Using Neural Networks and K-Means Clustering for Accurate Wildfire Condition Detection

By Gurik Mangat

Introduction

This Jupyter notebook is to be accompanied by the research paper written for this study. All code used for and written during this research has been presented here. This notebook will go over 2 main categories: "Dataset Construction" and "Neural Network Classifier Training and Testing", each of which will have numerous sub-categories.

Dataset Construction

1. Collecting Environmental Data From the OpenWeatherMap API

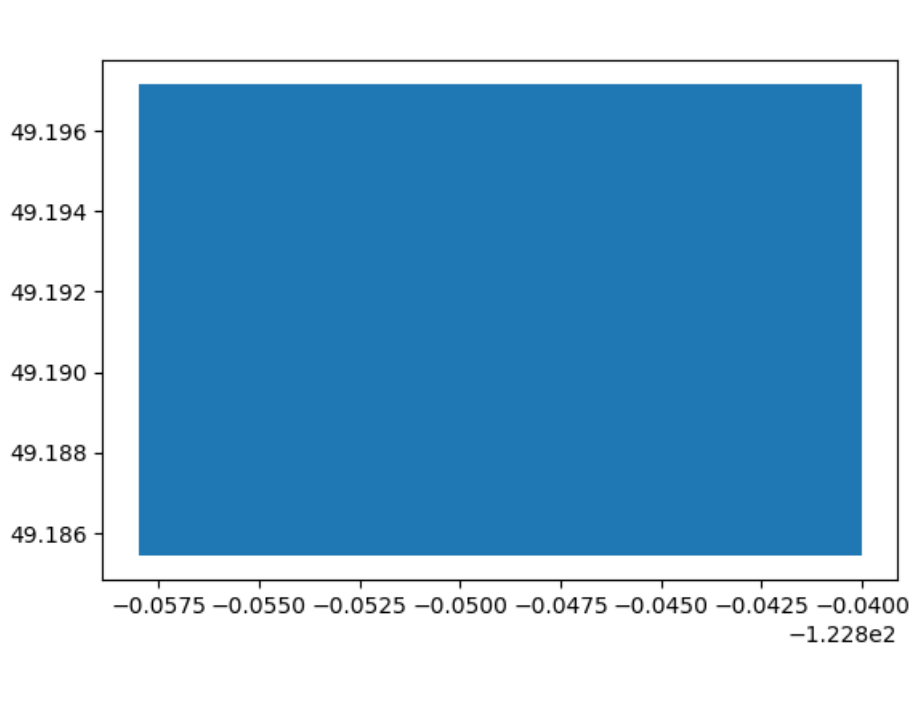
1.1 Constructing a 1 hectare polygon using an initial set of coordinates

In order to access accurate environmental conditions data, a one hectare polygon is constructed. The subsequent soil data and NDVI imagery is taken from this plot through the API.

In [75]:

```python
import pyowm
from pyowm.utils.geo import Polygon as GeoPolygon
from pyowm.commons.enums import ImageTypeEnum
from pyowm.agroapi10.enums import SatelliteEnum, PresetEnum
from pyowm.agroapi10.enums import PaletteEnum
import requests
from pprint import pprint
import geopandas as gpd
from shapely import geometry
import json

import warnings;
warnings.filterwarnings('ignore')


#Coordinates Script
lat = str(input("Enter Latitude: "))
lon = str(input("Enter Longitude: "))
lon_float = float(lon)
lat_float = float(lat)
gdf = gpd.GeoDataFrame(geometry=[geometry.Point(lon_float, lat_float)])
gdf.crs = {'init': 'epsg:4326'}
gdf = gdf.to_crs({'init': 'epsg:3857'})
gdf['geometry'] = gdf['geometry'].apply(lambda x: x.buffer(1000).envelope)
gdf = gdf.to_crs({'init': 'epsg:4326'})
gdf.plot()
data = json.loads(gdf.to_json())

fig= plt.figure()
plt.plot(range(10))
fig.savefig("save_file_name.pdf")
plt.close()

points = data['features'][0]['geometry']['coordinates'][0]
print("\nCoordinates of one hectare polygon: " + str(points))

print("\nPlot of 1 hectare area")
```

```
Enter Latitude: 49.1913
Enter Longitude: -122.8490

Coordinates of one hectare polygon: [[-122.8579831528412, 49.18542884199587], [-
122.84001684715881, 49.18542884199587], [-122.84001684715881, 49.197170461313], [-
122.8579831528412, 49.197170461313], [-122.8579831528412, 49.18542884199587]]

Plot of 1 hectare area
```



1.2 Using API to gather environmental data from the previously constructed polygon

This following code inputs the cordinates of the previously constructed 1-hectare polygon into the OpenWeatherMap API to harvest environmental data for dataset construction. This same code is also used in the app along with the polygon construction code in order to collect real time data from the inputted coordinates.

In [74]:

```python
def extractDigits(lst):
    return [el for el in lst]

lst = points
coordinates = [extractDigits(lst)]
lat_url = str(lat)
lon_url = str(lon)

API_key = " dae2b752022b59ac615eecbdd4b6a7c0"
base_url = "http://api.openweathermap.org/data/2.5/weather?"
Final_url = ("http://api.openweathermap.org/data/2.5/weather?
appid=dae2b752022b59ac615eecbdd4b6a7c0&"+ "&lat=" + lat_url + "&lon=" + lon_url)

weather_data = requests.get(Final_url).json()

temp = ("%.1f" % ((weather_data['main']['temp']) - 273))
wind_speed = ("%.1f" % ((weather_data['wind']['speed']) * 3.6))
humidity = ("%.1f" % (weather_data['main']['humidity']))


#soil temperature script
owm = pyowm.OWM('dae2b752022b59ac615eecbdd4b6a7c0')
mgr = owm.agro_manager()

polygon_points = GeoPolygon(coordinates)

polygon = mgr.create_polygon(polygon_points, 'api test 9')
polygon.id
polygon.user_id
soil = mgr.soil_data(polygon)
soil.polygon_id
soil.reference_time(timeformat='unix')
soil_temp = ("%.3f" %(soil.surface_temp(unit='celsius')))
soil_moist = ("%.3f" %(soil.moisture))

# Printing Data
print("\nData Collected on 6/14/2020 at 8am:")
print('\nHumidity : ',humidity)
print('\nWind Speed : ',wind_speed)
print('\nTemperature : ',temp)
print('\nSoil Temperature : ',soil_temp)
print('\nSoil Moisture : ',soil_moist)
```

```
Data Collected on 6/14/2020 at 8am:

Humidity :  91.0

Wind Speed :  4.8

Temperature :  10.8

Soil Temperature :  15.480

Soil Moisture :  0.227
```

2. Quantifying Fuel Moisture Using NDVI Imagery and K-Means Clustering

2.1 Example NDVI Imagery

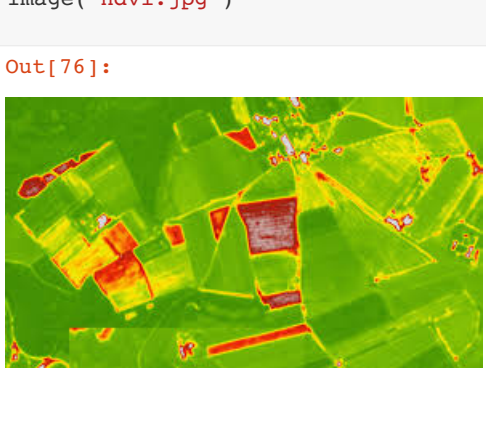
This following code will present two examples of NDVI images used from both fire scenarios.

Fire Conditions Not Present (High Fuel Moisture):

In [76]:

```python
from IPython.display import Image
Image("ndvi.jpg")
```

Out[76]:

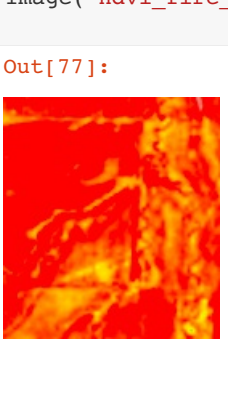


Fire Conditions Present (Low Fuel Moisture):

In [77]:

```python
from IPython.display import Image
Image("ndvi_fire_conditions.jpg")
```

Out[77]:



2.2 Analysing and Comparing K-Means Output

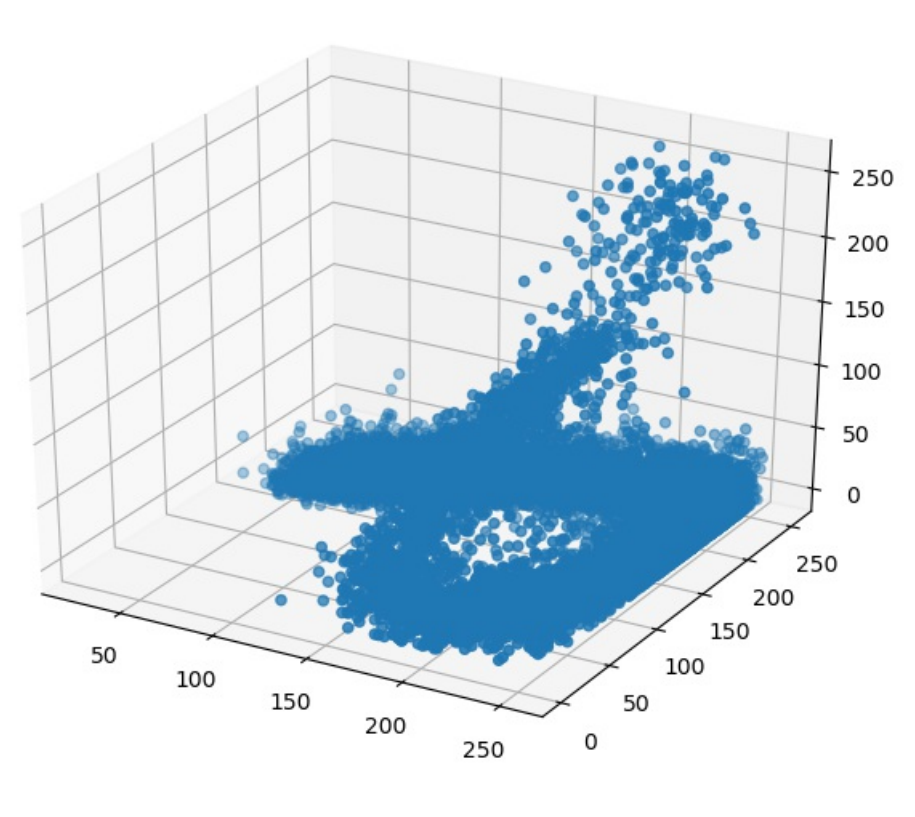
This following code runs the K-means clustering algorithm on the two previous example images and delivers the RGB values of the most dominant colour along with a graphical representation of the individual pixel data.

