```
# Imports ------------------------------------------------------------
--------

# Data
import pandas as pd
import pickle
import numpy as np
import os

# Geocoding
from geopy.geocoders import Nominatim
from geopy.exc import GeocoderTimedOut, GeocoderQuotaExceeded
import time
# import geopy
# import re
# from geopy.extra.rate_limiter import RateLimiter
# import matplotlib.pyplot as plt
# import folium
# from folium.plugins import FastMarkerCluster

# Machine learning
from subprocess import Popen, PIPE, DEVNULL
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf

# Global Variables ---------------------------------------------------
--------
#d = "drive/MyDrive/Classes/CSCE_5214_Software_Dev_for_AI/project/"
d = ""
data_d = "%sdata/"%d

class HurricaneLosses():

    def __init__(self, d=""):

        # Class variables
        self.d = d
        self.data_dir = "%sdata/"%self.d
        self.model_dir = "%smodel_files/"%self.d
        self.r_dir = "C:/Program Files/R/R-4.0.3/bin/Rscript"
        self.locator = None
        self.encoder = None
        self.model = None
        self.policies = None

        # Class function calls
        self.load_models()

    def reset_R_directory(self, d):
        self.r_dir = d

    def save_class(self, directory, file_name="HurricaneLoss"):
        with open("%s%s.pickle"%(directory, file_name), "wb") as f:
            pickle.dump(self, f)

    def load_models(self):
        print("Loading models...")
```

```
        # Load geocoder
        self.locator = Nominatim(user_agent="myGeocoder")
        print("Locator loaded")

        # Load one-hot-encoder
        self.encoder = None
        with open("%sencoder-one_hot.pickle"%self.model_dir, "rb") as
f:
            self.encoder = pickle.load(f)
        print("Encoder loaded")

        # Load model - 3_nn
        self.model = tf.keras.Sequential([
            tf.keras.layers.Input((15,)),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(32, activation='relu'),
            tf.keras.layers.Dense(1)
        ])
        self.model.compile(optimizer='adam',
                    loss=tf.keras.losses.MeanSquaredError(),
                    metrics=['mean_absolute_error'])
        self.model.load_weights("%smodel-3_nn.hdf5"%self.model_dir)
        print("Model loaded")
        print("All models loaded")

    def encode(self, x, wind, y, enc):
        x_hot = enc.transform(x).toarray()
        wind = wind.to_numpy()

        x = []
        for i, r in enumerate(x_hot):
            row = []
            for j in r:
                row.append(j)
            row.append(wind[i][0]*1.0)
            x.append(row)

        y_n = []
        for r in np.ndarray.tolist(y.to_numpy()):
            y_n.append(r[0])

        return x, y_n

    def train_model(self):

        # Get data
        train = None
        val = None
        test = None
        with open("%strain.csv"%self.data_dir) as file:
            train = pd.read_csv(file)
        with open("%sval.csv"%self.data_dir) as file:
            val = pd.read_csv(file)
        with open("%stest.csv"%self.data_dir) as file:
            test = pd.read_csv(file)
```

```
        # Transform data
        train_x = train[["Occupancy", "Construction", "YearBuilt",
"Floor", "SquareFootage"]]
        train_x_wind = train[["Windspeed"]]
        train_y = train[["Loss"]]
        val_x = val[["Occupancy", "Construction", "YearBuilt", "Floor",
"SquareFootage"]]
        val_x_wind = val[["Windspeed"]]
        val_y = val[["Loss"]]
        test_x = test[["Occupancy", "Construction", "YearBuilt",
"Floor", "SquareFootage"]]
        test_x_wind = test[["Windspeed"]]
        test_y = test[["Loss"]]

        # One-hot encoder
        ohe = OneHotEncoder()
        ohe.fit(train_x)
        with open("%sencoder-one_hot.pickle"%self.model_dir, "wb") as
f:
            pickle.dump(ohe, f)

        # Encode data
        train_x, train_y = self.encode(train_x, train_x_wind, train_y,
ohe)
        val_x, val_y = self.encode(val_x, val_x_wind, val_y, ohe)
        test_x, test_y = self.encode(test_x, test_x_wind, test_y, ohe)

        # Build model
        model = tf.keras.Sequential([
