DEPARTMENT OF CIVIL, ENVIRONMENTAL AND GEOMATIC ENGINEERING

Semester Project Report

Data-driven identificiation and classification of rail surface defectse

Aiyu Liu

Supervised by: Cyprien Hoelzl, Prof. Eleni Chatzi

Saturday 8th February, 2020

Acknowledgements

This semester project would not be possible without the help of my supervisors, Cyprien Hoelzl and professor Eleni Chatzi. I first reached out to professor Chatzi for a semester project opportunity during ETH week

Working alongside Cyprien has been a great experience.

I received all the guidance necessary Whenever I had issues I could always ask and the reply would come promptly provided me with many informative resources very good at explaining concepts very smart and very specialised in this field – huge understanding Can ask any questions, down-to-earth and very helpful. I could not ask for a better supervisor.

Chatzi is very approachable and kind, good at providing feedback at the intermediate sessions.

Contents

Chapter 1

Introduction

1.1 Problem description and motivation

Railway companies need to continuously and sufficiently maintain the train tracks and optimally detect defects in order to have a more punctual and more effective train system. However, the current system is expensive, time consuming and ineffective. That is, maintenance agents need to walk along tracks and check them for defects. For visualisation purposes, there is roughly 5200 km of rails in Switzerland which needs to be inspected by 40 experienced inspectors.

maybe remove this section

In order to cope with this issue, Swiss Federal Railways (SBB) has specifically built two new special diagnostic vehicles (SDV) designed for defect identification among other purposes. For this, two accelerometers have been installed at the front and back of the vehicle to collect the signal responses from the wheel and the train track

insert picture, mention boogey?

A defect in train tracks can be seen as a discontinuity. As a train passes over this discontinuity, it will result in a perturbation that can be detected by sensors. It is our main assumption that each type of defect will have a specific signature that will allow its identification and classification. This is similar to the idea presented in

https://blog-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff801a2cf

By succesfully identifying and classifying the defects, we take one step further towards reducing delays and making trains more punctual and reliable. The first step in this process consists of identification and classification, while the second step consists of future defect prediction.

1.2 Objective

As the title implies, the objective of this project is to identify and classify rail surface defects.

apply machine learning techniques on the problem

1.3 Defects

Evidently, a defect can be seen as a deviation from the standard train track. For the exact defect type, SBB has self-constructed a database for the individual defect definitions . Here is a few examples:

is this a recognized system?

insert pictures

Generally, a defect is separated into two overarching types: range- and point-defects. I.e. a defect that is detected at a single point versus a defect that is detected at varying give example

show signal types?

See list of defect types in appendix?

lengths – e.g. .

For this project, we have solely focused on the point defects for analysis, as this simplifies the problem statement. . A point defect is perceived as a sharp signal response, whereas a range-defect is perceived over a greater time period. We thus disregard range-defects such that we do not have to deal with the extra, associated factors.

1.4 Data

The data has been collected and provided by SBB. Using their SDV, SBB has made trips back and forth to different cities in Switzerland in order to collect various data including but not limited to accelerometer data. After getting the data from SBB, it then goes through a processing pipeline (designed by Cyprien), after which the data can be manipulated with `python` dataframes (from `pd.DataFrame`). The accelerometer captures the accelerations at the XYZ-axes (along with the timestamps at each recording), of which we are only concerned with the Y-axis for the vertical pertubations.

make a table of the equipment and sample frequencies

Furhtermore, the locations of the defects have to be retrieved from SBB's database. which were retrieved by Cyprien.

We need to define terminlogy of these: defects, ins joints = in the following we will use defect as an umbrella term for these entities.

1.5 Code

Which data did I work on, put in tables, switches, ins joints and defects

Brief explanation of the code?

The code is written purely in python. The code can be found on github: https://github.com/Aiyualive/SemesterProject2.0.