K-Means Clustering on "High Fuel Moisture" Imagery

In [55]:

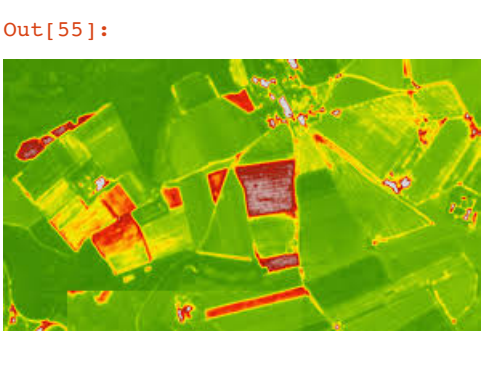
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import image as img
from sklearn.cluster import KMeans
import cv2


#Color Dominance Algorithm
def make_histogram(cluster):
    """
    Count the number of pixels in each cluster
    :param: KMeans cluster
    :return: numpy histogram
    """

    numLabels = np.arange(0, len(np.unique(cluster.labels_)) + 1)
    hist, _ = np.histogram(cluster.labels_, bins=numLabels)
    hist = hist.astype('float32')
    hist /= hist.sum()
    return hist


def make_bar(height, width, color):
    """
    Create an image of a given color
    :param: height of the image
    :param: width of the image
    :param: BGR pixel values of the color
    :return: tuple of bar, rgb values
    """
    bar = np.zeros((height, width, 3), np.uint8)
    bar[:] = color
    red, green, blue = int(color[2]), int(color[1]), int(color[0])
    return bar, (red, green, blue)

# Load image
img = cv2.imread('ndvi.jpg')
height, width, _ = np.shape(img)

# Reshape the image to be a list of RGB pixels
image = img.reshape((height * width, 3))

#convert from BGR to RGB for kmeans
img_kmeans = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# define a cluster
num_clusters = 1
clusters = KMeans(n_clusters=num_clusters)
clusters.fit(image)

# count the dominant colors and sort them into groups
histogram = make_histogram(clusters)

# sort the groups, most-common first
combined = zip(histogram, clusters.cluster_centers_)
combined = sorted(combined, key=lambda x: x[0], reverse=True)

# get rgb values from image to 1D array
r, g, b = cv2.split(img_kmeans)
r = r.flatten()
g = g.flatten()
b = b.flatten()

# plotting
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(r, g, b)
print("K-Means Graphical output on high fuel moisture NDVI image:")
plt.show()

# output the RGB values of the most dominant colour from the "High Fuel Moisture Image"
bars = []
hsv_values = []
for index, rows in enumerate(combined):
    bar, rgb = make_bar(100, 100, rows[1])
    dominance_text = (f'  RGB values: {rgb}')
    rgb_list = list(rgb)
    print("RGB coordinates of the most dominant colour in the high fuel moisture NDVI image: " + s
tr(rgb_list))

from IPython.display import Image
Image("ndvi.jpg")
```

```
K-Means Graphical output on high fuel moisture NDVI image:
```



```
RGB coordinates of the most dominant colour in the high fuel moisture NDVI image: [128, 170, 8]
```

Out[55]:

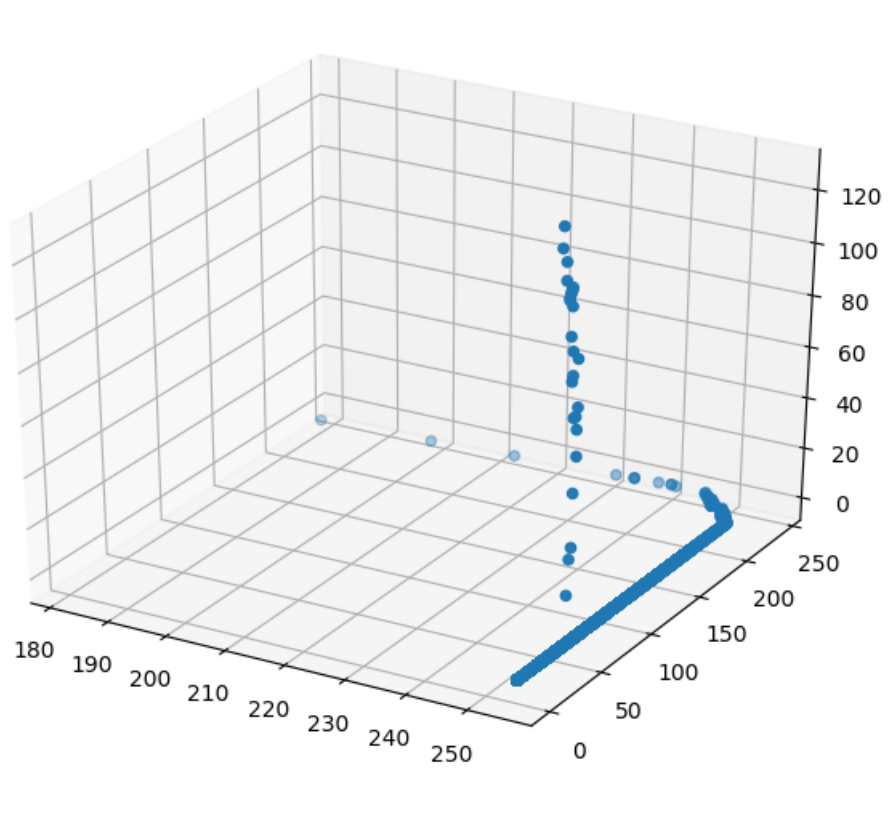


K-Means on "Low Fuel Moisture" Imagery

In [56]:

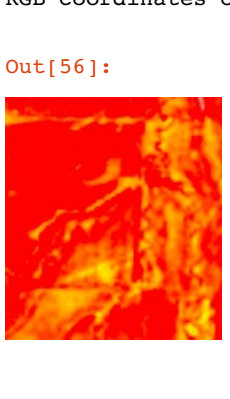
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import image as img
from sklearn.cluster import KMeans
import cv2


#Color Dominance Algorithm
def make_histogram(cluster):
    """
    Count the number of pixels in each cluster
    :param: KMeans cluster
    :return: numpy histogram
    """

    numLabels = np.arange(0, len(np.unique(cluster.labels_)) + 1)
    hist, _ = np.histogram(cluster.labels_, bins=numLabels)
    hist = hist.astype('float32')
    hist /= hist.sum()
    return hist


def make_bar(height, width, color):
    """
    Create an image of a given color
    :param: height of the image
    :param: width of the image
    :param: BGR pixel values of the color
    :return: tuple of bar, rgb values
    """
    bar = np.zeros((height, width, 3), np.uint8)
    bar[:] = color
    red, green, blue = int(color[2]), int(color[1]), int(color[0])
    return bar, (red, green, blue)

# Load image
img = cv2.imread('ndvi_fire_conditions.jpg')
height, width, _ = np.shape(img)

# Reshape the image to be a list of RGB pixels
image = img.reshape((height * width, 3))

#convert from BGR to RGB for kmeans
img_kmeans = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# define a cluster
num_clusters = 1
clusters = KMeans(n_clusters=num_clusters)
clusters.fit(image)

# count the dominant colors and sort them into groups
histogram = make_histogram(clusters)

# sort the groups, most-common first
combined = zip(histogram, clusters.cluster_centers_)
combined = sorted(combined, key=lambda x: x[0], reverse=True)

# get rgb values from image to 1D array
r, g, b = cv2.split(img_kmeans)
r = r.flatten()
g = g.flatten()
b = b.flatten()

# plotting
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(r, g, b)
print("K-Means Graphical output on high fuel moisture NDVI image:")
plt.show()

# output the RGB values of the most dominant colour
bars = []
hsv_values = []
for index, rows in enumerate(combined):
    bar, rgb = make_bar(100, 100, rows[1])
    dominance_text = (f'  RGB values: {rgb}')
    rgb_list = list(rgb)
    print("Raw RGB coordinates of the most dominant colour in the low fuel moisture NDVI image: "
+ str(rgb_list))

from IPython.display import Image
Image("ndvi_fire_conditions.jpg")
```

```
K-Means Graphical output on high fuel moisture NDVI image:
```



```
RGB coordinates of the most dominant colour in the low fuel moisture NDVI image: [254, 51, 0]
```

Out[56]:



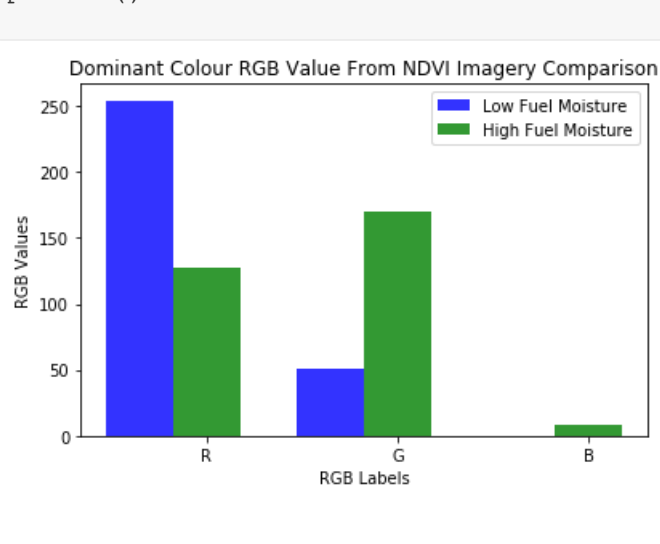
Notice how the "High Fuel Moisture" imagery contains a much broader range of RGB values than the "Low Fuel Moisture" image. This is further displayed when you look at the numerical RGB coordinates. The "High Fuel Moisture"image has more balanced R(Red) and G(Green) values, along with a small B(Blue) value. However, if you consider the "Low Fuel Moisture" image, it contains a significantly higher R(Red) value, a much smaller G(Green) value, and a non-existent B(Blue) value. This will be further explored in the next section.

2.3 Comparing RGB values from the above images

This following code graphically compares the raw RGB values of the most dominant colours from both the "High Fuel Moisture" and "Low Fuel Moisture" imagery. Doing so gives us a better understanding of the relation between K-means dominant color and fuel moisture in NDVI imagery.

In [4]:

```python
import numpy as np
import matplotlib.pyplot as plt

# data to plot
n_groups = 3
kmeans_moistfuel = (128, 170, 8) # RGB values of the most dominant colour from "High Fuel
Moisture" image
kmeans_dryfuel = (254, 51, 0)    # RGB values of the most dominant colour from "Low Fuel Moisture"
image

# create plot
fig, ax = plt.subplots()
index = np.arange(n_groups)
bar_width = 0.35
opacity = 0.8

rects1 = plt.bar(index, kmeans_dryfuel, bar_width,
alpha=opacity,
color='b',
label='Low Fuel Moisture')

rects2 = plt.bar(index + bar_width, kmeans_moistfuel, bar_width,
alpha=opacity,
color='g',
label='High Fuel Moisture')

plt.xlabel('RGB Labels')
plt.ylabel('RGB Values')
plt.title('Dominant Colour RGB Value From NDVI Imagery Comparison')
plt.xticks(index + bar_width, ('R', 'G', 'B'))
plt.legend()

plt.tight_layout()
plt.show()
```



Notice that how for NDVI imagery with Low Fuel Moisture, which is at a higher risk of fire ignition, has a very high R(Red) value, a relatively lower G(Green) value, and a non-existent B(Blue) value. In comparison, NDVI imagery with High Fuel Moisture, which is at a lower risk of fire ignition, has a very low R(Red) value, a relatively high G(Green) value, and still contains a B(Blue) value.