            tf.keras.layers.Input((15,)),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(32, activation='relu'),
            tf.keras.layers.Dense(1)
        ])
        model.compile(optimizer='adam',
                      loss=tf.keras.losses.MeanSquaredError(),
                      metrics=['mean_absolute_error'])

        # Set up checkpoint
        ckpt_dir = "%sckpts_3_nn/"%self.model_dir
        if not os.path.exists(ckpt_dir):
            os.makedirs(ckpt_dir)
        checkpoint_path = "%scp-{epoch:04d}.hdf5"%ckpt_dir
#        checkpoint_dir = os.path.dirname(checkpoint_path)
        cp_callback =
tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                         verbose=1,

save_weights_only=True,

save_freq="epoch"
                                                         )

        # Train model
```

```
        history = model.fit(train_x, train_y, epochs=100,
validation_data=(val_x, val_y), callbacks=[cp_callback])
        best = [-1, 1]
        for k in history.history:
            for i, r in enumerate(history.history[k]):
                if k == "val_mean_absolute_error" and r < best[1]:
                    best[0] = i+1
                    best[1] = r

        # Load new models
        self.load_models()

    # Geocoding functions
    def gcode (self, address, postalcode, lat):
        if (lat=='Yes'):
            if self.locator.geocode(address) is None:
                return self.locator.geocode(postalcode).point[0]
            else:
                return self.locator.geocode(address).point[0]
        else:
            if self.locator.geocode(address) is None:
                return self.locator.geocode(postalcode).point[1]
            else:
                return self.locator.geocode(address).point[1]


    def do_geocode(self, address, postalcode):
        try:
            if self.locator.geocode(address) is None:
                return ('Zip',
self.locator.geocode(postalcode).latitude,
self.locator.geocode(postalcode).longitude)
            else:
                return ('Street',
self.locator.geocode(address).latitude,
self.locator.geocode(address).longitude)

        except GeocoderTimedOut:
            return self.do_geocode(address, postalcode)
        except GeocoderQuotaExceeded:
            time.sleep(15)
            return self.do_geocode(address, postalcode)

    def return_gran(self, address, postalcode, point):
        z=self.do_geocode (address, postalcode)
        if (point==0):
            return (z[0])
        if (point==1):
            return(z[1])
        if (point==2):
            return (z[2])
        if (point==3):
            return ((self.locator.reverse(str(z[1]) +" , " +
str(z[2])).raw['address']['county']).replace(' County', ''))

    def geolocate_data(self, df):
```

```
        # Select which cols to use
        add_fields = ['street', 'streetname', 'city', 'state',
'statecode', 'zip', 'zipcode' , 'postalcode' , 'cntrycode']
        found = []
        not_found = []
        for f in add_fields:
            if f in df.columns:
                found.append(f)
            else:
                not_found.append(f)

        # Check and Clean the Fields that were Found
        null_fields=[]
        for f in found:
            if (df[f].isnull().sum(axis=0)) > 0:
                print (f, df[f].isnull().sum(axis=0))
                null_fields.append ([f, df[f].isnull().sum(axis=0)])
        for f in null_fields:
            df[f[0]].fillna('No ' + f[0], inplace = True)

        # Remove all Special Characters
        df['streetname']=df['streetname'].str.translate({ord(c): None
for c in '?!#@#$,.;-@!%^&*)('})
        df['city']=df['city'].str.translate({ord(c): None for c in
'?!@#$,.;-@!#%^&*)('})
        df['statecode']=df['statecode'].str.translate({ord(c): None for
c in '?#!@#$,.;-@!%^&*)('})

        #pad postal codes with zero to the left where only 4 digits
        df['postalcode']=df['postalcode'].astype(str).str.pad(5,
side='left', fillchar='0')

        # if no Nulls then combine the address field
        df['address'] = df['streetname'] + ', ' + df['city'] + ', ' +