Chapter 2

Design and Implementation

For the process of defect classification we employed the following pipeline:

insert pipeline picture

In the following, I would like to give an overview of how these was implemented

which track entities are we actually analysing

2.1 Shift of GPS timestamps

The SDV has its GPS sensor installed at a specified location on the vehicle body. However, what we need to achieve is the position (covered distance) at each accelerometer at either sides of the GPS. Since the GPS sensor is sampled at a lower frequency compared to the accelerometers, we first need to get the corresponding positions for each accelerometer sample. This is done by interpolation using the timestamps of the accelerometers and GPS.

show a few defects and their signals

Depending on the direction of the vehicle we then subtract/add the offset between the accelerometers and the GPS sensor with regard to the position of these sensors on the vehicle body.

appendix for more signals?

2.2 Peak windows

Retrieving the signal response around the defect location forms a crucial aspect in the overarching pipeline. The goal of this step is to, around each defect, create a "window" containing accelerometer accelerations of a specified time length – wherein Within the highest acceleration recording around is found in the center. As a result, all of these windows would be uniform in the sense that they are all centered according to the highest recording of a defect. It is then assumed that each window forms the signature of each track entity.

insert drawing of how it is calculated?

Since we are assuming that each track entity is identified by a well-formed peak, we first need to find this peak within a reasonable offset from the defect location, after which we center around that within another reasonable offset.

In the code, this is done by defining two parameters: `find_peak_offset = 1` and `window_offset = 0.5`. I.e. given a defect timestamp, we search for the highest acceleration recording that has occured 1 second after and 1 second before the defect timestamp. Once the peak has been found, we then center it in a 1 second window (0.5 sec on each side).

insert a drawing of how this works?

First finds the highest peak within a peak finding window. Then this highest peak is centered by defining a window offset. Then we get the start and end index of this window These indexes are then used to index the timestamps and acceleration for the axle This was altered.

find

2.3 Neural network architecture

Trained a neural network, although we were only able to achieve max

Based on the analysis we

2.4 Visualisation

This step should have been done first

Chapter 3

Evaluation

3.1 Results

insert table for different architectures

3.2 Discussion

I tried to increase the outliers, but this was a hugely naive approach

Chapter 4

Conclusion and future work

4.1 Conclusion

4.2 Future work

- Might be interesting to also consider the XZ-axes.
- Line defects
- tune the peak finding parameters
- track entity dependent/specific window offsets
- we must not set the findpeakoffset too high

4.3 TODO

- very fast speed, overlap between switch and ins, old vs new rail, ax1 arrow 2 arrow 3 arrow 4
- 3D plots?
- change the defect library to use pandas instead?
- visualise what the network is doing using Harry's code
- use speed as a feature also
- be consistent with function naming and variable names
-

4.4 Notes

whats this

```
def conv(df):
    """
    has to be series
    """
    return np.vstack([v for v in df])

dup_ins = s_features.ins_joints.copy()[['accelerations']]
dup_swi = s_features.switches.copy()[['accelerations']]
dup_def = s_features.defects.copy()[['accelerations']]

dup_ins['accelerations'] = np.sum(conv(dup_ins.accelerations),1)
dup_swi['accelerations'] = np.sum(conv(dup_swi.accelerations),1)
dup_def['accelerations'] = np.sum(conv(dup_def.accelerations),1)

# s_features.ins_joints[['vehicle_speed(m/s)', 'Axle', 'campagin_ID']].duplicated()

idx_ins = dup_ins.accelerations.duplicated()
idx_swi = dup_swi.accelerations.duplicated()
idx_def = dup_def.accelerations.duplicated()
new_ins = s_features.ins_joints[~idx_ins]
new_swi = s_features.switches[~idx_swi]
new_def = s_features.switches[~idx_def]

print("Duplcated samples: ", len(dup_ins) - len(new_ins))
print("Duplcated samples: ", len(dup_swi) - len(new_swi))
print("Duplcated samples: ", len(dup_def) - len(new_def))

# Load weight example
# Could just save entire model and then load entire model
# Could also make this into a function
clf2 = NN(N_FEATURES, N_CLASSES)
clf2.prepare_data(X, y)
clf2.make_model2()
clf2.load_weights('model_01-12-2019_150004.hdf5')
clf2.predict() ### on validation set
clf2.measure_performance(accuracy_score)
```

Test sample

```
ii = pd.DataFrame([
    [np.array([1,2]),2],
    [np.array([1,2]),2],
    [np.array([1,2]),2]])
```

```
x = a
[u,I,J] = unique(x, 'rows', 'first')
hasDuplicates = size(u,1) < size(x,1)
ixDupRows = setdiff(1:size(x,1), I)
dupRowValues = x(ixDupRows,:)

s_features.ins_joints.timestamps[:2].duplicated()
```

Chapter 5

Appendix

Figure out references

New paper with train

Appendix A

Appendix

```
import numpy as np
import pandas as pd
from scipy.signal import find_peaks
from tqdm import tqdm

class featureset():
    """
    Generate dataframe containing features for classification
    """
    def __init__(self, obj, peak_offset=1, window_offset=0.5):
        self.peak_offset   = peak_offset
        self.window_offset = window_offset
        self.defects      = makeDefectDF(obj,
                                        peak_offset=peak_offset,
                                        window_offset=window_offset)
        self.switches     = makeGenericDF(obj, "switches",
                                          peak_offset=peak_offset,
                                          window_offset=window_offset)
        self.ins_joints   = makeGenericDF(obj, "insulationjoint",
                                          peak_offset=peak_offset,
                                          window_offset=window_offset)

    def makeDefects(self, obj):
        self.defect11     = makeDefectDF(obj, "AXLE_11")
        self.defect12     = makeDefectDF(obj, "AXLE_12")
        self.defect41     = makeDefectDF(obj, "AXLE_41")
        self.defect42     = makeDefectDF(obj, "AXLE_42")
        self.defects      = pd.concat([self.defect11,
                                       self.defect12,
                                       self.defect41,
                                       self.defect42])

        return self.defects

    def makeSwitches(self, obj):
        """
        DEPRECATED
        """
        self.switches11   = makeSwitchesDF(obj, "AXLE_11")
        self.switches12   = makeSwitchesDF(obj, "AXLE_12")
        self.switches41   = makeSwitchesDF(obj, "AXLE_41")
        self.switches42   = makeSwitchesDF(obj, "AXLE_42")
        self.switches     = pd.concat([self.switches11,
                                       self.switches12,
                                       self.switches41,
                                       self.switches42])
        return self.switches

    def makeInsJoints(self, obj):
        """
        DEPRECATED
        """
        self.ins_joints11 = makeInsulationJointsDF(obj, "AXLE_11")
        self.ins_joints12 = makeInsulationJointsDF(obj, "AXLE_12")
```

```
        self.ins_joints41 = makeInsulationJointsDF(obj, "AXLE_41")
        self.ins_joints42 = makeInsulationJointsDF(obj, "AXLE_42")
        self.ins_joints   = pd.concat([self.ins_joints11,
                                       self.ins_joints12,
                                       self.ins_joints41,
                                       self.ins_joints42])
        return self.ins_joints

def findIndex(timestamps, start, end):
    """
    Given starting and ending time timestamps it returns the indexes
    of the closest timestamps in the first arg
    params:
        timestamps: timestamps array to search within
        start, end: timestamps to be within start and end
    """
    # Finds all indexes which satisfy the condition
    # nonzero gets rid of the non-matching conditions
    indexes = np.nonzero((timestamps >= start) & ( timestamps < end))[0]

    return indexes

def findVehicleSpeed(time, obj):
    """
    Gets the vehicle speed closest to the specified time.
    params:
        time: time at which to get the vehicle speed
        speed_df: needs to be obj.MEAS_DYN.VEHICLE_MOVEMENT_1HZ
    """
    speed_df = obj.MEAS_DYN.VEHICLE_MOVEMENT_1HZ
    speed_times = speed_df['DFZ01.POS.VEHICLE_MOVEMENT_1HZ.timestamp'].values
    speed_values = speed_df['DFZ01.POS.VEHICLE_MOVEMENT_1HZ.SPEED.data'].values

    # Minus 1 since using > and we want value before
    bef = np.nonzero(speed_times > time)[0][0] - 1
    aft = bef + 1

    # Finds the closest timestamp
    idx = np.argmin([abs(speed_times[bef] - time), abs(speed_times[aft] - time)])
    closest = bef + idx # plus 0 for bef, plus 1 for after

    speed = speed_values[closest]

    return speed

def getPeakWindow(von, bis, find_peak_offset, window_offset, acc_time, a):
    """
    First finds the highest peak within a peak finding window.
    Then this highest peak is centered by defining a window offset.
    Then we get the start and end index of this window
    These indexes are then used to index the timestamps and acceleration for the axle
    params:
        von, bis: the start and end of a defect
        find_peak_offset, window_offset:
            the offset of which to search for peak, and the size of the actual
            defect window
        acc_time, a:
            all the accelerationn times and corresponding acceleratoins
    OBS:
        use of np.argmax() since find_peaks() does not work consistently if duplicate heights.
    alternative:
        to findIndexes
        acc_window = a_df[(aaa[time_label] >= von - find_peak_offset) &
                          (aaa[time_label] < bis + find_peak_offset)]
        but current method is faster
    """

    # Accounting for shift between von and bis
    if von > bis:
        tmp = von
        von = bis
        bis = tmp

```

```
    # Find all indexes contained within the peak searching window
    indexes = findIndex(acc_time,
                        von - find_peak_offset,
                        bis + find_peak_offset)

    # Get highest peak
    peak_idx = np.argmax(a[indexes]) + indexes[0]

    # Center the peak
    start = int(peak_idx - window_offset)
    end   = int(peak_idx + window_offset)
    if (start < 0) or (end > len(acc_time)):
        raise Warning("Out of bounds for peak centering")

    timestamps    = acc_time[start:end]
    accelerations = a[start:end]
    return timestamps, accelerations

def getSeverity(severity):
    """
    Converts the recorded severity into integer codes
    """
    if 'sehr' in severity:
        return 1
    elif 'hoch' in severity:
        return 2
    elif 'mittel' in severity:
        return 3
    elif 'gering' in severity:
        return 4
    else:
        return -1 # undefined

def getDirection(obj):
    """
    Gets the driving direction of the vehicle for a measurement ride
    """
    direction_label = 'DFZ01.POS.FINAL_POSITION.POSITION.data.direction'
    direction = np.unique(obj.MEAS_DYN.POS_FINAL_POSITION[[direction_label]])

    if len(direction) == 1:
         direction = direction[0]
    else:
        raise Warning("Driving direction not unique")
    return direction

def getSwitchComponent(obj):
    """
    Adds the vehicle direction and returns the switch DataFrame
    """
    component=obj.MEAS_POS.POS_TRACK[obj.MEAS_POS.POS_TRACK['TRACK.data.switchtype']==1]
    df_postrack = component.copy()
    df_postrack['TRACK.data.direction_vehicleref'] = df_postrack['TRACK.data.direction']
    cond_left  = (df_postrack['TRACK.data.direction']=='left') & (df_postrack['DFZ01.POS.FINAL_POSITION.POSITION.data.kilome
    cond_right = (df_postrack['TRACK.data.direction']=='right')& (df_postrack['DFZ01.POS.FINAL_POSITION.POSITION.data.kilome
    df_postrack.loc[cond_left, 'TRACK.data.direction_vehicleref']  = 'right'
    df_postrack.loc[cond_right, 'TRACK.data.direction_vehicleref'] = 'left'
    return df_postrack

def makeDefectDF(obj, axle='all', find_peak_offset=1, window_offset=0.5):
    """
    Makes the defect dataframe containing all relevant features.
    params:
        obj: the gdfz measurement ride
        axle: axle for which to find defect
        peak_offset: time in seconds for which to find the highest peak around a defect
        window_offset: time in seconds for which to center around the highest peak
    """

    if axle == 'all':
        axle = ['AXLE_11', 'AXLE_12', 'AXLE_41', 'AXLE_42']
    else:
        axle = [axle]
```

```
    defect_type_names = np.unique(obj.ZMON['ZMON.Abweichung.Objekt_Attribut'])

    d_df      = pd.DataFrame()
    nanosec   = 10**9
    samp_freq = 24000 # per sec
    window_offset = window_offset * samp_freq

    driving_direction = getDirection(obj)

    for ax in axle:
        dict_def_n  = dict.fromkeys(defect_type_names, 0)
        defectToClass   = {defect_type_names[i] : (i + 2)
                           for i in range(len(defect_type_names))}

        time_label     = 'DFZ01.DYN.ACCEL_AXLE_T.timestamp'
        acc_label      = 'DFZ01.DYN.ACCEL_AXLE_T.Z_' + ax + '_T.data'
        acc_time = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[time_label].values
        acc      = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[acc_label].values

        columns = ["timestamps", "accelerations", "window_length(s)",
                   "severity", "vehicle_speed(m/s)", "axle",
                   "campagin_ID", "driving_direction",
                   "defect_type", "defect_length(m)", "line, defect_ID",
                   "class_label"]

        for i, row in tqdm((obj.ZMON).iterrows(), total = len(obj.ZMON), desc="ZMON " + ax):
            von      = row['ZMON.gDFZ.timestamp_von.' + ax[:6]]
            bis      = row['ZMON.gDFZ.timestamp_bis.' + ax[:6]]

            # For detecting point or range defect
            interval  = abs(int(von) - int(bis))/nanosec
            if interval == 0:
                # Point defects
                find_peak_offset = find_peak_offset * nanosec
                vehicle_speed    = findVehicleSpeed(von, obj)
            else:
                ### Just using von and bis
                find_peak_offset = 0
                # Vehicle speed is found at the middle of the interval
                midpoint         = int(( von + bis)/2 )
                vehicle_speed    = findVehicleSpeed(midpoint, obj)

            timestamps, acceleration = getPeakWindow(von, bis,
                                                     find_peak_offset, window_offset,
                                                     acc_time, acc)

            # Each defect type number count
            d_type              = row['ZMON.Abweichung.Objekt_Attribut']
            n                   = dict_def_n[d_type]
            dict_def_n[d_type] = n + 1

            window_length = (timestamps[-1] - timestamps[0]) / nanosec
            severity      = getSeverity(row['ZMON.Abweichung.Dringlichkeit'])
            #print(d_type, row['ZMON.Abweichung.Dringlichkeit'])
            identifier    = (row['ZMON.Abweichung.Linie_Nr'], row['ZMON.Abweichung.ID'])
            defect_length = interval * vehicle_speed

            temp_df = pd.DataFrame([[timestamps, acceleration, window_length,
                                     severity, vehicle_speed, ax,
                                     obj.campaign, driving_direction,
                                     d_type, defect_length, identifier,
                                     defectToClass[d_type]]],
                                   index    = [d_type + "_" + str(n) + "_" + ax],
                                   columns = columns)

            d_df = pd.concat([d_df, temp_df], axis=0)

    return d_df

def makeGenericDF(obj, type, axle='all', peak_offset=1, window_offset=0.5):
    if axle == 'all':
```

```
        axle = ['AXLE_11', 'AXLE_12', 'AXLE_41', 'AXLE_42']
    else:
        axle = [axle]

    # Offsets
    nanosec = 10**9
    sampling_freq = 24000
    window_offset = window_offset * 24000
    peak_offset = peak_offset * nanosec

    # datarame
    df = pd.DataFrame()
    driving_direction = getDirection(obj)

    for ax in axle:
        columns = ["timestamps", "accelerations", "window_length(s)",
                   "severity", "vehicle_speed(m/s)", "axle",
                   "campagin_ID", "driving_direction"]

        ### DEFECT ###
        if type == 'defect':
            raise Warning("Not yet implemented for defects")

        ### INSULATION JOINT ###
        elif type == 'insulationjoint':
            COMPONENT  = obj.DfA.DFA_InsulationJoints
            time_label = "DfA.gDFZ.timestamp." + ax[:-1]
            columns.extend(["ID", "class_label"])

        ### SWITCHES ###
        elif type == 'switches':
            COMPONENT = getSwitchComponent(obj)
            time_label = "DFZ01.POS.FINAL_POSITION.timestamp." + ax[:-1]
            columns.extend(["crossingpath", "track_name",
                            "track_direction", "switch_ID", "class_label"])

        # Accelerometer accelerations
        acc_time_label = 'DFZ01.DYN.ACCEL_AXLE_T.timestamp'
        acc_label  = 'DFZ01.DYN.ACCEL_AXLE_T.Z_' + ax + '_T.data'
        acc_time   = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[acc_time_label].values
        acc        = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[acc_label].values

        count = 0
        for i, row in tqdm(COMPONENT.iterrows(), total = len(COMPONENT), desc=type + " " + ax):
            timestamp = row[time_label]

            if np.isnan(timestamp):
                continue

            timestamps, accelerations = getPeakWindow(
                                            timestamp, timestamp,
                                            peak_offset, window_offset,
                                            acc_time, acc)

            window_length = (timestamps[-1] - timestamps[0]) / nanosec
            severity = 5
            vehicle_speed = findVehicleSpeed(timestamp, obj)

            features = [timestamps, accelerations, window_length,
                        severity, vehicle_speed, ax,
                        obj.campaign, driving_direction]

            ### INSULATION JOINT ###
            if type == 'insulationjoint':
                ID            = row["DfA.IPID"]
                class_label   = 0
                features.extend([ID, class_label])

            elif type == 'switches':
                # timestamp is start_time
                # end_time   = row[ax_time_label] + row[end_time_label] - row[timestamp_label]
                switch_id  = row['TRACK.data.gtgid']
                track_name = row['TRACK.data.name']
```

```
                track_direction = row['TRACK.data.direction_vehicleref']
                crossingpath = str(row["crossingpath"])
                class_label = 1
                features.extend([crossingpath, track_name,
                                track_direction, switch_id, class_label])

            temp_df = pd.DataFrame([features],
                                   index    = [type + "_" + str(count) + "_" + ax],
                                   columns = columns)

            df = pd.concat([df, temp_df], axis=0)
            count += 1

    return df

def savePickle(campaign_objects, identifier, path="AiyuDocs/pickles/"):
    """
    campaign_objects: list of objects

    """
    defects    = pd.DataFrame()
    ins_joints = pd.DataFrame()
    switches   = pd.DataFrame()

    for o in campaign_objects:
        defects = pd.concat([defects, o.defects])
        ins_joints = pd.concat([ins_joints, o.ins_joints])
        switches = pd.concat([switches, o.switches])

    defects.to_pickle(path + identifier + "_defects_df.pickle")
    switches.to_pickle(path + identifier + "_switches_df.pickle")
    ins_joints.to_pickle(path + identifier + "_ins_joints_df.pickle")

##################
### DEPRECATED ###
##################

def makeSwitchesDF(obj, axle):
    """
    DEPRECATED
    Makes a dataframe of ordinary switches and
    params:
        axle: the desired axle channel to work with
    """
    switches = obj.MEAS_POS.POS_TRACK[obj.MEAS_POS.POS_TRACK['TRACK.data.switchtype']==1]

    # The start time of my switch with respect to axle1:
    ax_time_label = 'DFZ01.POS.FINAL_POSITION.timestamp.' + axle[:-1]
    timestamp_label = 'DFZ01.POS.FINAL_POSITION.timestamp'
    end_time_label = 'DFZ01.POS.FINAL_POSITION.timestamp_end'

    time     = 'DFZ01.DYN.ACCEL_AXLE_T.timestamp'
    acc      = 'DFZ01.DYN.ACCEL_AXLE_T.Z_' + axle + '_T.data'
    acc_time = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[time].values
    a        = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[acc].values

    normal_df = pd.DataFrame()
    switches = obj.MEAS_POS.POS_TRACK[obj.MEAS_POS.POS_TRACK['TRACK.data.switchtype']==1]
    switches_time_label  = "DFZ01.POS.FINAL_POSITION.timestamp." + axle[:-1]

    nanosec = 10**9
    find_peak_offset = 1 * nanosec
    window_offset = 12000

    columns = ["timestamps",
               "accelerations",
               "window_length(s)",
               "severity",
               "vehicle_speed(m/s)",
               "crossingpath",
               "driving_direction",
               "axle",
               "class_label"]
```

```python

    driving_direction = getDirection(obj)

    count = 0
    for i, row in tqdm(switches.iterrows(), total = len(switches), desc="Switches " + axle):

        start_time = row[ax_time_label]
        end_time   = row[ax_time_label] + row[end_time_label] - row[timestamp_label]

        switches_time = row[switches_time_label]

        if np.isnan(switches_time):
            continue

        timestamps, accelerations = getPeakWindow(switches_time, switches_time,
                                     find_peak_offset, window_offset,
                                     acc_time, a)

        severity = 5
        vehicle_speed = findVehicleSpeed(switches_time, obj)
        actual_window_length = (timestamps[-1] - timestamps[0]) / nanosec
        crossingpath = str(row["crossingpath"])
        class_label = 1

        temp_df = pd.DataFrame([[timestamps,
                                 accelerations,
                                 actual_window_length,
                                 severity,
                                 vehicle_speed,
                                 crossingpath,
                                 driving_direction,
                                 axle,
                                 class_label]],
                            index = ["Switches" + "_" + str(count)],
                            columns = columns)

        normal_df = pd.concat([normal_df, temp_df], axis=0)
        count += 1

    return normal_df

def makeInsulationJointsDF(obj, axle, find_peak_offset=1, window_offset=0.5):
    """
    DEPRECATED
    Makes the defect dataframe containing all relevant features.
    params:
        axle: axle for which to find defect
        peak_height: this height determines the peak classification
    """
    time     = 'DFZ01.DYN.ACCEL_AXLE_T.timestamp'
    acc      = 'DFZ01.DYN.ACCEL_AXLE_T.Z_' + axle + '_T.data'
    acc_time = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[time].values
    a        = obj.MEAS_DYN.DFZ01_DYN_ACCEL_AXLE_T[acc].values

    normal_df = pd.DataFrame()
    dfa       = obj.DfA.DFA_InsulationJoints
    insulation_time_label  = "DfA.gDFZ.timestamp." + axle[:-1]

    nanosec = 10**9
    sampling_freq = 24000
    window_offset = window_offset * 24000
    find_peak_offset = find_peak_offset * nanosec

    columns = ["timestamps",
               "accelerations",
               "window_length(s)",
               "severity",
               "vehicle_speed(m/s)",
               "ID",
               "axle",
               "class_label"]

    driving_direction = getDirection(obj)
```

```
    count = 0
    for i, row in tqdm(dfa.iterrows(), total = len(dfa), desc="Insulation Joints " + axle):
        insulation_time = row[insulation_time_label]

        timestamps, accelerations = getPeakWindow(insulation_time, insulation_time,
                                     find_peak_offset, window_offset,
                                     acc_time, a)

        actual_window_length = (timestamps[-1] - timestamps[0]) / nanosec
        severity = 5
        vehicle_speed = findVehicleSpeed(insulation_time, obj)
        ID             = row["DfA.IPID"]
        class_label    = 0

        temp_df = pd.DataFrame([[timestamps,
                                 accelerations,
                                 actual_window_length,
                                 severity,
                                 vehicle_speed,
                                 ID,
                                 driving_direction,
                                 axle,
                                 class_label]],
                            index = ["InsulationJoint" + "_" + str(count)],
                            columns = columns)

        normal_df = pd.concat([normal_df, temp_df], axis=0)
        count += 1

    return normal_df
```