3. Dataset

3.1 Dataset Overview

The following code presents the final constructed dataset used to train the neural network:

Indpendant Factors: Soil Moisture, Soil Temperature, Temperature, Wind Speed, Humidity, Kmeans Red, Kmeans Green, Kmeans Blue

Dependant Factor: Fire Conditions Present (1 = yes, 0 = no)

In [111]:

```python
import pandas as pd
df = pd.read_csv('dataset.csv')
print("Total # of cases where fire conditions are present: 1016")
print("Total # of cases where fire conditions are not present: 981")
df
```

```
Total # of cases where fire conditions are present: 1016
Total # of cases where fire conditions are not present: 981
```

Out[111]:

	Soil Moisture	Soil Temperature	Temperature	Wind Speed	Humidity	kmeansR	kmeansG	kmeansB	Fire
0	0.104000	6.250000	15.400000	25.900000	33.00000	252.00000	29.00000	18.00000	1
1	0.166000	5.980000	12.700000	9.400000	100.00000	64.00000	183.00000	1.00000	0
2	0.319000	-0.440000	-5.800000	16.600000	79.00000	254.00000	117.00000	1.00000	0
3	0.098000	14.720000	16.600000	13.600000	87.00000	245.00000	74.00000	61.00000	1
4	0.179000	5.890000	11.100000	8.200000	95.00000	168.00000	171.00000	0.00000	0
...	...	...	...	...	...	...	...	...	...
1992	0.081109	12.294374	32.987770	13.884532	14.94108	241.19172	105.65478	51.22656	1
1993	0.122730	19.017860	14.087304	5.762988	106.72200	73.63818	196.36848	2.13444	0
1994	0.114193	13.446972	34.161712	8.121544	21.34440	272.14110	16.00830	2.13444	1
1995	0.126999	12.144964	34.908766	10.042540	11.73942	157.94856	229.45230	1.06722	1
1996	0.302023	0.512266	4.162158	3.521826	72.57096	163.28466	192.09960	4.26888	0

1997 rows × 9 columns

3.2 Dataset Analysis

The following code calculates the average environmental values using the above completed dataset. It presents the average values for areas classified as "1", and for areas classified as "0".

In [113]:

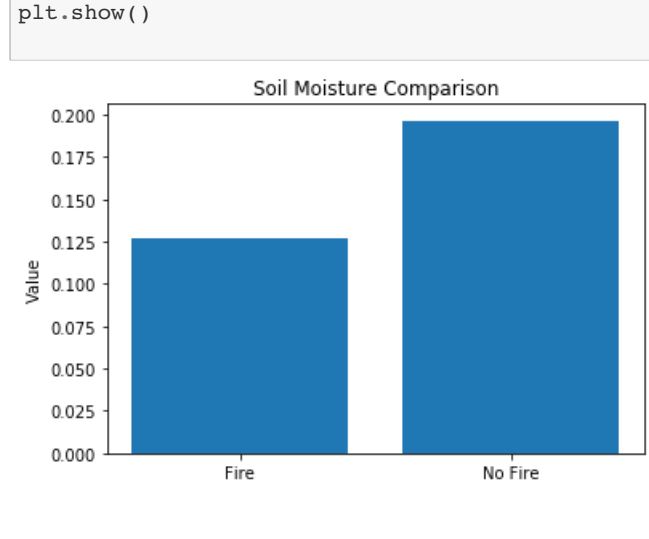
```python
import matplotlib.pyplot as plt
#importing the dataset which ONLY has fire conditions present data
df1 = pd.read_csv('dataset_1.csv')

#collecting data from the different independant variables into lists
lst_1 = list(df1.Soil_Moisture)
lst_2 = list(df1.Soil_Temp)
lst_3 = list(df1.Temp)
lst_4 = list(df1.Wind)
lst_5 = list(df1.Humidity)
lst_6 = list(df1.kmeansr)
lst_7 = list(df1.kmeansg)
lst_8 = list(df1.kmeansb )

#finding the average values of all the independant factors
average_soil_moisture = sum(lst_1) / len(lst_1)
average_soil_temp = sum(lst_2) / len(lst_2)
average_temp = sum(lst_3) / len(lst_3)
average_wind = sum(lst_4) / len(lst_4)
average_humidity = sum(lst_5) / len(lst_5)
average_kmeansr = sum(lst_6) / len(lst_6)
average_kmeansg = sum(lst_7) / len(lst_7)
average_kmeansb = sum(lst_8) / len(lst_8)

#printing this data
print("Average Environmental Conditions in Areas With Fire Conditions Present")
print("")
print("Average Soil Moisture: " + str(average_soil_moisture))
print("Average Soil Temperature: " + str(average_soil_temp))
print("Average Temperature: " + str(average_temp))
print("Average Wind Speed: " + str(average_wind))
print("Average Humidity: " + str(average_humidity))
print("Average Kmeans R Value: " + str(average_kmeansr))
print("Average Kmeans G Value: " + str(average_kmeansg))
print("Average Kmeans B Value: " + str(average_kmeansb))

print("")
print("Average Environmental Conditions in Areas Without Fire Conditions Present")
print("")
import matplotlib.pyplot as plt
#importing the dataset which ONLY has fire conditions present data
df1 = pd.read_csv('dataset_0.csv')

#collecting data from the different independant variables into lists
lst_1 = list(df1.Soil_Moisture)
lst_2 = list(df1.Soil_Temp)
lst_3 = list(df1.Temp)
lst_4 = list(df1.Wind)
lst_5 = list(df1.Humidity)
lst_6 = list(df1.kmeansr)
lst_7 = list(df1.kmeansg)
lst_8 = list(df1.kmeansb )

#finding the average values of all the independant factos
average_soil_moisture_0 = sum(lst_1) / len(lst_1)
average_soil_temp_0 = sum(lst_2) / len(lst_2)
average_temp_0 = sum(lst_3) / len(lst_3)
average_wind_0 = sum(lst_4) / len(lst_4)
average_humidity_0 = sum(lst_5) / len(lst_5)
average_kmeansr_0 = sum(lst_6) / len(lst_6)
average_kmeansg_0 = sum(lst_7) / len(lst_7)
average_kmeansb_0 = sum(lst_8) / len(lst_8)

#printing this data
print("Average Soil Moisture: " + str(average_soil_moisture_0))
print("Average Soil Temperature: " + str(average_soil_temp_0))
print("Average Temperature: " + str(average_temp_0))
print("Average Wind Speed: " + str(average_wind_0))
print("Average Humidity: " + str(average_humidity_0))
print("Average Kmeans R Value: " + str(average_kmeansr_0))
print("Average Kmeans G Value: " + str(average_kmeansg_0))
print("Average Kmeans B Value: " + str(average_kmeansb_0))
```

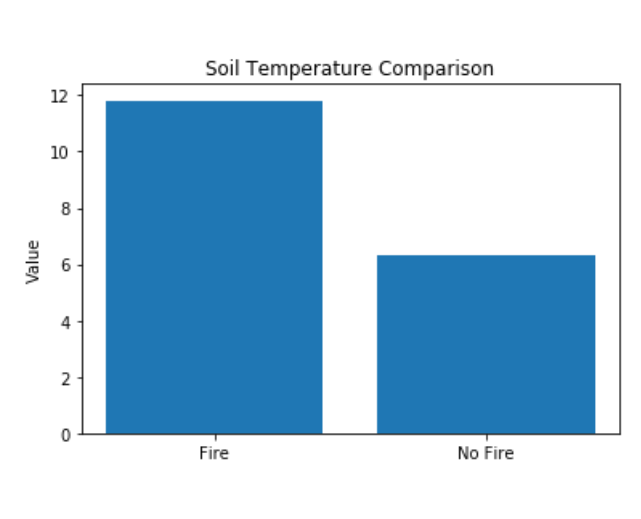
```
Average Environmental Conditions in Areas With Fire Conditions Present

Average Soil Moisture: 0.1270811019370078
Average Soil Temperature: 11.809961067204723
Average Temperature: 24.888946932637825
Average Wind Speed: 10.55650030625985
Average Humidity: 42.42064780314956
Average Kmeans R Value: 203.52379291338602
Average Kmeans G Value: 113.20263830708652
Average Kmeans B Value: 26.606805543307097

Average Environmental Conditions in Areas Without Fire Conditions Present

Average Soil Moisture: 0.19642752181039755
Average Soil Temperature: 6.321897556411823
Average Temperature: 6.0218973776146765
Average Wind Speed: 8.758573419612642
Average Humidity: 84.94897482976555
Average Kmeans R Value: 190.40092317227348
Average Kmeans G Value: 162.9695088114169
Average Kmeans B Value: 13.90074861977574
```

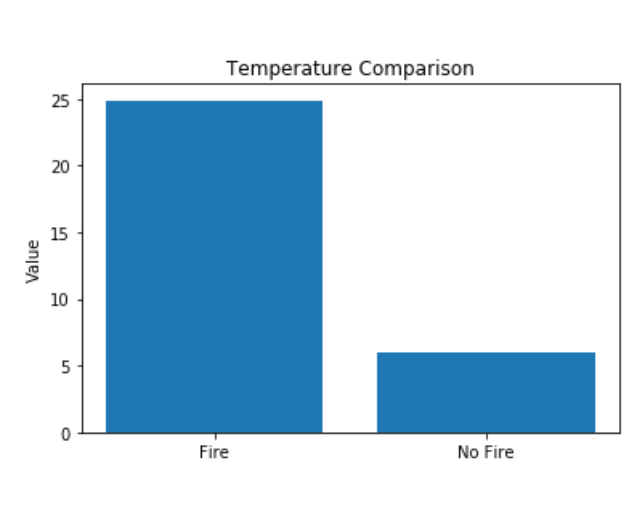
This following code takes the previously computed average values and graphically compares them against each other in order to further better understand the data.