df['statecode'] + ', '+ df['cntrycode'] + ', ' +
df['postalcode'].astype(str)

        # check there are no null addresses
        df['address']=df['address'].fillna(df['postalcode'].astype(str)
+ ', ' + df['cntrycode'])

        # Add geolocation
        df['Code_Level']= df.apply(lambda row: self.return_gran
(row['address'], row['postalcode'], 0), axis= 1)
        df['glat']= df.apply(lambda row: self.return_gran
(row['address'], row['postalcode'], 1), axis= 1)
        df['glon']= df.apply(lambda row: self.return_gran
(row['address'], row['postalcode'], 2), axis= 1)
        df['county']= df.apply(lambda row: self.return_gran
(row['address'], row['postalcode'], 3), axis= 1)

        # Add rowid
        df.insert(loc=0, column="rowid", value=[i+1 for i in
range(len(df.index))])
        df.set_index("rowid", inplace=True)

        return df
```

```
# Binning functions
def bin_occupancy(self, x):
    if x < 2:
        return "SingleFamily"
    else:
        return "MultiFamily"

def bin_construction(self, x):
    if x == 1:
        return "Frame"
    elif x == 2:
        return "Masonry"
    else:
        return "Concrete"

def bin_numfloors(self, x):
    if x == 0:
        return "0"
    elif x == 1:
        return "1"
    else:
        return "2+"

def bin_yearbuilt(self, x):
    if x <= 1995:
        return "<=1995"
    elif x >= 2005:
        return "2005+"
    else:
        return "1995-2005"

def bin_floorarea(self, x):
    if x < 1500:
        return "<1500"
    elif x >= 1700:
        return "1700+"
    else:
        return "1500-1700"

def bin_windspeed(self, x):
    mph = x * 2.23694
    if mph < 50:
        return 0
    elif mph <= 60:
        return 1
    elif mph <= 70:
        return 2
    elif mph <= 90:
        return 3
    elif mph <= 100:
        return 4
    elif mph <= 120:
        return 5
    elif mph <= 140:
        return 6
    else:
```

```
            return 7

    def split_prelocated(self, df):
        located = None

        # Check for pre-geolocated policies
        if self.policies is not None:
            print("Making located")
            located = pd.DataFrame([[-1 for i in range(16)]],
columns=["policynumber",

"streetname",

"city",

"statecode",

"postalcode",

"cntrycode",

"occtype",

"bldgclass",

"numfloors",

"yearbuilt",

"floorarea",

"address",

"Code_Level",

"glat",

"glon",

"county"

])
            located.set_index("policynumber", inplace=True)
            located.drop(-1)
            policy_nums = df["policynumber"].values
            for num in policy_nums:
                if num in self.policies.index:
                    located.append(self.policies.loc[num],
ignore_index=True)
                    df.drop(df.index[df["policynumber"] ==
num].tolist()[0])

            print("Located:")
            print(located)
            print()
            print("New:")
            print(df)
```

```
        return df, located

    def update_master_policies(self, df, located):

        # Append new policies to master policy list
        if self.policies is None:
            self.policies = pd.DataFrame([[-1 for i in range(16)]],
columns=["policynumber",

"streetname",

"city",

"statecode",

"postalcode",

"cntrycode",

"occtype",

"bldgclass",

"numfloors",

"yearbuilt",

"floorarea",

"address",

"Code_Level",

"glat",

"glon",

"county"

])
            self.policies.set_index("policynumber", inplace=True)
        policy_nums = df["policynumber"].values
        for num in policy_nums:
            if num not in self.policies.index:
                self.policies.append(df.loc[num])
        if -1 in self.policies.index:
            self.policies.drop(index=-1)
        print("Updtaed policy")
        print(self.policies)

        # Attach pre-geolocated policies to newly-geolocated policies
        if located is not None:
            located["rowid"] = pd.Series([i for i in
range(len(located["city"].values))])
            located.reset_index(inplace=True)
            located.set_index("rowid", inplace=True)
```

```
            df.append(located, ignore_index=True)
#            df.drop([-1])
            print("Fixing ...")
