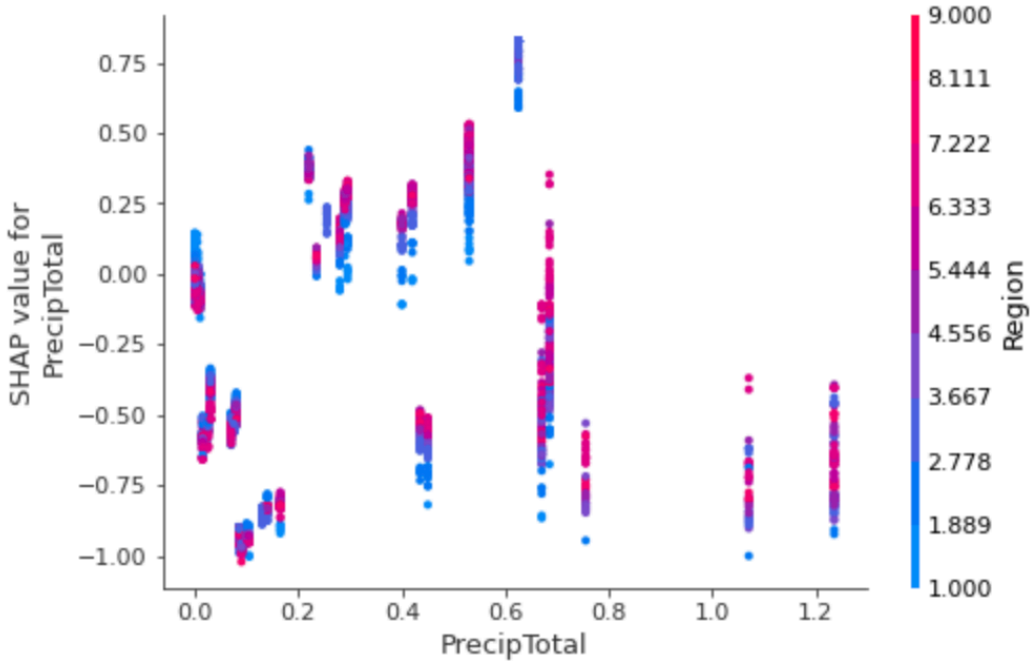
Region:



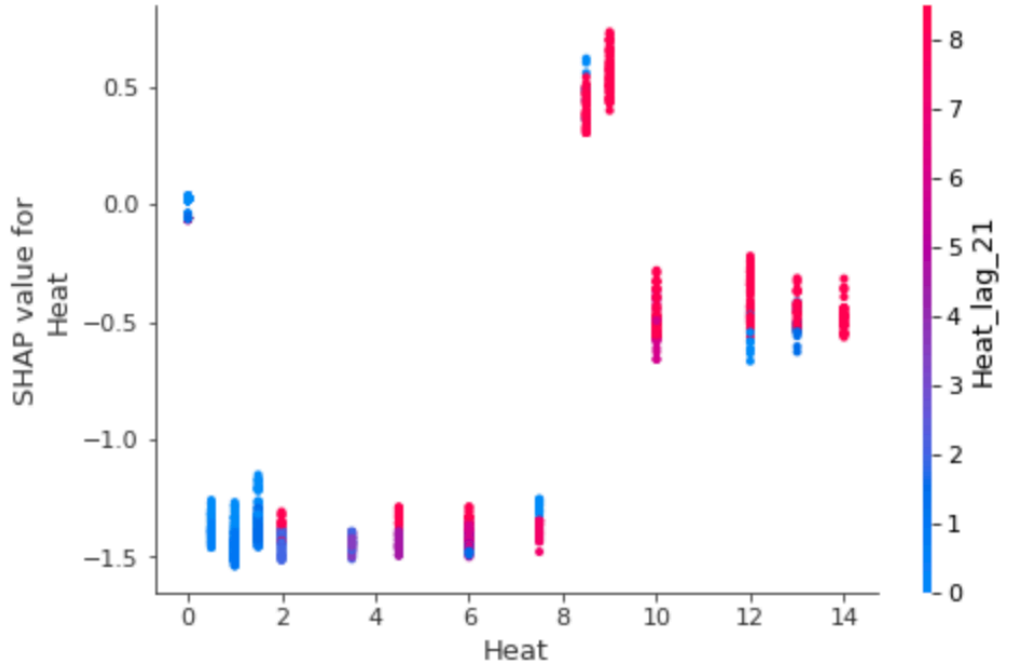
Heat_lag_21:



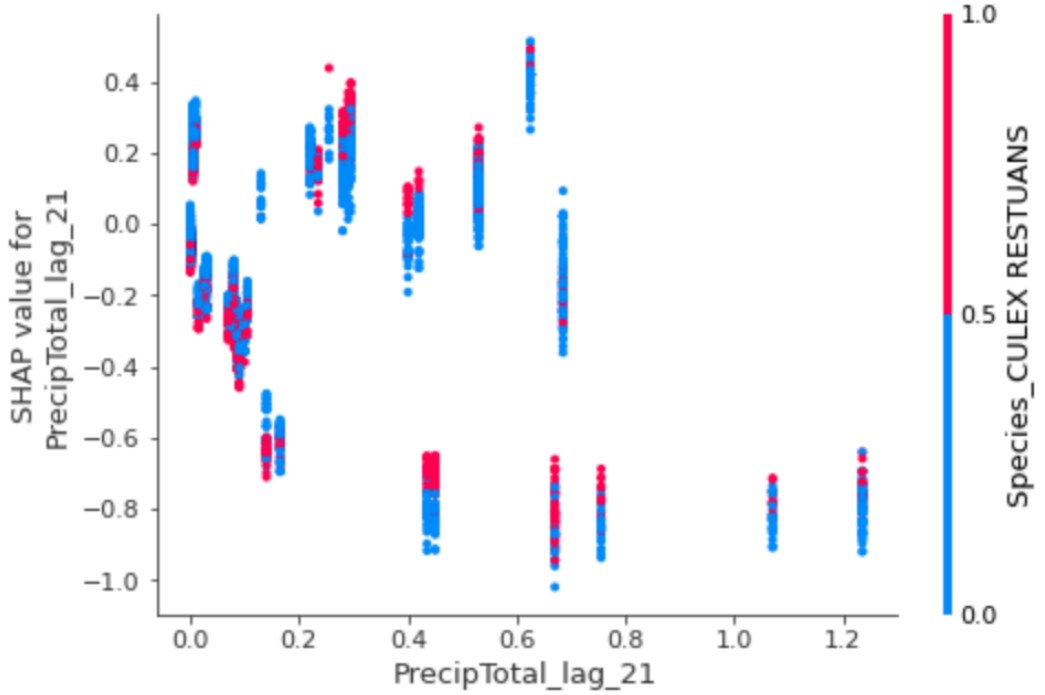
PrecipTotal:



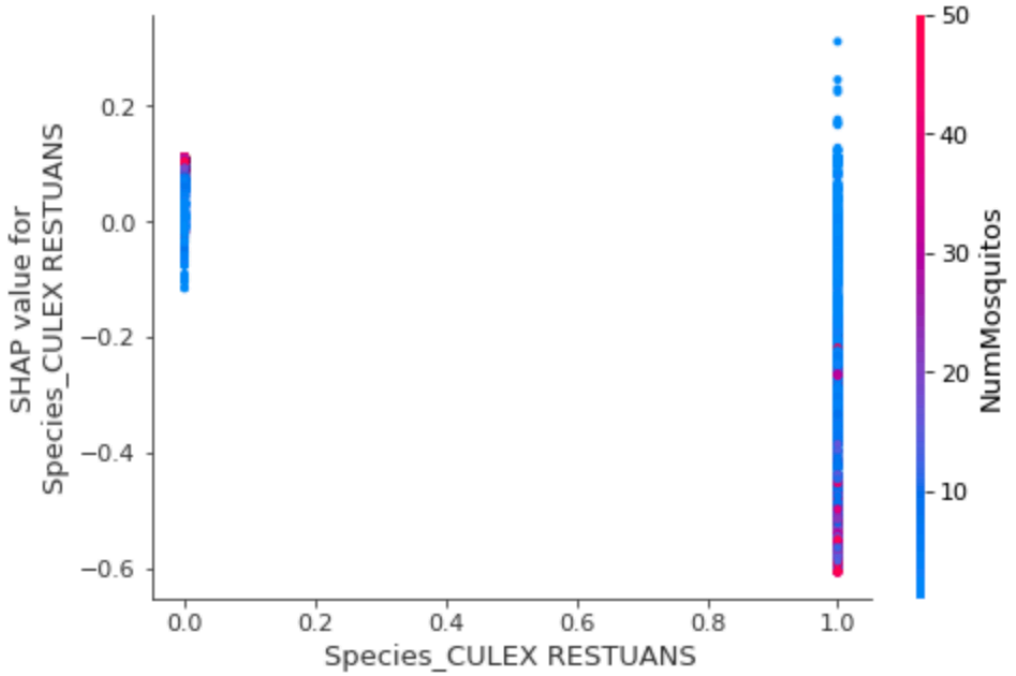
Heat:



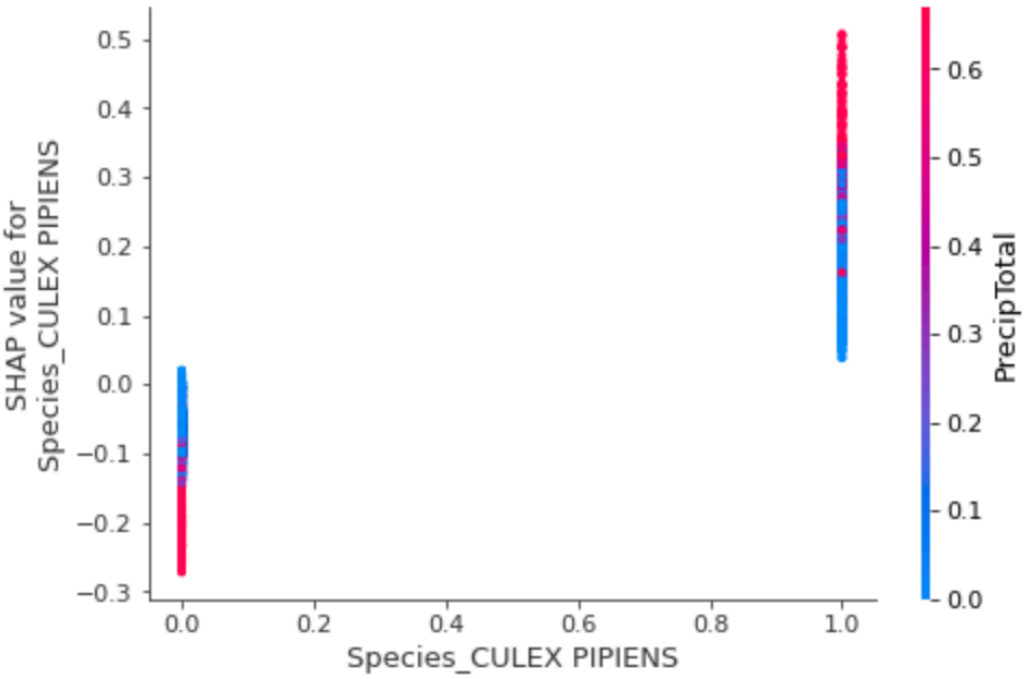
PrecipTotal_lag_21:



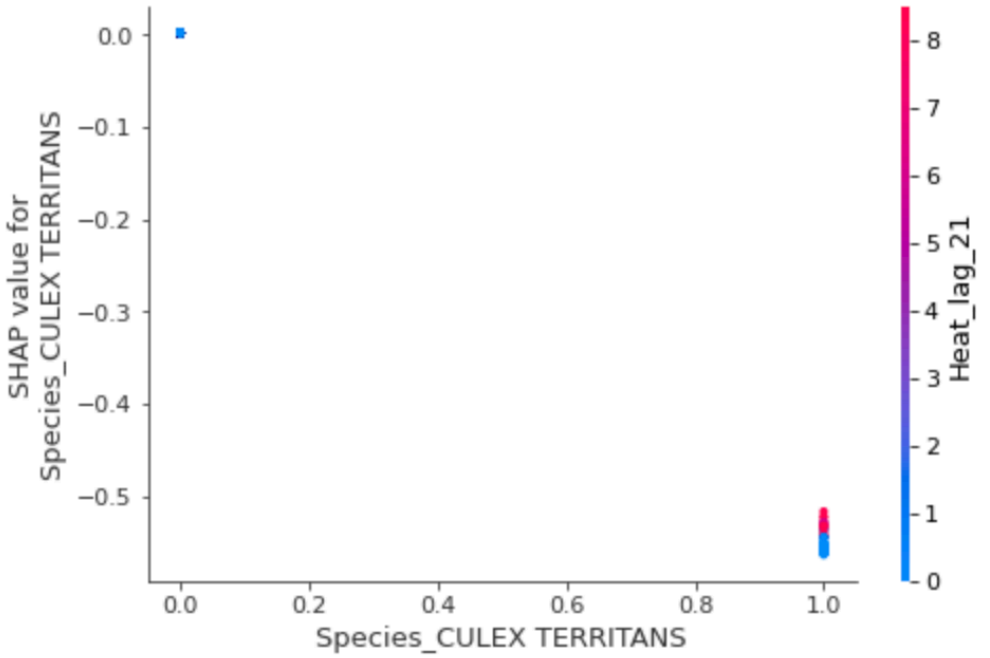
Species_CULEX RESTUANS:



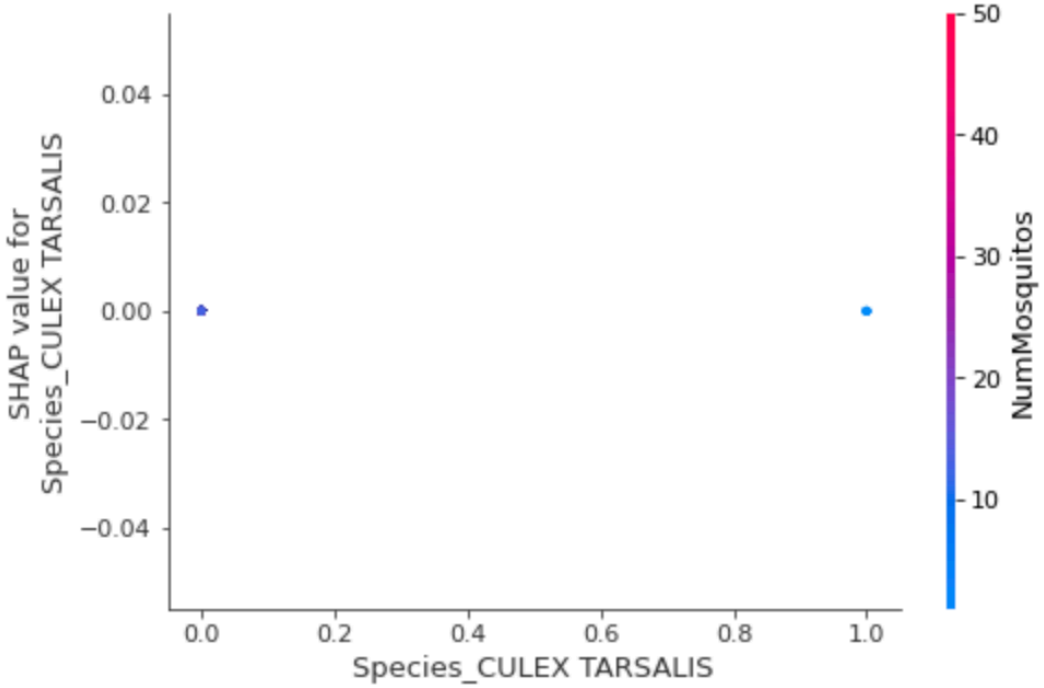
Species_CULEX PIPIENS:



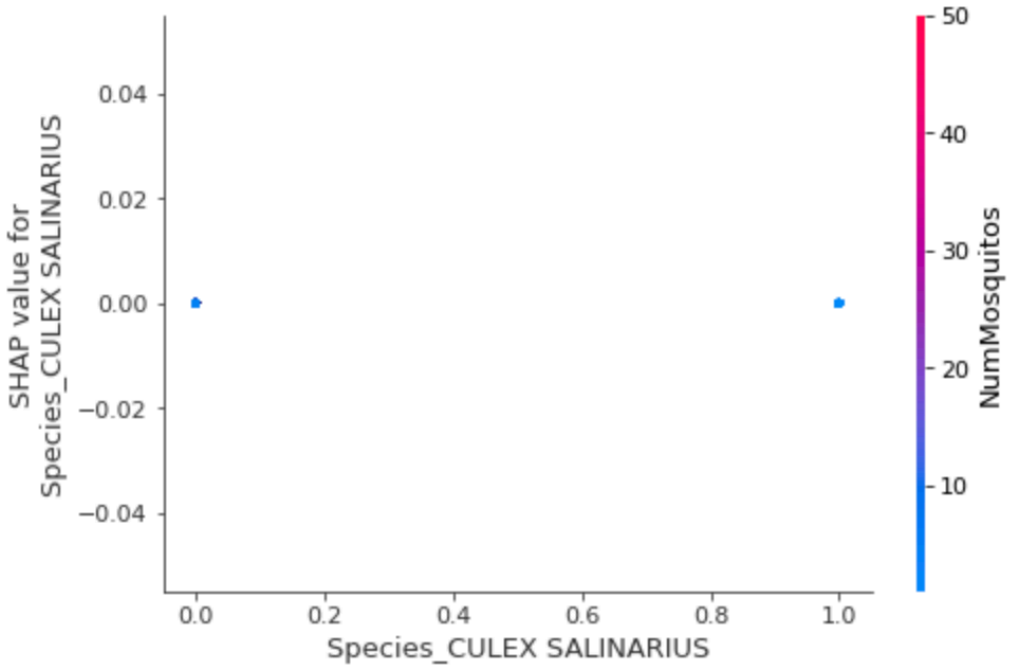
Species_CULEX TERRITANS:



Species_CULEX TARSALIS:



Species_CULEX SALINARIUS:



VII. Conclusion

Reviewing all of the exploratory analysis, a few conclusions can be made about the presence of the West Nile virus. First of all, the number of mosquitos plays a huge roll in how often the virus will be found. Although this is trivial, it does suggest that if the mosquito season brings in an influx of mosquitos, more resources are going to be necessary to help prevent the spread, simply due to higher volumes of possible carriers. Interestingly enough, the wind direction also had a pretty significant impact on the presence of the virus. My understanding of this is that it changes patterns for birds and mosquitos in terms of migrations, and can lead to them being found in different parts of the city at different times. Culex Pipiens is the species the city needs to track the most, because this is the species that has the highest chance of carrying the virus.

Moving forward, I have found that the Chicago Department of Public Health was correct in their assumptions that warmer and wetter weather is going to help the spread of the virus. Given the final 12 features I used to create models, and an XG boost model using the best found randomized parameters, it is

possible to predict the presence of the virus with fair certainty. Although not perfect, the metrics indicated that it is very likely to properly classify the virus. This is not the end of the fight to stop the spread of the virus, simply a better means of allocating resources to prevent more potential illness. Further efforts to entirely eliminate the virus would have to analyze the city's ability to stop the breeding of the afflicted species of mosquitos that carry it. This is not a small task, but it is worth looking into as the next step of buckling down and really ridding the city of the harmful virus.