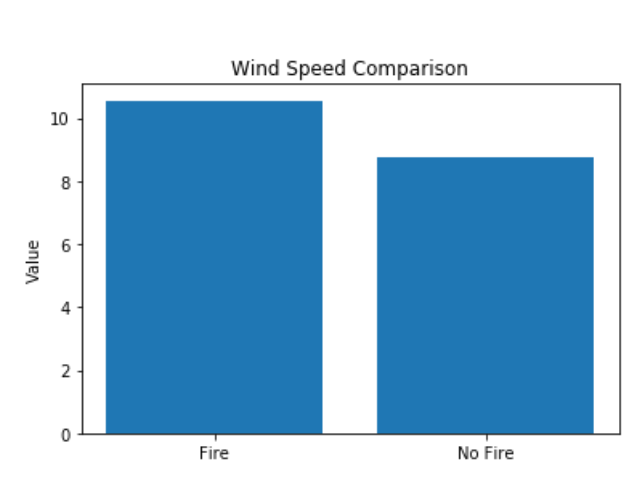
In [114]:

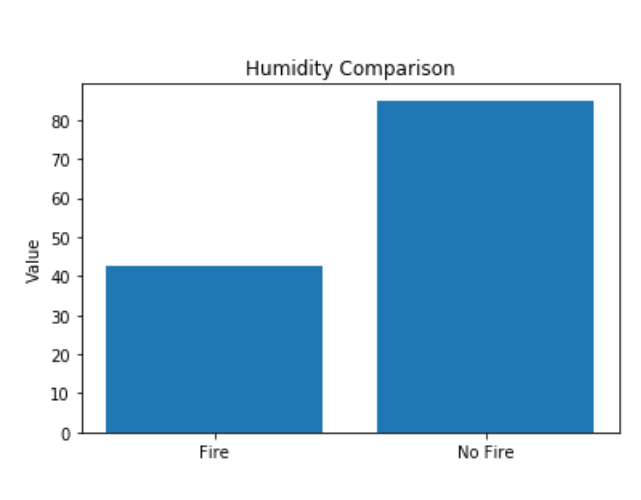
```python
labels = ['Fire', 'No Fire']

# Soil moisture plot
soil_moisture = [average_soil_moisture, average_soil_moisture_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, soil_moisture, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('Soil Moisture Comparison')
plt.show()

# Soil temperature plot
soil_temperature = [average_soil_temp, average_soil_temp_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, soil_temperature, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('Soil Temperature Comparison')
plt.show()

# Temperature plot
temperature = [average_temp, average_temp_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, temperature, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('Temperature Comparison')
plt.show()

# Wind speed plot
wind_speed = [average_wind, average_wind_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, wind_speed, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('Wind Speed Comparison')
plt.show()

# Humidity Plot
humidity = [average_humidity, average_humidity_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, humidity, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('Humidity Comparison')
plt.show()

# Kmeans Red plot
kmeansr = [average_kmeansr, average_kmeansr_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, kmeansr, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('K-Means Red Comparison')
plt.show()

# Kmeans Green plot
kmeansg = [average_kmeansg, average_kmeansg_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, kmeansg, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('K-Means Green Comparison')
plt.show()

# Kmeans Blue plot
kmeansb = [average_kmeansb, average_kmeansb_0]
y_pos = np.arange(len(labels))
plt.bar(y_pos, kmeansb, align='center')
plt.xticks(y_pos, labels)
plt.ylabel('Value')
plt.title('K-Means Blue Comparison')
plt.show()
```

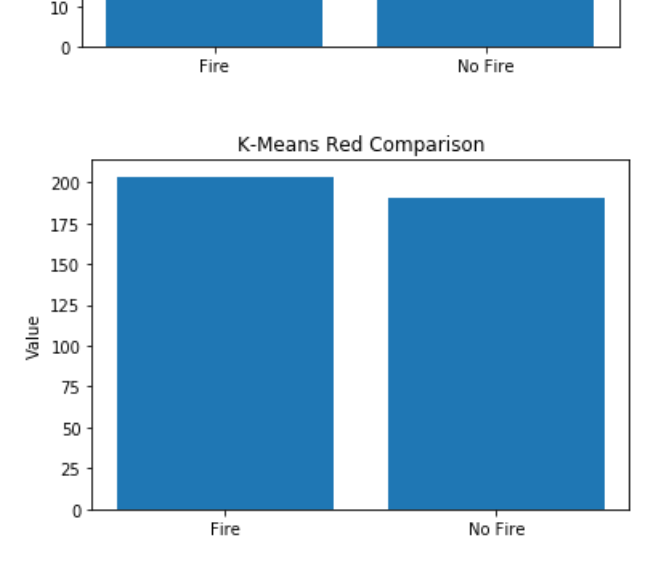


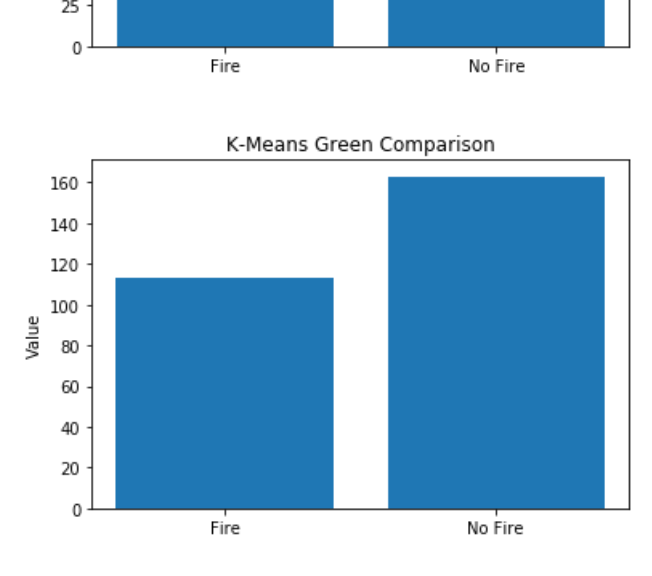


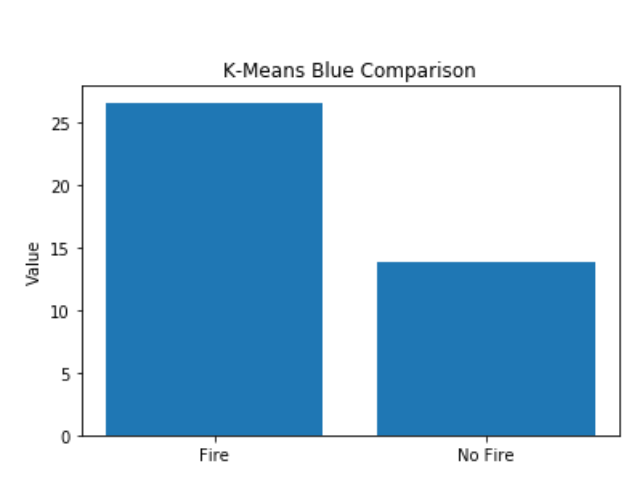












As you can see, there are some key differences in environmental conditions between the two "Fire" and "No Fire" cases. Soil moisture levels and humidity levels are higher when fires are not present. This makes sense due to the fact that higher levels of humidity and soil moisture hinder fire ignition and do not reflect ideal wildfire conditions. Conversely, soil temperature, temperature, and wind speed are substantially higher when fire conditions *are* present. This is also expected due to the fact those factors are known wildfire aggravators and significantly contribute to fire spread and ignition. Furthermore, when looking at the K-means data, the Red and Blue values are larger in areas with fire conditions while Green is higher in areas without fire conditions. This is due to the fact that Red and Blue correlate to areas with lower amounts of moisture which as mentioned previously, hinder fire ignition. The Green value is higher in areas without fire conditions present because that correlates to higher levels of fuel moisture.

Neural Network Training and Testing

1. Classifier Construction

1.1 Classifier Summary

In [116]:

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import csv
import matplotlib.pyplot as plt
import numpy
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Importing the dataset
dataset = pd.read_csv('dataset.csv')
dataset.head()

X=dataset.iloc[:,0:8]
Y=dataset.iloc[:,8]

X.head()
obj=StandardScaler()
X=obj.fit_transform(X)

X_train,X_test,y_train,y_test=train_test_split(X, Y, test_size=0.10)

classifier = Sequential()

# Adding the input layer and first hidden layer
# Adding input dim for first input layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation =
                                                    'relu', input_dim = 8))
# Adding a dropout
classifier.add(Dropout(rate =  0.20))

# Adding the second hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation
                                                               = 'relu'))

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation
                                                           = 'sigmoid'))

# Compiling the ANN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
                                                      = ['accuracy'])


print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
print("")
print(classifier.summary())
```

```
(1797, 8)
(200, 8)
(1797,)
(200,)

Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 8)                 72
_________________________________________________________________
dropout_2 (Dropout)          (None, 8)                 0
_________________________________________________________________
dense_5 (Dense)              (None, 8)                 72
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 9
=================================================================
Total params: 153
Trainable params: 153
Non-trainable params: 0
_________________________________________________________________
None
```

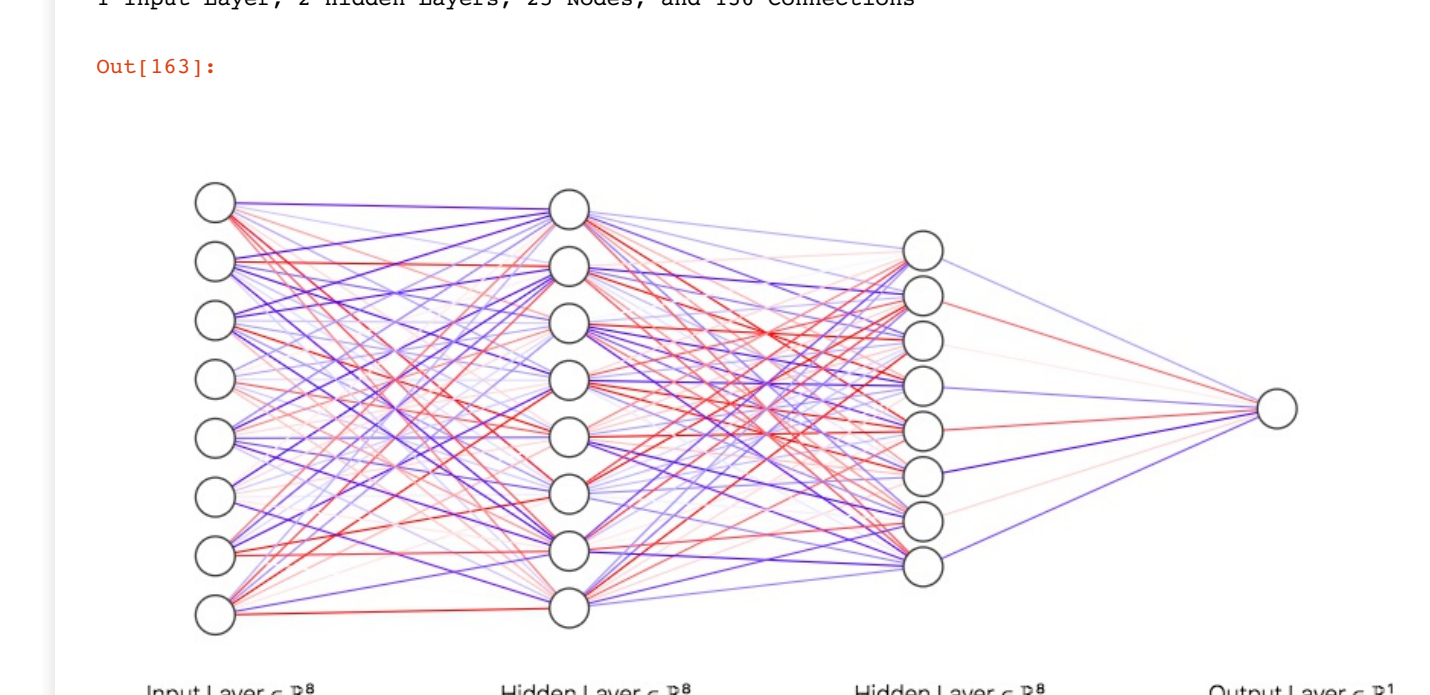
1.2 Classifier Architecture

This following code displays the architecture of the neural network constructed for this study