            print(df)

        return df

    def transform_data(self, df):

        # Bin occtype, bldgclass, numfloors, yearbuilt, and floorarea
        print("Binning columns")
        for i, row in df.iterrows():
            df.loc[i, "occtype"] = self.bin_occupancy(row["occtype"])
            df.loc[i, "bldgclass"] =
self.bin_construction(row["bldgclass"])
            df.loc[i, "numfloors"] =
self.bin_numfloors(row["numfloors"])
            df.loc[i, "yearbuilt"] =
self.bin_yearbuilt(row["yearbuilt"])
            df.loc[i, "floorarea"] =
self.bin_floorarea(row["floorarea"])

        # Geolocate data
        print("Geolocating addresses")
#        df, located = self.split_prelocated(df)
        df = self.geolocate_data(df)
#        df = self.update_master_policies(df, located)
        df.to_csv("%sGeoAdd_Coded.csv"%self.data_dir)

        # Get windspeeds
        print("Predicting windspeeds")
        p = Popen([self.r_dir,      # Note: Occasionally generates
error (OSError: [WinError 6] The handle is invalid) and requires kernel
restart. Something about subprocess trying to cleanup previous
subprocess executions.
                   "RWindModel.R",
                   "%sGeoAdd_Coded.csv"%self.data_dir,
#                   "katrina_tracks",
                   "%sGeoAdd_Wind.csv"%self.data_dir
                   ],
                  stdout=PIPE,
                  stderr=PIPE,
                  stdin=DEVNULL
                  )
#        (out, err) = p.communicate();
#        print ("\ncat returned code = %d\n" % p.returncode)
#        print ("cat output:\n%s\n" % out)
#        print ("cat errors:\n%s\n" % err)
        df = pd.read_csv("%sGeoAdd_Wind.csv"%self.data_dir)
        df["windspeed"] = pd.Series([i for i in range(len(df.index))])
        for i, row in df.iterrows():
            df.loc[i, "windspeed"] =
self.bin_windspeed(row["vmax_gust"])

        # Split dataframe
        x = df[["occtype", "bldgclass", "yearbuilt", "numfloors",
"floorarea"]]
```

```
        x_wind = df[["windspeed"]]

        # Encode data
        print("Encoding data")
        x_hot = self.encoder.transform(x).toarray()
        x_wind = x_wind.to_numpy()

        # Attach windspeed
        x = []
        for i, r in enumerate(x_hot):
            row = []
            for j in r:
                row.append(j)
            row.append(x_wind[i][0]*1.0)
            x.append(row)

        return x

    def predict_losses(self, input_data_dir):

        # Load data
        inp = pd.read_csv(input_data_dir)
        inp.columns = map(str.lower, inp.columns)
        print("Data loaded")

        # Geolocate, get windspeed, and encode
        print("Transforming data")
        inp = self.transform_data(inp)

        # Predict loss
        print("Making predictions")
        predictions = self.model.predict(inp)

        # Attach predictions to policies.
        df = pd.read_csv("%sGeoAdd_Wind.csv"%self.data_dir)
        predictions = [i[0] for i in predictions]
        df["loss_prediction"] = pd.Series(predictions)
        df.set_index("rowid", inplace=True)
        df.to_csv("%sGeoAdd_Predictions.csv"%self.data_dir)
        return df

#c = HurricaneLosses(d)
#p = c.predict_losses("%sGeoAdd.csv"%data_d)
#print(p)
```