```
print('1 Input Layer, 2 Hidden Layers, 25 Nodes, and 136 Connections')
from IPython.display import Image
Image("nn_architecture.png")
```

```
1 Input Layer, 2 Hidden Layers, 25 Nodes, and 136 Connections
```

Out[163]:



2. Artificial Neural Network (ANN) Training and Optimization

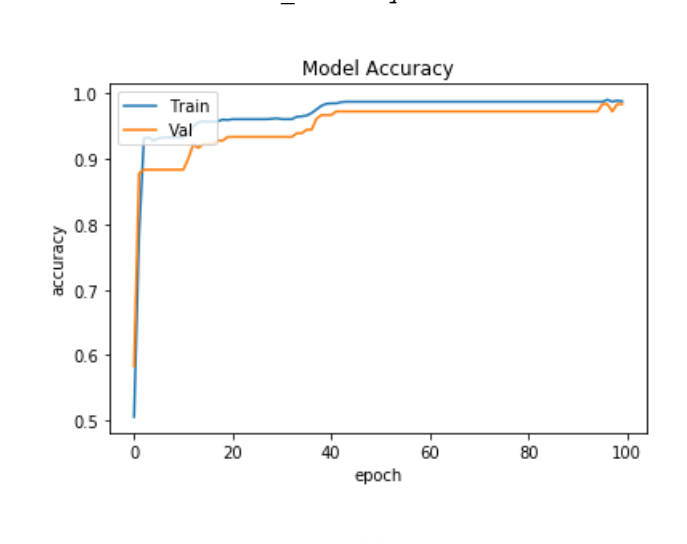
2.1 Original Neural Network Training

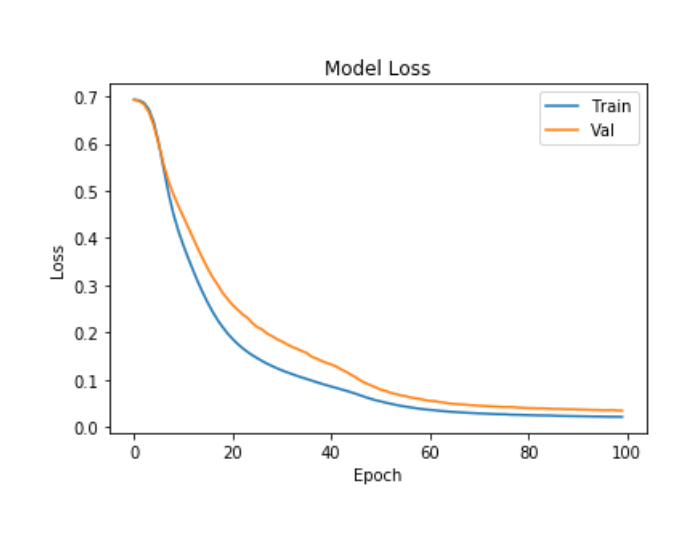
The following code is the original, un-optimized network that does not use fine-tuned parameters. It contains no dropout layer, a conventional training:validation split of 80:20, and a batch size of 100.

In [222]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import csv
import matplotlib.pyplot as plt
import numpy
from keras.callbacks import EarlyStopping, ModelCheckpoint

#THIS IS USED TO TRAIN THE MODEL

# Importing the dataset
dataset = pd.read_csv('dataset.csv')
dataset.head()

X=dataset.iloc[:,1:9]
Y=dataset.iloc[:,0]

X.head()
obj=StandardScaler()
X=obj.fit_transform(X)

X_train,X_test,y_train,y_test=train_test_split(X, Y, test_size=0.10)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

classifier = Sequential()

# Adding the input layer and first hidden layer
# Adding input dim for first input layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation =
                     'relu', input_dim = 8))

# Adding a dropout
#classifier.add(Dropout(rate =  0.10))

# Adding the second hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation
                                                   = 'relu'))

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation
                                                   = 'sigmoid'))

# Compiling the ANN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
                   = ['accuracy'])

history = classifier.fit(X_train, y_train, validation_split=0.10, batch_size = 100, epochs = 100)
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.96)

# summarize history for loss and accuracy (note that val stands for 'validate')
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Train', 'Val'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()

# saving the classifier
classifier.save("model.h5")
```

```
(1797, 8)
(200, 8)
(1797,)
(200,)
Train on 1617 samples, validate on 180 samples
Epoch 1/100
1617/1617 [==============================] - 3s 2ms/step - loss: 0.6930 - accuracy: 0.5059 - val_l
oss: 0.6925 - val_accuracy: 0.5833
Epoch 2/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.6916 - accuracy: 0.7817 - val_
loss: 0.6863 - val_accuracy: 0.8778
Epoch 3/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.6865 - accuracy: 0.9314 - val_
loss: 0.6831 - val_accuracy: 0.8833
Epoch 4/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.6726 - accuracy: 0.9326 - val_
loss: 0.6668 - val_accuracy: 0.8833
Epoch 5/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.6449 - accuracy: 0.9276 - val_
loss: 0.6378 - val_accuracy: 0.8833
Epoch 6/100
1617/1617 [==============================] - 0s 99us/step - loss: 0.6009 - accuracy: 0.9314 - val_
loss: 0.5962 - val_accuracy: 0.8833
Epoch 7/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.5478 - accuracy: 0.9326 - val_
loss: 0.5561 - val_accuracy: 0.8833
Epoch 8/100
1617/1617 [==============================] - 0s 73us/step - loss: 0.4967 - accuracy: 0.9326 - val_
loss: 0.5198 - val_accuracy: 0.8833
Epoch 9/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.4503 - accuracy: 0.9326 - val_
loss: 0.4920 - val_accuracy: 0.8833
Epoch 10/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.4168 - accuracy: 0.9326 - val_
loss: 0.4677 - val_accuracy: 0.8833
Epoch 11/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.3866 - accuracy: 0.9326 - val_
loss: 0.4448 - val_accuracy: 0.8833
Epoch 12/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.3570 - accuracy: 0.9382 - val_
loss: 0.4220 - val_accuracy: 0.9000
Epoch 13/100
1617/1617 [==============================] - 0s 70us/step - loss: 0.3311 - accuracy: 0.9499 - val_
loss: 0.3997 - val_accuracy: 0.9222
Epoch 14/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.3059 - accuracy: 0.9555 - val_
loss: 0.3764 - val_accuracy: 0.9167
Epoch 15/100
1617/1617 [==============================] - 0s 117us/step - loss: 0.2830 - accuracy: 0.9573 - val
_loss: 0.3550 - val_accuracy: 0.9222
Epoch 16/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.2619 - accuracy: 0.9567 - val_
loss: 0.3342 - val_accuracy: 0.9222
Epoch 17/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.2427 - accuracy: 0.9567 - val_
loss: 0.3158 - val_accuracy: 0.9222
Epoch 18/100
1617/1617 [==============================] - 0s 89us/step - loss: 0.2257 - accuracy: 0.9573 - val_
loss: 0.3004 - val_accuracy: 0.9278
Epoch 19/100
1617/1617 [==============================] - 0s 87us/step - loss: 0.2110 - accuracy: 0.9598 - val_
loss: 0.2826 - val_accuracy: 0.9222
Epoch 20/100
1617/1617 [==============================] - 0s 114us/step - loss: 0.1977 - accuracy: 0.9592 - val
_loss: 0.2705 - val_accuracy: 0.9333
Epoch 21/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1862 - accuracy: 0.9604 - val_
loss: 0.2582 - val_accuracy: 0.9333
Epoch 22/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1759 - accuracy: 0.9604 - val_
loss: 0.2484 - val_accuracy: 0.9333
Epoch 23/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1670 - accuracy: 0.9604 - val_
loss: 0.2386 - val_accuracy: 0.9333
Epoch 24/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.1590 - accuracy: 0.9604 - val_
loss: 0.2311 - val_accuracy: 0.9333
Epoch 25/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1518 - accuracy: 0.9604 - val_
loss: 0.2250 - val_accuracy: 0.9333
Epoch 26/100
1617/1617 [==============================] - 0s 70us/step - loss: 0.1457 - accuracy: 0.9604 - val_
loss: 0.2215 - val_accuracy: 0.9333
Epoch 27/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1394 - accuracy: 0.9604 - val_
loss: 0.2064 - val_accuracy: 0.9333
Epoch 28/100
1617/1617 [==============================] - 0s 86us/step - loss: 0.1339 - accuracy: 0.9604 - val_
loss: 0.1978 - val_accuracy: 0.9333
Epoch 29/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.1290 - accuracy: 0.9610 - val_
loss: 0.1922 - val_accuracy: 0.9333
Epoch 30/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1243 - accuracy: 0.9617 - val_
loss: 0.1858 - val_accuracy: 0.9333
Epoch 31/100
1617/1617 [==============================] - 0s 70us/step - loss: 0.1199 - accuracy: 0.9604 - val_
loss: 0.1810 - val_accuracy: 0.9333
Epoch 32/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.1159 - accuracy: 0.9604 - val_
loss: 0.1752 - val_accuracy: 0.9333
Epoch 33/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.1122 - accuracy: 0.9604 - val_
loss: 0.1700 - val_accuracy: 0.9333
Epoch 34/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.1086 - accuracy: 0.9641 - val_
loss: 0.1657 - val_accuracy: 0.9389
Epoch 35/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.1050 - accuracy: 0.9647 - val_
loss: 0.1610 - val_accuracy: 0.9389
Epoch 36/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.1017 - accuracy: 0.9660 - val_
loss: 0.1568 - val_accuracy: 0.9444
Epoch 37/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0983 - accuracy: 0.9697 - val_
loss: 0.1490 - val_accuracy: 0.9444
Epoch 38/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0950 - accuracy: 0.9753 - val_
loss: 0.1450 - val_accuracy: 0.9611
Epoch 39/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0917 - accuracy: 0.9808 - val_
loss: 0.1400 - val_accuracy: 0.9667
Epoch 40/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0887 - accuracy: 0.9839 - val_
loss: 0.1358 - val_accuracy: 0.9667
Epoch 41/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0856 - accuracy: 0.9845 - val_
loss: 0.1323 - val_accuracy: 0.9667
Epoch 42/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0828 - accuracy: 0.9845 - val_
loss: 0.1281 - val_accuracy: 0.9722
Epoch 43/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0798 - accuracy: 0.9864 - val_
loss: 0.1220 - val_accuracy: 0.9722
Epoch 44/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0768 - accuracy: 0.9870 - val_
loss: 0.1168 - val_accuracy: 0.9722
Epoch 45/100
1617/1617 [==============================] - 0s 73us/step - loss: 0.0736 - accuracy: 0.9870 - val_
loss: 0.1108 - val_accuracy: 0.9722
Epoch 46/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0702 - accuracy: 0.9870 - val_
loss: 0.1051 - val_accuracy: 0.9722
Epoch 47/100
1617/1617 [==============================] - 0s 73us/step - loss: 0.0666 - accuracy: 0.9870 - val_
loss: 0.0979 - val_accuracy: 0.9722
Epoch 48/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0630 - accuracy: 0.9870 - val_
loss: 0.0927 - val_accuracy: 0.9722
Epoch 49/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0597 - accuracy: 0.9870 - val_
loss: 0.0883 - val_accuracy: 0.9722
Epoch 50/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0566 - accuracy: 0.9870 - val_
loss: 0.0839 - val_accuracy: 0.9722
Epoch 51/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0543 - accuracy: 0.9870 - val_
loss: 0.0790 - val_accuracy: 0.9722
Epoch 52/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.0516 - accuracy: 0.9870 - val_
loss: 0.0766 - val_accuracy: 0.9722
Epoch 53/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0494 - accuracy: 0.9870 - val_
loss: 0.0725 - val_accuracy: 0.9722
Epoch 54/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0471 - accuracy: 0.9870 - val_
loss: 0.0701 - val_accuracy: 0.9722
Epoch 55/100
1617/1617 [==============================] - 0s 70us/step - loss: 0.0451 - accuracy: 0.9870 - val_
loss: 0.0671 - val_accuracy: 0.9722
Epoch 56/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.0433 - accuracy: 0.9870 - val_
loss: 0.0655 - val_accuracy: 0.9722
Epoch 57/100
1617/1617 [==============================] - 0s 87us/step - loss: 0.0417 - accuracy: 0.9870 - val_
loss: 0.0627 - val_accuracy: 0.9722
Epoch 58/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0400 - accuracy: 0.9870 - val_
loss: 0.0610 - val_accuracy: 0.9722
Epoch 59/100
1617/1617 [==============================] - 0s 89us/step - loss: 0.0386 - accuracy: 0.9870 - val_
loss: 0.0593 - val_accuracy: 0.9722
Epoch 60/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.0373 - accuracy: 0.9870 - val_
loss: 0.0570 - val_accuracy: 0.9722
Epoch 61/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0363 - accuracy: 0.9870 - val_
loss: 0.0554 - val_accuracy: 0.9722
Epoch 62/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0354 - accuracy: 0.9870 - val_
loss: 0.0548 - val_accuracy: 0.9722
Epoch 63/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0342 - accuracy: 0.9870 - val_
loss: 0.0528 - val_accuracy: 0.9722
Epoch 64/100
1617/1617 [==============================] - 0s 91us/step - loss: 0.0334 - accuracy: 0.9870 - val_
loss: 0.0516 - val_accuracy: 0.9722
Epoch 65/100
1617/1617 [==============================] - 0s 119us/step - loss: 0.0324 - accuracy: 0.9870 - val
_loss: 0.0497 - val_accuracy: 0.9722
Epoch 66/100
1617/1617 [==============================] - 0s 84us/step - loss: 0.0318 - accuracy: 0.9870 - val_
loss: 0.0487 - val_accuracy: 0.9722
Epoch 67/100
1617/1617 [==============================] - 0s 96us/step - loss: 0.0310 - accuracy: 0.9870 - val_
loss: 0.0480 - val_accuracy: 0.9722
Epoch 68/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.0304 - accuracy: 0.9870 - val_
loss: 0.0474 - val_accuracy: 0.9722
Epoch 69/100
1617/1617 [==============================] - 0s 70us/step - loss: 0.0299 - accuracy: 0.9870 - val_
loss: 0.0464 - val_accuracy: 0.9722
Epoch 70/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.0293 - accuracy: 0.9870 - val_
loss: 0.0457 - val_accuracy: 0.9722
Epoch 71/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0288 - accuracy: 0.9870 - val_
loss: 0.0452 - val_accuracy: 0.9722
Epoch 72/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0284 - accuracy: 0.9870 - val_
loss: 0.0446 - val_accuracy: 0.9722
Epoch 73/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0279 - accuracy: 0.9870 - val_
loss: 0.0439 - val_accuracy: 0.9722
Epoch 74/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0274 - accuracy: 0.9870 - val_
loss: 0.0434 - val_accuracy: 0.9722
Epoch 75/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.0270 - accuracy: 0.9870 - val_
loss: 0.0430 - val_accuracy: 0.9722
Epoch 76/100
1617/1617 [==============================] - 0s 73us/step - loss: 0.0270 - accuracy: 0.9870 - val_
loss: 0.0422 - val_accuracy: 0.9722
Epoch 77/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0263 - accuracy: 0.9870 - val_
loss: 0.0423 - val_accuracy: 0.9722
Epoch 78/100
1617/1617 [==============================] - 0s 82us/step - loss: 0.0260 - accuracy: 0.9870 - val_
loss: 0.0419 - val_accuracy: 0.9722
Epoch 79/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.0258 - accuracy: 0.9870 - val_
loss: 0.0409 - val_accuracy: 0.9722
Epoch 80/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0254 - accuracy: 0.9870 - val_
loss: 0.0406 - val_accuracy: 0.9722
Epoch 81/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.0252 - accuracy: 0.9870 - val_
loss: 0.0400 - val_accuracy: 0.9722
Epoch 82/100
1617/1617 [==============================] - 0s 93us/step - loss: 0.0249 - accuracy: 0.9870 - val_
loss: 0.0399 - val_accuracy: 0.9722
Epoch 83/100
1617/1617 [==============================] - 0s 79us/step - loss: 0.0246 - accuracy: 0.9870 - val_
loss: 0.0394 - val_accuracy: 0.9722
Epoch 84/100
1617/1617 [==============================] - 0s 75us/step - loss: 0.0245 - accuracy: 0.9870 - val_
loss: 0.0395 - val_accuracy: 0.9722
Epoch 85/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0244 - accuracy: 0.9870 - val_
loss: 0.0390 - val_accuracy: 0.9722
Epoch 86/100
1617/1617 [==============================] - 0s 81us/step - loss: 0.0240 - accuracy: 0.9870 - val_
loss: 0.0383 - val_accuracy: 0.9722
Epoch 87/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0235 - accuracy: 0.9870 - val_
loss: 0.0383 - val_accuracy: 0.9722
Epoch 88/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0234 - accuracy: 0.9870 - val_
loss: 0.0379 - val_accuracy: 0.9722
Epoch 89/100
1617/1617 [==============================] - 0s 76us/step - loss: 0.0232 - accuracy: 0.9870 - val_
loss: 0.0375 - val_accuracy: 0.9722
Epoch 90/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0231 - accuracy: 0.9870 - val_
loss: 0.0377 - val_accuracy: 0.9722
Epoch 91/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0228 - accuracy: 0.9870 - val_
loss: 0.0370 - val_accuracy: 0.9722
Epoch 92/100
1617/1617 [==============================] - 0s 71us/step - loss: 0.0227 - accuracy: 0.9870 - val_
loss: 0.0368 - val_accuracy: 0.9722
Epoch 93/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0225 - accuracy: 0.9870 - val_
loss: 0.0365 - val_accuracy: 0.9722
Epoch 94/100
1617/1617 [==============================] - 0s 73us/step - loss: 0.0222 - accuracy: 0.9870 - val_
loss: 0.0361 - val_accuracy: 0.9722
Epoch 95/100
1617/1617 [==============================] - 0s 85us/step - loss: 0.0222 - accuracy: 0.9870 - val_
loss: 0.0361 - val_accuracy: 0.9722
Epoch 96/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0220 - accuracy: 0.9870 - val_
loss: 0.0354 - val_accuracy: 0.9833
Epoch 97/100
1617/1617 [==============================] - 0s 78us/step - loss: 0.0218 - accuracy: 0.9901 - val_
loss: 0.0354 - val_accuracy: 0.9833
Epoch 98/100
1617/1617 [==============================] - 0s 72us/step - loss: 0.0217 - accuracy: 0.9870 - val_
loss: 0.0360 - val_accuracy: 0.9722
Epoch 99/100
1617/1617 [==============================] - 0s 77us/step - loss: 0.0216 - accuracy: 0.9889 - val_
loss: 0.0349 - val_accuracy: 0.9833
Epoch 100/100
1617/1617 [==============================] - 0s 74us/step - loss: 0.0215 - accuracy: 0.9876 - val_
loss: 0.0347 - val_accuracy: 0.9833
```





As you can see, this model shows large signs of overfitting. Overfitting is when the neural network has simply memorized the training samples and has not learned to generalize to new situations. This is evident when the validation loss plot goes higher than the training loss plot. This model is also achieving a validation accuracy of 98.33% and this is something we can also improve. In the next section, we will see how our new model, this time fine-tuned with custom hyperparameters will perform.

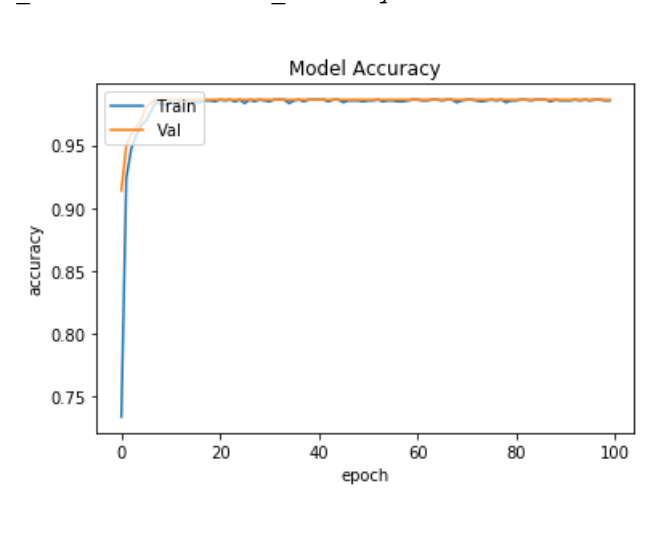
2.2 Training Wildfire Conditions Model with Fine-Tuned Hyperparameters

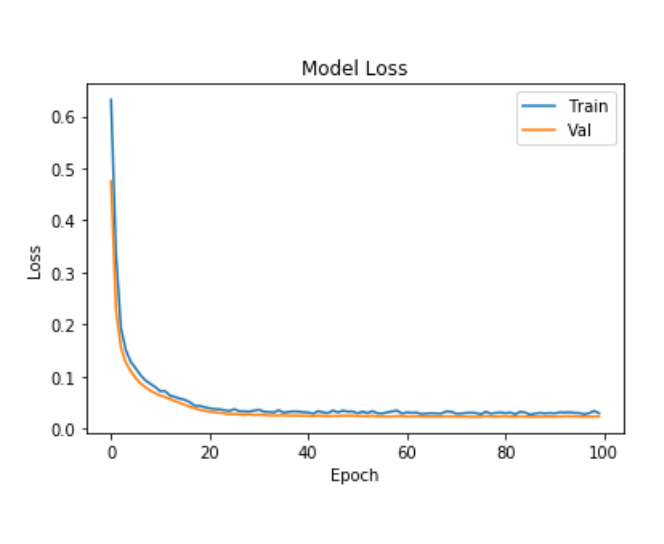
This network takes advantage of our fine-tuned optimized hyperparameters and is a major improvement over the previous network. It employs a dropout with rate 0.20 after the first hidden layer to account for overfitting, a training:validation:test split of 70:20:10, and a batch size of 5.

In [195]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import csv
import matplotlib.pyplot as plt
import numpy
from keras.callbacks import EarlyStopping, ModelCheckpoint

#THIS IS USED TO TRAIN THE MODEL

# Importing the dataset
dataset = pd.read_csv('dataset.csv')
dataset.head()

X=dataset.iloc[:,1:9]
Y=dataset.iloc[:,0]

X.head()
obj=StandardScaler()
X=obj.fit_transform(X)

X_train,X_test,y_train,y_test=train_test_split(X, Y, test_size=0.10)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

classifier = Sequential()

# Adding the input layer and first hidden layer
# Adding input dim for first input layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation =
                     'relu', input_dim = 8))

# Adding a dropout
classifier.add(Dropout(rate =  0.20))

# Adding the second hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation
                                                   = 'relu'))

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation
                                                   = 'sigmoid'))

# Compiling the ANN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
                   = ['accuracy'])

history = classifier.fit(X_train, y_train, validation_split=0.20, batch_size = 5, epochs = 100)
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.96)

# summarize history for loss and accuracy (note that val stands for 'validate')
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Train', 'Val'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()

# saving the classifier
classifier.save("model.h5")
```

```
(1797, 8)
(200, 8)
(1797,)
(200,)
Train on 1437 samples, validate on 360 samples
Epoch 1/100
1437/1437 [==============================] - 2s 2ms/step - loss: 0.6317 - accuracy: 0.7342 - val_l
oss: 0.4752 - val_accuracy: 0.9139
Epoch 2/100
1437/1437 [==============================] - 1s 690us/step - loss: 0.3356 - accuracy: 0.9235 - val
_loss: 0.2281 - val_accuracy: 0.9500
Epoch 3/100
1437/1437 [==============================] - 1s 720us/step - loss: 0.1932 - accuracy: 0.9471 - val
_loss: 0.1554 - val_accuracy: 0.9583
Epoch 4/100
1437/1437 [==============================] - 1s 722us/step - loss: 0.1509 - accuracy: 0.9576 - val
_loss: 0.1272 - val_accuracy: 0.9639
Epoch 5/100
1437/1437 [==============================] - 1s 706us/step - loss: 0.1284 - accuracy: 0.9659 - val
_loss: 0.1102 - val_accuracy: 0.9694
Epoch 6/100
1437/1437 [==============================] - 1s 700us/step - loss: 0.1158 - accuracy: 0.9694 - val
_loss: 0.0972 - val_accuracy: 0.9806
Epoch 7/100
1437/1437 [==============================] - 1s 702us/step - loss: 0.1027 - accuracy: 0.9770 - val
_loss: 0.0869 - val_accuracy: 0.9833
Epoch 8/100
1437/1437 [==============================] - 1s 708us/step - loss: 0.0925 - accuracy: 0.9840 - val
_loss: 0.0796 - val_accuracy: 0.9861
Epoch 9/100
1437/1437 [==============================] - 1s 689us/step - loss: 0.0862 - accuracy: 0.9826 - val
_loss: 0.0730 - val_accuracy: 0.9861
Epoch 10/100
1437/1437 [==============================] - 1s 706us/step - loss: 0.0797 - accuracy: 0.9819 - val
_loss: 0.0680 - val_accuracy: 0.9861
Epoch 11/100
1437/1437 [==============================] - 1s 705us/step - loss: 0.0721 - accuracy: 0.9847 - val
_loss: 0.0637 - val_accuracy: 0.9861
Epoch 12/100
1437/1437 [==============================] - 1s 680us/step - loss: 0.0720 - accuracy: 0.9812 - val
_loss: 0.0606 - val_accuracy: 0.9861
Epoch 13/100
1437/1437 [==============================] - 1s 850us/step - loss: 0.0635 - accuracy: 0.9854 - val
_loss: 0.0570 - val_accuracy: 0.9861
Epoch 14/100
1437/1437 [==============================] - 1s 738us/step - loss: 0.0610 - accuracy: 0.9847 - val
_loss: 0.0531 - val_accuracy: 0.9861
Epoch 15/100
1437/1437 [==============================] - 1s 713us/step - loss: 0.0576 - accuracy: 0.9854 - val
_loss: 0.0492 - val_accuracy: 0.9861
Epoch 16/100
1437/1437 [==============================] - 1s 758us/step - loss: 0.0553 - accuracy: 0.9840 - val
_loss: 0.0458 - val_accuracy: 0.9861
Epoch 17/100
1437/1437 [==============================] - 1s 739us/step - loss: 0.0509 - accuracy: 0.9840 - val
_loss: 0.0423 - val_accuracy: 0.9861
Epoch 18/100
1437/1437 [==============================] - 1s 725us/step - loss: 0.0440 - accuracy: 0.9854 - val
_loss: 0.0389 - val_accuracy: 0.9861
Epoch 19/100
1437/1437 [==============================] - 1s 722us/step - loss: 0.0438 - accuracy: 0.9854 - val
_loss: 0.0362 - val_accuracy: 0.9861
Epoch 20/100
1437/1437 [==============================] - 1s 697us/step - loss: 0.0413 - accuracy: 0.9847 - val
_loss: 0.0343 - val_accuracy: 0.9861
Epoch 21/100
1437/1437 [==============================] - 1s 679us/step - loss: 0.0393 - accuracy: 0.9861 - val
_loss: 0.0328 - val_accuracy: 0.9861
Epoch 22/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0378 - accuracy: 0.9854 - val
_loss: 0.0315 - val_accuracy: 0.9861
Epoch 23/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0374 - accuracy: 0.9861 - val
_loss: 0.0307 - val_accuracy: 0.9861
Epoch 24/100
1437/1437 [==============================] - 1s 676us/step - loss: 0.0356 - accuracy: 0.9847 - val
_loss: 0.0294 - val_accuracy: 0.9861
Epoch 25/100
1437/1437 [==============================] - 1s 682us/step - loss: 0.0342 - accuracy: 0.9861 - val
_loss: 0.0280 - val_accuracy: 0.9861
Epoch 26/100
1437/1437 [==============================] - 1s 678us/step - loss: 0.0375 - accuracy: 0.9833 - val
_loss: 0.0282 - val_accuracy: 0.9861
Epoch 27/100
1437/1437 [==============================] - 1s 680us/step - loss: 0.0337 - accuracy: 0.9861 - val
_loss: 0.0274 - val_accuracy: 0.9861
Epoch 28/100
1437/1437 [==============================] - 1s 713us/step - loss: 0.0333 - accuracy: 0.9847 - val
_loss: 0.0268 - val_accuracy: 0.9861
Epoch 29/100
1437/1437 [==============================] - 1s 709us/step - loss: 0.0330 - accuracy: 0.9861 - val
_loss: 0.0278 - val_accuracy: 0.9861
Epoch 30/100
1437/1437 [==============================] - 1s 725us/step - loss: 0.0344 - accuracy: 0.9854 - val
_loss: 0.0265 - val_accuracy: 0.9861
Epoch 31/100
1437/1437 [==============================] - 1s 757us/step - loss: 0.0366 - accuracy: 0.9847 - val
_loss: 0.0269 - val_accuracy: 0.9861
Epoch 32/100
1437/1437 [==============================] - 1s 741us/step - loss: 0.0326 - accuracy: 0.9861 - val
_loss: 0.0263 - val_accuracy: 0.9861
Epoch 33/100
1437/1437 [==============================] - 1s 722us/step - loss: 0.0322 - accuracy: 0.9861 - val
_loss: 0.0255 - val_accuracy: 0.9861
Epoch 34/100
1437/1437 [==============================] - 1s 734us/step - loss: 0.0313 - accuracy: 0.9861 - val
_loss: 0.0255 - val_accuracy: 0.9861
Epoch 35/100
1437/1437 [==============================] - 1s 741us/step - loss: 0.0355 - accuracy: 0.9833 - val
_loss: 0.0255 - val_accuracy: 0.9861
Epoch 36/100
1437/1437 [==============================] - 1s 680us/step - loss: 0.0308 - accuracy: 0.9854 - val
_loss: 0.0258 - val_accuracy: 0.9861
Epoch 37/100
1437/1437 [==============================] - 1s 717us/step - loss: 0.0324 - accuracy: 0.9861 - val
_loss: 0.0251 - val_accuracy: 0.9861
Epoch 38/100
1437/1437 [==============================] - 1s 796us/step - loss: 0.0331 - accuracy: 0.9847 - val
_loss: 0.0253 - val_accuracy: 0.9861
Epoch 39/100
1437/1437 [==============================] - 1s 759us/step - loss: 0.0329 - accuracy: 0.9861 - val
_loss: 0.0251 - val_accuracy: 0.9861
Epoch 40/100
1437/1437 [==============================] - 1s 706us/step - loss: 0.0317 - accuracy: 0.9861 - val
_loss: 0.0249 - val_accuracy: 0.9861
Epoch 41/100
1437/1437 [==============================] - 1s 813us/step - loss: 0.0311 - accuracy: 0.9861 - val
_loss: 0.0246 - val_accuracy: 0.9861
Epoch 42/100
1437/1437 [==============================] - 1s 814us/step - loss: 0.0290 - accuracy: 0.9861 - val
_loss: 0.0240 - val_accuracy: 0.9861
Epoch 43/100
1437/1437 [==============================] - 1s 780us/step - loss: 0.0338 - accuracy: 0.9847 - val
_loss: 0.0252 - val_accuracy: 0.9861
Epoch 44/100
1437/1437 [==============================] - 1s 902us/step - loss: 0.0313 - accuracy: 0.9861 - val
_loss: 0.0245 - val_accuracy: 0.9861
Epoch 45/100
1437/1437 [==============================] - 1s 755us/step - loss: 0.0299 - accuracy: 0.9861 - val
_loss: 0.0241 - val_accuracy: 0.9861
Epoch 46/100
1437/1437 [==============================] - 1s 820us/step - loss: 0.0350 - accuracy: 0.9840 - val
_loss: 0.0245 - val_accuracy: 0.9861
Epoch 47/100
1437/1437 [==============================] - 1s 803us/step - loss: 0.0314 - accuracy: 0.9854 - val
_loss: 0.0243 - val_accuracy: 0.9861
Epoch 48/100
1437/1437 [==============================] - 1s 709us/step - loss: 0.0347 - accuracy: 0.9854 - val
_loss: 0.0250 - val_accuracy: 0.9861
Epoch 49/100
1437/1437 [==============================] - 1s 793us/step - loss: 0.0329 - accuracy: 0.9854 - val
_loss: 0.0249 - val_accuracy: 0.9861
Epoch 50/100
1437/1437 [==============================] - 1s 758us/step - loss: 0.0332 - accuracy: 0.9847 - val
_loss: 0.0251 - val_accuracy: 0.9861
Epoch 51/100
1437/1437 [==============================] - 1s 689us/step - loss: 0.0295 - accuracy: 0.9854 - val
_loss: 0.0243 - val_accuracy: 0.9861
Epoch 52/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0328 - accuracy: 0.9854 - val
_loss: 0.0244 - val_accuracy: 0.9861
Epoch 53/100
1437/1437 [==============================] - 1s 695us/step - loss: 0.0299 - accuracy: 0.9861 - val
_loss: 0.0241 - val_accuracy: 0.9861
Epoch 54/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0340 - accuracy: 0.9847 - val
_loss: 0.0249 - val_accuracy: 0.9861
Epoch 55/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0297 - accuracy: 0.9854 - val
_loss: 0.0237 - val_accuracy: 0.9861
Epoch 56/100
1437/1437 [==============================] - 1s 670us/step - loss: 0.0293 - accuracy: 0.9854 - val
_loss: 0.0237 - val_accuracy: 0.9861
Epoch 57/100
1437/1437 [==============================] - 1s 667us/step - loss: 0.0313 - accuracy: 0.9847 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 58/100
1437/1437 [==============================] - 1s 664us/step - loss: 0.0332 - accuracy: 0.9847 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 59/100
1437/1437 [==============================] - 1s 666us/step - loss: 0.0350 - accuracy: 0.9854 - val
_loss: 0.0244 - val_accuracy: 0.9861
Epoch 60/100
1437/1437 [==============================] - 1s 662us/step - loss: 0.0292 - accuracy: 0.9861 - val
_loss: 0.0241 - val_accuracy: 0.9861
Epoch 61/100
1437/1437 [==============================] - 1s 777us/step - loss: 0.0313 - accuracy: 0.9861 - val
_loss: 0.0238 - val_accuracy: 0.9861
Epoch 62/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0307 - accuracy: 0.9854 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 63/100
1437/1437 [==============================] - 1s 681us/step - loss: 0.0310 - accuracy: 0.9854 - val
_loss: 0.0240 - val_accuracy: 0.9861
Epoch 64/100
1437/1437 [==============================] - 1s 670us/step - loss: 0.0283 - accuracy: 0.9861 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 65/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0292 - accuracy: 0.9861 - val
_loss: 0.0234 - val_accuracy: 0.9861
Epoch 66/100
1437/1437 [==============================] - 1s 754us/step - loss: 0.0298 - accuracy: 0.9854 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 67/100
1437/1437 [==============================] - 1s 721us/step - loss: 0.0293 - accuracy: 0.9861 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 68/100
1437/1437 [==============================] - 1s 698us/step - loss: 0.0290 - accuracy: 0.9861 - val
_loss: 0.0233 - val_accuracy: 0.9861
Epoch 69/100
1437/1437 [==============================] - 1s 722us/step - loss: 0.0333 - accuracy: 0.9840 - val
_loss: 0.0239 - val_accuracy: 0.9861
Epoch 70/100
1437/1437 [==============================] - 1s 703us/step - loss: 0.0325 - accuracy: 0.9854 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 71/100
1437/1437 [==============================] - 1s 702us/step - loss: 0.0290 - accuracy: 0.9861 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 72/100
1437/1437 [==============================] - 1s 685us/step - loss: 0.0291 - accuracy: 0.9861 - val
_loss: 0.0233 - val_accuracy: 0.9861
Epoch 73/100
1437/1437 [==============================] - 1s 683us/step - loss: 0.0304 - accuracy: 0.9854 - val
_loss: 0.0233 - val_accuracy: 0.9861
Epoch 74/100
1437/1437 [==============================] - 1s 735us/step - loss: 0.0307 - accuracy: 0.9847 - val
_loss: 0.0228 - val_accuracy: 0.9861
Epoch 75/100
1437/1437 [==============================] - 1s 713us/step - loss: 0.0302 - accuracy: 0.9854 - val
_loss: 0.0230 - val_accuracy: 0.9861
Epoch 76/100
1437/1437 [==============================] - 1s 710us/step - loss: 0.0276 - accuracy: 0.9861 - val
_loss: 0.0228 - val_accuracy: 0.9861
Epoch 77/100
1437/1437 [==============================] - 1s 697us/step - loss: 0.0327 - accuracy: 0.9854 - val
_loss: 0.0244 - val_accuracy: 0.9861
Epoch 78/100
1437/1437 [==============================] - 1s 699us/step - loss: 0.0289 - accuracy: 0.9861 - val
_loss: 0.0234 - val_accuracy: 0.9861
Epoch 79/100
1437/1437 [==============================] - 1s 711us/step - loss: 0.0304 - accuracy: 0.9840 - val
_loss: 0.0233 - val_accuracy: 0.9861
Epoch 80/100
1437/1437 [==============================] - 1s 697us/step - loss: 0.0311 - accuracy: 0.9854 - val
_loss: 0.0238 - val_accuracy: 0.9861
Epoch 81/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0300 - accuracy: 0.9854 - val
_loss: 0.0238 - val_accuracy: 0.9861
Epoch 82/100
1437/1437 [==============================] - 1s 692us/step - loss: 0.0308 - accuracy: 0.9861 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 83/100
1437/1437 [==============================] - 1s 699us/step - loss: 0.0279 - accuracy: 0.9861 - val
_loss: 0.0230 - val_accuracy: 0.9861
Epoch 84/100
1437/1437 [==============================] - 1s 689us/step - loss: 0.0325 - accuracy: 0.9854 - val
_loss: 0.0238 - val_accuracy: 0.9861
Epoch 85/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0311 - accuracy: 0.9861 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 86/100
1437/1437 [==============================] - 1s 685us/step - loss: 0.0270 - accuracy: 0.9861 - val
_loss: 0.0230 - val_accuracy: 0.9861
Epoch 87/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0290 - accuracy: 0.9861 - val
_loss: 0.0231 - val_accuracy: 0.9861
Epoch 88/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0304 - accuracy: 0.9847 - val
_loss: 0.0231 - val_accuracy: 0.9861
Epoch 89/100
1437/1437 [==============================] - 1s 686us/step - loss: 0.0294 - accuracy: 0.9861 - val
_loss: 0.0233 - val_accuracy: 0.9861
Epoch 90/100
1437/1437 [==============================] - 1s 688us/step - loss: 0.0301 - accuracy: 0.9854 - val
_loss: 0.0232 - val_accuracy: 0.9861
Epoch 91/100
1437/1437 [==============================] - 1s 699us/step - loss: 0.0295 - accuracy: 0.9854 - val
_loss: 0.0231 - val_accuracy: 0.9861
Epoch 92/100
1437/1437 [==============================] - 1s 686us/step - loss: 0.0313 - accuracy: 0.9854 - val
_loss: 0.0236 - val_accuracy: 0.9861
Epoch 93/100
1437/1437 [==============================] - 1s 700us/step - loss: 0.0310 - accuracy: 0.9861 - val
_loss: 0.0239 - val_accuracy: 0.9861
Epoch 94/100
1437/1437 [==============================] - 1s 686us/step - loss: 0.0315 - accuracy: 0.9854 - val
_loss: 0.0239 - val_accuracy: 0.9861
Epoch 95/100
1437/1437 [==============================] - 1s 689us/step - loss: 0.0306 - accuracy: 0.9861 - val
_loss: 0.0237 - val_accuracy: 0.9861
Epoch 96/100
1437/1437 [==============================] - 1s 690us/step - loss: 0.0301 - accuracy: 0.9854 - val
_loss: 0.0235 - val_accuracy: 0.9861
Epoch 97/100
1437/1437 [==============================] - 1s 687us/step - loss: 0.0279 - accuracy: 0.9861 - val
_loss: 0.0232 - val_accuracy: 0.9861
Epoch 98/100
1437/1437 [==============================] - 1s 690us/step - loss: 0.0299 - accuracy: 0.9861 - val
_loss: 0.0230 - val_accuracy: 0.9861
Epoch 99/100
1437/1437 [==============================] - 1s 684us/step - loss: 0.0343 - accuracy: 0.9854 - val
_loss: 0.0237 - val_accuracy: 0.9861
Epoch 100/100
1437/1437 [==============================] - 1s 704us/step - loss: 0.0297 - accuracy: 0.9854 - val
_loss: 0.0236 - val_accuracy: 0.9861
```





Training Accuracy: 98.68%

Validation Accuracy: 98.61%

As you can see, this fine-tuned model achieved a validation accuracy of 98.61% as opposed to 98.33% and shows virtually no signs of overfitting when the training and validation plots converge in the loss graph. The training was also much more accurate, more stable, and less volatile. This is the final completed model which will be used for testing.

3. Completed Model Testing

This code below will test our final model on the test dataset which contains 200 unique cases. All those different cases will be fed into the network which will output it's predictions as either a 1 (fire conditions present), or 0 (fire conditions not present). The final accuracy will be derived by dividing the number of correct predictions over 200 (the number of test cases). Note how it classifies a case as "1" only if it is atleast 95% accurate. This makes sure that the algorithm is robust and held to high standards.

In [163]:

```
y_pred = classifier.predict(X_test)
# It's classified as "1" if the model is 95% sure that fire conditions are present. Else, it's "0"
y_pred=[1 if y>=0.95 else 0 for y in y_pred]
accuracy=0
for value in range(200):# The test dataset includes 200 cases
    if(y_pred[value]==list(y_test)[value]):
        accuracy+=1
print(y_pred) # Prints all the classifications made by the neural network
print('Final Test Accuracy: ' + str(accuracy/200*100) + "%") # Getting accuracy in percentage for
```

```
[1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0,
1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0,
0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1,
0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1,
1, 1]
Final Test Accuracy: 98.0%
```

Our final completed model was able to correctly determine and classify 196/200 fire conditions cases and achieved a test accuracy of 98%

Conclusion

With this research, artificial neural networks (ANNs) were successfully constructed and trained to classify regions on whether or not wildfire conditions were present based on environmental conditions inputs. Relying on current methods, firefighters must rely on inaccurate and highly subjective human inference and experience. However, by using a myriad of environmental conditions based on the Canadian Forest Fire Weather Index System, and by using new innovative techniques such as k-means clustering for fuel moisture quantification on NDVI imagery, a dataset was created which was used to train an artificial neural network. By taking advantage of highly optimized hyperparameters and neural network architectures, this model achieved an accuracy of 98% during testing. This has successfully demonstrated the effectiveness of our technique and will provide major needed leaps to the fields of fire dynamics, sustainable forestry management, and wildfire danger mitigation.

In [ ]: