homework2

September 21, 2019

NAME: **Zhi Li**

ID: **113523595**

1 Homework 2

1.0.1 Objectives

- Object orientation in Python
- Constructing Data Pre-processing Pipelines
  - Imputing
  - Filtering
  - Simple Numerical Methods
- Do not save work within the ml_practices folder
  - create a folder in your home directory for assignments, and copy the templates there

1.0.2 General References

- Sci-kit Learn Pipelines
- Sci-kit Learn Impute
- Sci-kit Learn Preprocessing
- Pandas Interpolate
- Pandas fillna()

[34]:
```
import pandas as pd
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin

FIGWIDTH = 10
FIGHEIGHT = 2
FONTSIZE= 15

%matplotlib inline
```

2 LOAD DATA

[35]:
```
fname = '~/ml_practices/imports/datasets/baby1/subject_k1_w10_hw2.csv'
baby_data_raw = pd.read_csv(fname)# TODO
baby_data_raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15000 entries, 0 to 14999
Data columns (total 7 columns):
time             15000 non-null float64
left_wrist_x     13458 non-null float64
left_wrist_y     13454 non-null float64
left_wrist_z     13454 non-null float64
right_wrist_x    13514 non-null float64
right_wrist_y    13514 non-null float64
right_wrist_z    13514 non-null float64
dtypes: float64(7)
memory usage: 820.4 KB
```

[36]:
```
""" TODO
Call describe() on the data to get summary statistics
"""
baby_data_raw.describe()
```

[36]:
```
               time  left_wrist_x  left_wrist_y  left_wrist_z  right_wrist_x  \
count  15000.000000  13458.000000  13454.000000  13454.000000   13514.000000
mean     149.990000      0.243580      0.162076     -0.044767       0.271218
std       86.605427      0.084823      0.093114      0.060566       0.055190
min        0.000000      0.027525     -0.046680     -0.186060       0.081230
25%       74.995000      0.177911      0.096319     -0.082849       0.238649
50%      149.990000      0.251879      0.154445     -0.045112       0.277340
75%      224.985000      0.308732      0.245144     -0.004720       0.314673
max      299.980000      0.389957      0.334027      0.147053       0.396959

       right_wrist_y  right_wrist_z
count   13514.000000   13514.000000
mean       -0.120768      -0.207248
std         0.047123       0.054263
min        -0.275120      -0.311197
25%        -0.140773      -0.245453
50%        -0.111330      -0.216992
75%        -0.085764      -0.158773
max        -0.040851      -0.007693
```

[37]:
```
""" TODO
Call head() on the data to observe the first few examples
"""
```

```
baby_data_raw.head()
```

[37]:
```
   time  left_wrist_x  left_wrist_y  left_wrist_z  right_wrist_x  \
0  0.00           NaN      0.293503     -0.092803       0.314738
1  0.02           NaN      0.293445     -0.092968       0.315143
2  0.04           NaN           NaN           NaN       0.315974
3  0.06           NaN      0.293285     -0.093356       0.316709
4  0.08      0.163611      0.293237     -0.093475       0.317206

   right_wrist_y  right_wrist_z
0      -0.113438      -0.154972
1      -0.113476      -0.154807
2      -0.113521      -0.154429
3      -0.113555      -0.154063
4      -0.113534      -0.153886
```

[38]:
```
""" TODO
Call tail() on the data to observe the last few examples
"""
baby_data_raw.tail()
```

[38]:
```
         time  left_wrist_x  left_wrist_y  left_wrist_z  right_wrist_x  \
14995  299.90      0.371656           NaN           NaN       0.202332
14996  299.92      0.371723           NaN           NaN       0.202157
14997  299.94      0.371801           NaN           NaN       0.201895
14998  299.96      0.371866           NaN           NaN       0.201533
14999  299.98      0.371907           NaN           NaN       0.201166

       right_wrist_y  right_wrist_z
14995      -0.073395      -0.310776
14996      -0.073288      -0.310726
14997      -0.073102      -0.310798
14998      -0.072929      -0.310848
14999      -0.072672      -0.310929
```

[39]:
```
""" TODO
Display the column names for the data
"""
print('columns: ',baby_data_raw.columns)
```

```
columns:  Index(['time', 'left_wrist_x', 'left_wrist_y', 'left_wrist_z',
'right_wrist_x',
       'right_wrist_y', 'right_wrist_z'],
      dtype='object')
```

[40]:
```
""" TODO
Determine whether any data are NaN. Use isna() and
```

```
any() to obtain a summary of which features have at
least one missing value
"""
print(baby_data_raw.isna().any())
```

```
time              False
left_wrist_x       True
left_wrist_y       True
left_wrist_z       True
right_wrist_x      True
right_wrist_y      True
right_wrist_z      True
dtype: bool
```

3 Create Pipeline Elements

In the lecture, some of the Pipeline components might have taken in or returned numpy arrays and others pandas DataFrames. For this assignment, transform methods for all the Pipeline components will take input as a pandas DataFrame and return a DataFrame.

[41]:
```
a= [1,2,3,4,5]
a[2:-1]
```

[41]:
```
[3, 4]
```

[42]:
```
""" PROVIDED
Pipeline component object for selecting a subset of specified features
"""
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribs):
        self.attribs = attribs

    def fit(self, x, y=None):
        return self

    def transform(self, X):
        '''
        PARAMS:
            X: is a DataFrame
        RETURNS: a DataFrame of the selected attributes
        '''
        return X[self.attribs]


""" TODO
Complete the Pipeline component object for interpolating and filling in
```

```
gaps within the data. Whenever data are missing inbetween valid values,
use interpolation to fill in the gaps. For example,
    1.2 NaN NaN 1.5
becomes
    1.2 1.3 1.4 1.5

Whenever data are missing on the edges of the data, fill in the gaps
with the first available valid value. For example,
    NaN NaN 2.3 3.6 3.2 NaN
becomes
    2.3 2.3 2.3 3.6 3.2 3.2
The transform() method should fill in the holes and the edge cases.
"""
class InterpolationImputer(BaseEstimator, TransformerMixin):
    def __init__(self, method='quadratic'):
        self.method = method

    def fit(self, x, y=None):
        return self

    def transform(self, X): # TODO
        '''
        PARAMS:
            X: is a DataFrame
        RETURNS: a DataFrame without NaNs
        '''
        # TODO: Interpolate holes within the data
        X= X.apply(lambda f: f.interpolate(method=self.method,␣
↪limit_area='inside'), axis=0)
        # TODO: Fill in the NaNs on the edges of the data
        X.fillna(method= 'ffill', inplace=True)
        X.fillna(method='bfill', inplace= True)
        # TODO: return the imputed dataframe

        return X

""" TODO
Complete the Pipeline component object for smoothing specific features
using a gaussian kernel. Use the following formula to apply the filter:
    x'[t] = ( w[0]*x[t-3] + w[1]*x[t-2] + w[2]*x[t-1] + w[3]*x[t]
           + w[4]*x[t+1] + w[5]*x[t+2] + w[6]*x[t+3])
    DISCLAIMER: if you implement this computation on more than one line,
                make sure to place parentheses around the entire expression
                such that the interpreter reads the lines as all part of
                one expression
This can be implemented similarly to how the derivative is computed.
Additionally, pad both ends of x with three instances of the adjacent
```

```
value, before filtering, to maintain the original signal length and
smoothness. For example,
                1.3 2.1 4.4 4.1 3.2
would be padded as
    1.3 1.3 1.3 1.3 2.1 4.4 4.1 3.2 3.2 3.2 3.2
"""

def computeweights(length=3, sig=1):
    '''
    Computes the weights for a Gaussian filter kernel
    PARAMS:
        length: the number of terms in the filter kernel
        sig: the standard deviation (i.e. the scale) of the Gaussian
    RETURNS: a list of filter weights for the Gaussian kernel
    '''
    x = np.linspace(-2.5, 2.5, length)
    kernel = stats.norm.pdf(x, scale=sig)
    return kernel / kernel.sum()

class GaussianFilter(BaseEstimator, TransformerMixin):
    def __init__(self, attribs=None, kernelsize=3, sig=1):
        self.attribs = attribs
        self.kernelsize = kernelsize
        self.sig = sig
        self.weights = computeweights(length=kernelsize, sig=sig)
        print("KERNEL WEIGHTS", self.weights)

    def fit(self, x, y=None):
        return self

    def transform(self, X): # TODO
        '''
        PARAMS:
            X: is a DataFrame
        RETURNS: a DataFrame with the smoothed signals
        '''
        def gfilter(x):
            '''
            inner function helper to use apply method in pandas

            Inputs:
            -------------
            :x - pandas.Series object; column-wise feature

            Returns:
            -------------
            :new_arr - pd.Series object; series after gaussian filter
```

```
            '''
            new_arr= x.copy().values
            it= x//self.kernelsize
            y= np.zeros((x.values[self.kernelsize//2:-self.kernelsize//2+1].
 ↪shape))
            for i in range(self.kernelsize):
#                 print(len(new_arr[i:-self.kernelsize+i+1]), len(y))
                y+= self.weights[i]*new_arr[i:-self.kernelsize+i+1] if i<self.
 ↪kernelsize-1 else self.weights[i]*new_arr[i:]

            x.iloc[self.kernelsize//2:-self.kernelsize//2+1]= y

            return x

        w = self.weights
        Xout = X.copy()
        if self.attribs == None:
            self.attribs = Xout.columns

        # TODO for each attribute:
            # TODO: pad the data
        upper= pd.concat([Xout.iloc[[0],:]]*(self.kernelsize//2))
        lower= pd.concat([Xout.iloc[[-1],:]]*(self.kernelsize//2))
        Xout= pd.concat([upper, Xout, lower])
        Xout.reset_index(inplace=True) #ajust the index
            # TODO: filter the data
        Xout= Xout.apply(gfilter, axis=0) #column-wise apply
        # TODO: return filtered dataframe

        return Xout


""" PROVIDED
Pipeline component object for computing the derivative for specified features
"""
class DerivativeComputer(BaseEstimator, TransformerMixin):
    def __init__(self, attribs=None, prefix='d_', dt=1.0):
        self.attribs = attribs
        self.prefix = prefix
        self.dt = dt

    def fit(self, x, y=None):
        return self

    def transform(self, X):
        '''
        PARAMS:
```

```
            X: is a DataFrame
        RETURNS: a DataFrame with additional features for the derivatives
        '''
        Xout = X.copy()
        if self.attribs == None:
            self.attribs = Xout.columns

        for attrib in self.attribs:
            vals = Xout[attrib].values
            diff = vals[1:] - vals[0:-1]
            deriv = diff / self.dt
            deriv = np.append(deriv, 0)
            attrib_name = self.prefix + attrib
            Xout[attrib_name] = pd.Series(deriv)

        return Xout
```

4 Construct Pipeline

[43]:
```
selected_names = ['left_wrist_x', 'left_wrist_y', 'left_wrist_z']
selected_inds = [baby_data_raw.columns.get_loc(name) for name in selected_names]
nselected = len(selected_names)
time = baby_data_raw['time'].values
Xsel_raw = baby_data_raw[selected_names].values
```

[44]:
```
""" TODO
Create a pipeline that:
1. Selects a subset of features
2. Fills gaps within the data by linearly interpolating the values
   in between existing data and fills the remaining gaps at the edges
   of the data with the first or last valid value
3. Compute the derivatives of the selected features. The data are
   sampled at 50 Hz, therefore, the period or elapsed time (dt) between
   the samples is .02 seconds (dt=.02)
"""
DT= .02 #global variable for dt
pipe1 = Pipeline(steps= [
                        ('selectFeature',DataFrameSelector(selected_names)),
                        ('interpolation', InterpolationImputer(method=␣
↪'linear')),
                        ('computeDerivative',␣
↪DerivativeComputer(attribs=selected_names, prefix='d_', dt=DT))
                        ]) #first piprline

""" TODO
```

```
Create a pipeline that:
1. Selects a subset of features
2. Fills gaps within the data by linearly interpolating the values
   in between existing data and fills the remaining gaps at the edges
   of the data with the first or last valid value
3. Smooth the data with a Gaussian Filter. Use a standard deviation
   of 2 and a kernel size of 7 for the filter
4. Compute the derivatives of the selected features. The data are
   sampled at 50 Hz, therefore, the period or elapsed time (dt) between
   the samples is .02 seconds (dt=.02)
"""
KERNELSIZE= 7 #global variable for kernelsize
SIGMA= 2      #global variable for gaussian sigma
pipe2 = Pipeline(steps= [
                        ('selectFeature', DataFrameSelector(selected_names)),
                        ('interpolation', InterpolationImputer(method=␣
 ↪'linear')),
                        ('gaussianFilter',␣
 ↪GaussianFilter(attribs=selected_names, kernelsize=KERNELSIZE, sig=SIGMA)),
                        ('computeDerivative',␣
 ↪DerivativeComputer(attribs=selected_names, prefix='d_', dt=DT))
                        ]) #second pipeline
```

```
KERNEL WEIGHTS [0.08868144 0.13687641 0.17759311 0.19369807 0.17759311
0.13687641
 0.08868144]
```

[45]:

```
""" TODO
Fit both Pipelines to the data and transform the data
"""
baby_data1 = pipe1.transform(baby_data_raw)# TODO
baby_data2 = pipe2.transform(baby_data_raw).iloc[KERNELSIZE//2:-(KERNELSIZE//
 ↪2)].reset_index()#to trim the padded rows and reset index

""" TODO
Display the summary statistics for the pre-processed data
from both pipelines
"""
print('statistical description of baby data 1:')
baby_data1.describe()
```

```
statistical description of baby data 1:
```

[45]:

```
       left_wrist_x  left_wrist_y  left_wrist_z  d_left_wrist_x  \
count  15000.000000  15000.000000  15000.000000    15000.000000
mean       0.244186      0.161478     -0.044664        0.000694
std        0.084979      0.093011      0.060630        0.082732
```

min	0.027525	-0.046680	-0.186060	-1.024850
25%	0.178381	0.096099	-0.082856	-0.012800
50%	0.254316	0.153330	-0.044753	0.000750
75%	0.308836	0.244393	-0.004493	0.014775
max	0.389957	0.334027	0.147053	1.469050

	d_left_wrist_y	d_left_wrist_z
count	15000.000000	15000.000000
mean	-0.000705	0.000002
std	0.058960	0.087525
min	-0.970700	-1.600800
25%	-0.011800	-0.018100
50%	-0.001000	-0.001650
75%	0.010150	0.014550
max	0.717350	0.810550

[46]:
```python
print('statistical description of baby data 2:')
baby_data2.describe()
```

statistical description of baby data 2:

[46]:

	level_0	index	left_wrist_x	left_wrist_y	left_wrist_z \
count	15000.000000	15000.000000	15000.000000	15000.000000	15000.000000
mean	7502.500000	7499.500000	0.244186	0.161478	-0.044664
std	4330.271354	4330.271095	0.084935	0.092992	0.060562
min	3.000000	0.717390	0.027684	-0.046085	-0.185986
25%	3752.750000	3749.750000	0.178182	0.096089	-0.082861
50%	7502.500000	7499.500000	0.254310	0.153358	-0.044708
75%	11252.250000	11249.250000	0.308846	0.244420	-0.004485
max	15002.000000	14998.282610	0.387130	0.331056	0.146256

	d_left_wrist_x	d_left_wrist_y	d_left_wrist_z
count	15000.000000	15000.000000	15000.000000
mean	0.000694	-0.000705	0.000002
std	0.073687	0.050054	0.077370
min	-0.910723	-0.642914	-1.177039
25%	-0.011618	-0.011303	-0.016420
50%	0.000842	-0.001050	-0.001751
75%	0.013784	0.009236	0.012902
max	1.052638	0.533001	0.725959

[47]:
```python
""" TODO
Display the first few values for the pre-processed data
from both pipelines
"""
print('the head table for baby data 1: ')
baby_data1.head()
```

```
the head table for baby data 1:
```

[47]:

```
   left_wrist_x  left_wrist_y  left_wrist_z  d_left_wrist_x  d_left_wrist_y  \
0      0.163611      0.293503     -0.092803         0.00000         -0.0029
1      0.163611      0.293445     -0.092968         0.00000         -0.0040
2      0.163611      0.293365     -0.093162         0.00000         -0.0040
3      0.163611      0.293285     -0.093356         0.00000         -0.0024
4      0.163611      0.293237     -0.093475        -0.01165         -0.0017

   d_left_wrist_z
0        -0.00825
1        -0.00970
2        -0.00970
3        -0.00595
4        -0.00915
```

[48]:

```python
print('the head table for baby data 2: ')
baby_data2.head(10)
```

```
the head table for baby data 2:
```

[48]:

```
   level_0     index  left_wrist_x  left_wrist_y  left_wrist_z  \
0        3  0.717390      0.163611      0.293454     -0.092930
1        4  1.314239      0.163611      0.293414     -0.093034
2        5  2.088681      0.163590      0.293364     -0.093168
3        6  3.000000      0.163537      0.293312     -0.093315
4        7  4.000000      0.163446      0.293265     -0.093468
5        8  5.000000      0.163310      0.293229     -0.093606
6        9  6.000000      0.163139      0.293200     -0.093736
7       10  7.000000      0.162936      0.293178     -0.093855
8       11  8.000000      0.162717      0.293156     -0.093973
9       12  9.000000      0.162500      0.293135     -0.094085

   d_left_wrist_x  d_left_wrist_y  d_left_wrist_z
0        0.000000       -0.002032       -0.005176
1       -0.001033       -0.002479       -0.006693
2       -0.002654       -0.002599       -0.007381
3       -0.004538       -0.002366       -0.007645
4       -0.006805       -0.001789       -0.006904
5       -0.008588       -0.001438       -0.006467
6       -0.010108       -0.001136       -0.005942
7       -0.010991       -0.001075       -0.005937
8       -0.010829       -0.001052       -0.005563
9       -0.010186       -0.001050       -0.005522
```

[49]:

```python
""" TODO
Display the last few values for the pre-processed data
from both pipelines
```

```
"""
print('the tail table for baby data 1: ')
baby_data1.tail()
```

```
the tail table for baby data 1:
```

[49]:

	left_wrist_x	left_wrist_y	left_wrist_z	d_left_wrist_x	d_left_wrist_y	d_left_wrist_z
14995	0.371656	0.082065	-0.092307	0.00335	0.0	0.0
14996	0.371723	0.082065	-0.092307	0.00390	0.0	0.0
14997	0.371801	0.082065	-0.092307	0.00325	0.0	0.0
14998	0.371866	0.082065	-0.092307	0.00205	0.0	0.0
14999	0.371907	0.082065	-0.092307	0.00000	0.0	0.0

[50]:
```
print('the head table for baby data 2: ')
baby_data2.tail()
```

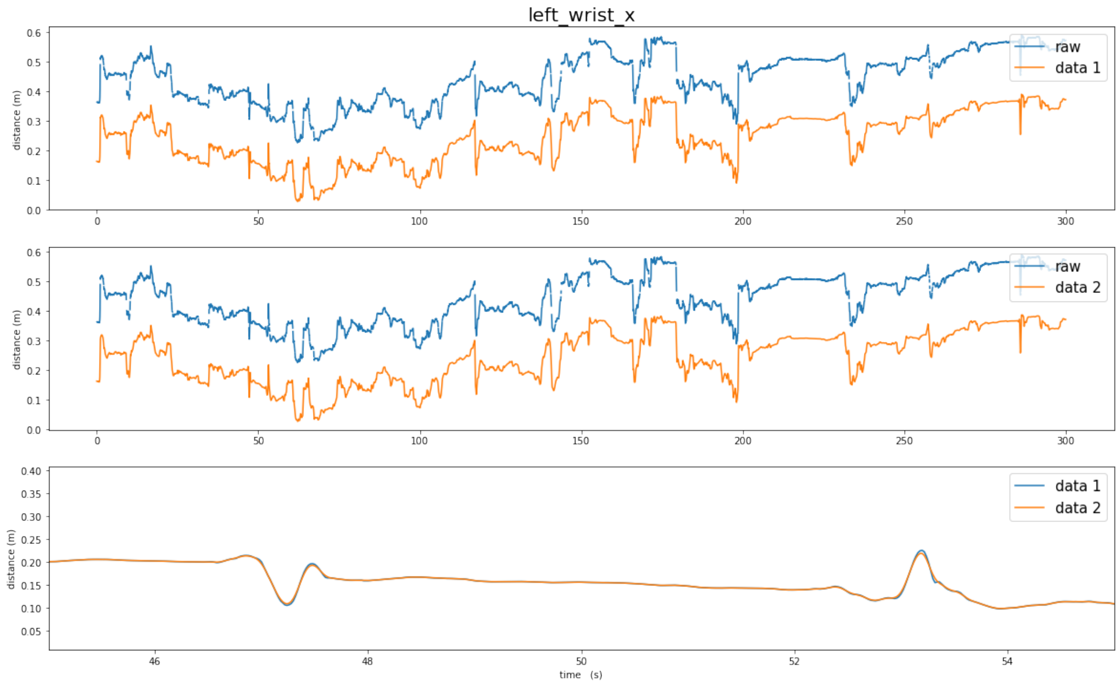
```
the head table for baby data 2:
```

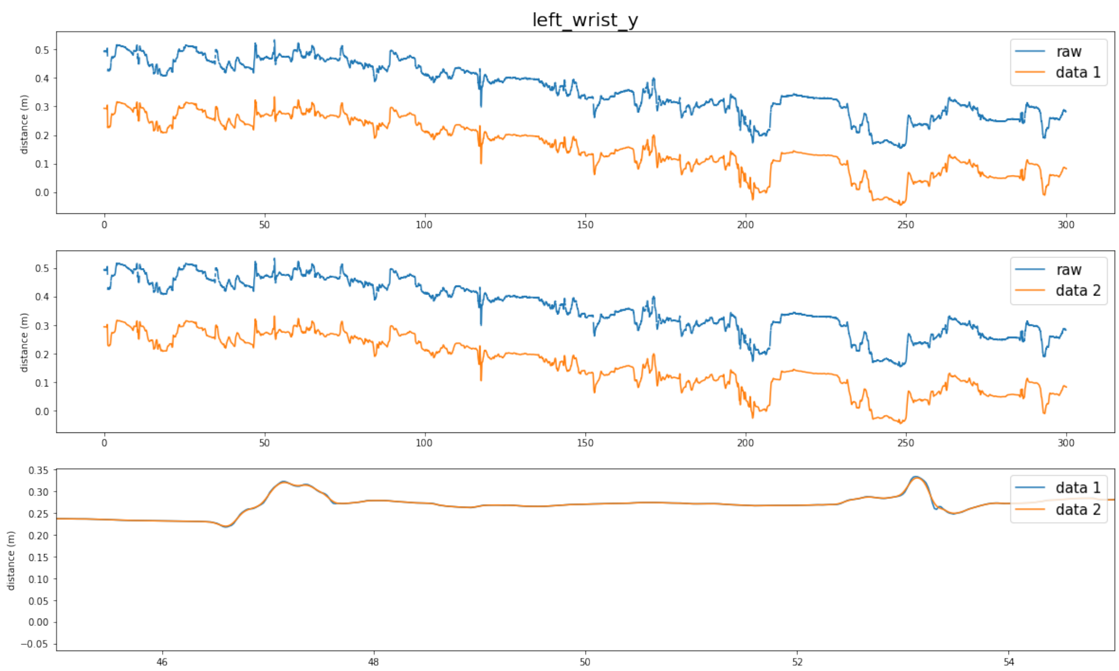
[50]:

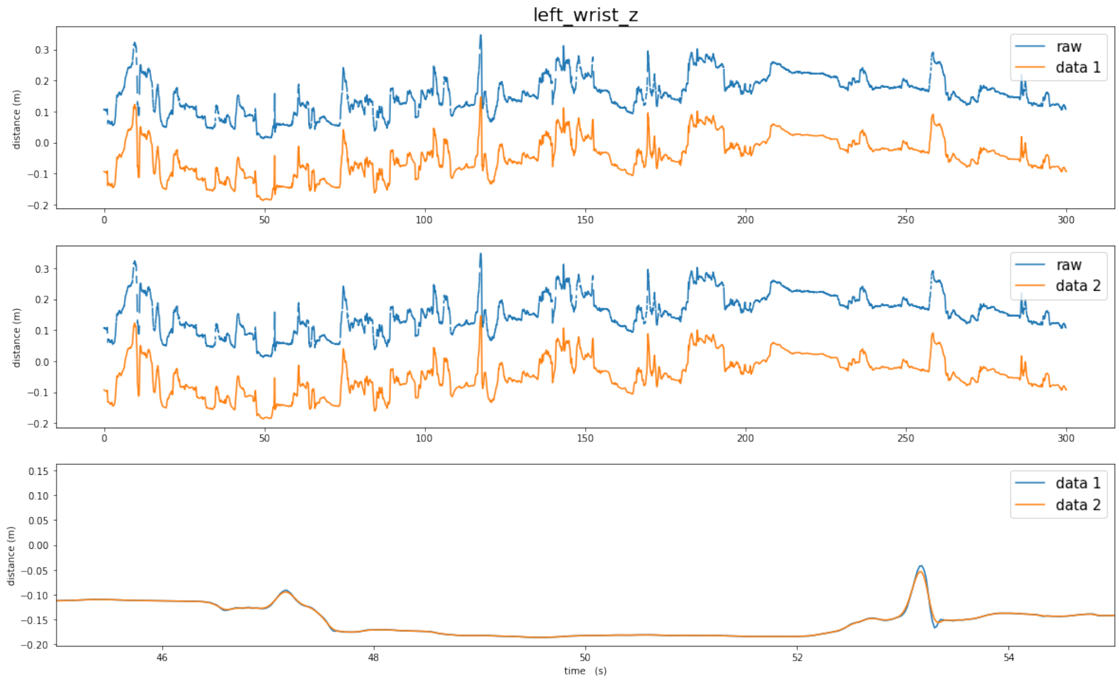
	level_0	index	left_wrist_x	left_wrist_y	left_wrist_z	d_left_wrist_x	d_left_wrist_y	d_left_wrist_z
14995	14998	14995.000000	0.371689	0.082065	-0.092307	0.002698	0.000000e+00	0.0
14996	14999	14996.000000	0.371743	0.082065	-0.092307	0.002315	0.000000e+00	0.0
14997	15000	14996.911319	0.371789	0.082065	-0.092307	0.002187	0.000000e+00	0.0
14998	15001	14997.685761	0.371833	0.082065	-0.092307	0.001805	0.000000e+00	0.0
14999	15002	14998.282610	0.371869	0.082065	-0.092307	0.001905	-6.938894e-16	0.0

[57]:
```
""" TODO
Construct plots comparing the raw data to the pre-processed data
for each selected feature from both pipelines. For each selected
feature, create a figure displaying the raw data andthe cleaned
data in the same subplot. The raw data should be shifted upwards
to clearly observe where the gaps are filled in the cleaned data.
There should be three subplots per feature figure. Each subplot
is in a separate row.
```

```python
    subplot(1) will compare the original raw data to the pipeline1
               pre-processed data
    subplot(2) will compare the original raw data to the pipeline2
               pre-processed data
    subplot(3) will compare pipeline1 to pipeline2. Set the x limit
               to 45 and 55 seconds
For all subplots, include axis labels, legends and titles.
"""


for i, name in enumerate(selected_names):
    fig, ax= plt.subplots(3,1,figsize=(20, 12))
    ax[0].plot(time, baby_data_raw[name].values+.2, label='raw')
    ax[0].plot(time, baby_data1[name].values, label='data 1')
    ax[0].set_title(name,fontsize= FONTSIZE+5)

    ax[0].set_ylabel('distance (m)')
    ax[0].legend(loc='upper right', fontsize= FONTSIZE)
    ax[1].plot(time, baby_data_raw[name].values+.2, label='raw')
    ax[1].plot(time, baby_data2[name].values, label='data 2')
    ax[1].legend(loc='upper right',fontsize= FONTSIZE)

    ax[1].set_ylabel('distance (m)')
    ax[2].plot(time, baby_data1[name].values, label='data 1')
    ax[2].plot(time, baby_data2[name].values, label='data 2')
    ax[2].legend(loc='upper right',fontsize= FONTSIZE)
    ax[2].set_xlabel('time   (s)')
    ax[2].set_ylabel('distance (m)')
    ax[2].set_xlim([45,55])
```

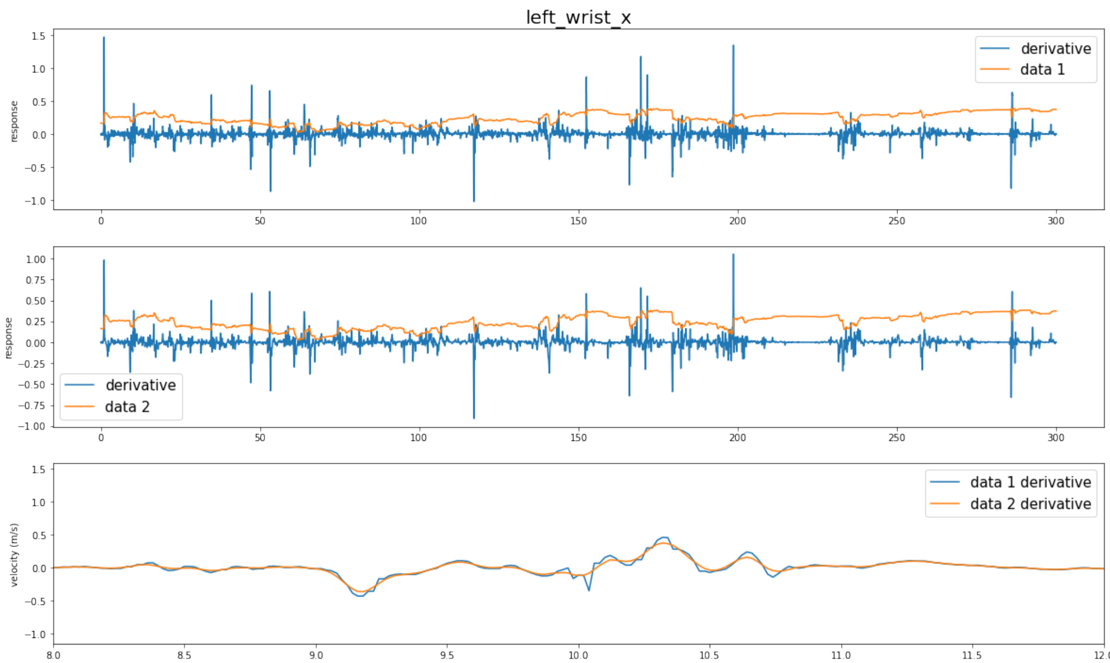

left_wrist_x
raw
data 1
raw
data 2
data 1
data 2
distance (m)
time (s)

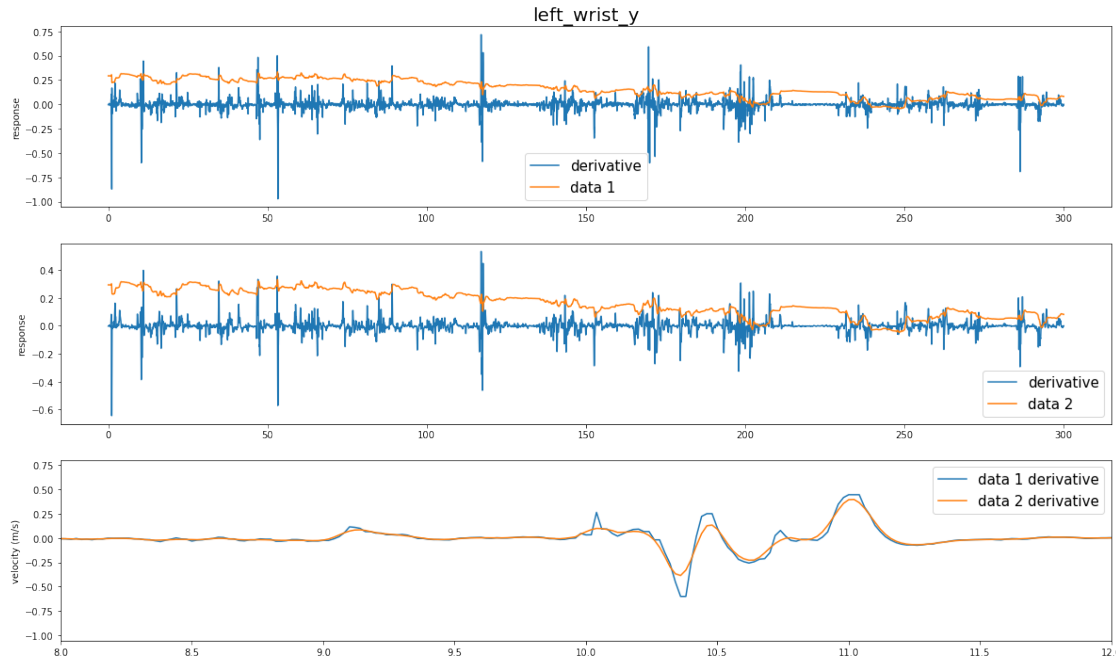

left_wrist_y
raw
data 1
raw
data 2
data 1
data 2
distance (m)
time (s)

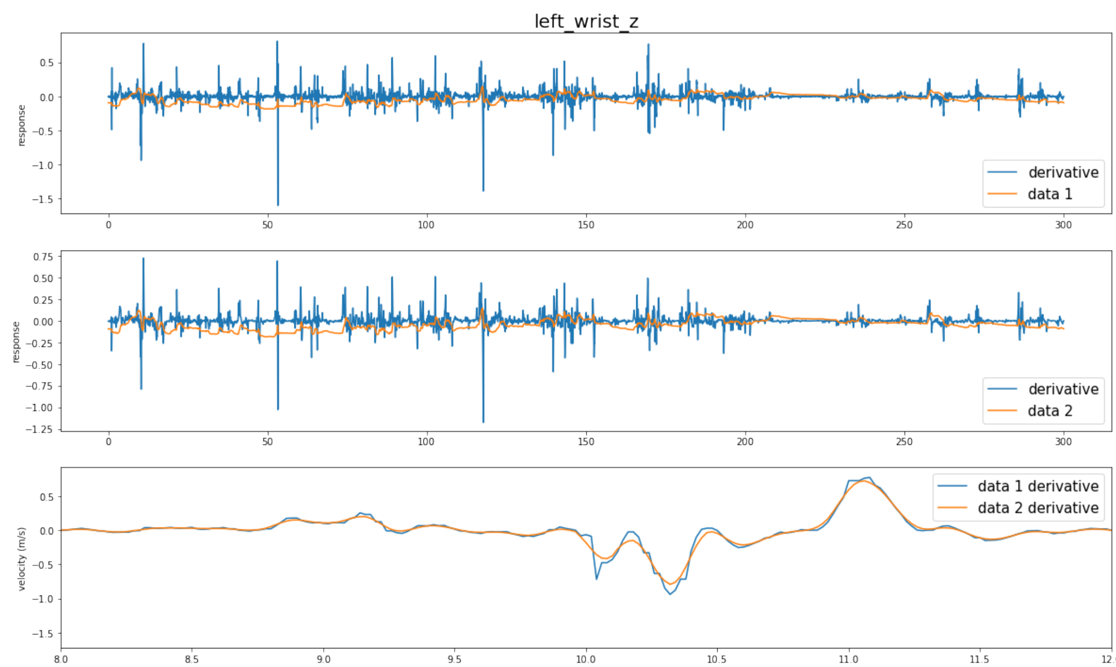


[58]:
```
""" TODO
Construct plots for each feature presenting the feature and its
derivative from both pipelines. Each figure should have
3 subplots:
    1: the pipeline1 feature data and cooresponding derivative
    2: the pipeline2 feature data and corresponding derivative
    3: pipeline1 derivative and pipeline2 derivative. Set the x limit
       to 8 and 12 seconds.
For all subplots, include axis labels, legends and titles.
"""

for i, name in enumerate(selected_names):
    fig, ax= plt.subplots(3,1,figsize=(20, 12))
    ax[0].plot(time, baby_data1['d_'+name].values, label='derivative')
    ax[0].plot(time, baby_data1[name].values, label='data 1')
    ax[0].set_title(name,fontsize= FONTSIZE+5)
    ax[0].set_ylabel('response')
    ax[0].legend(fontsize= FONTSIZE)
    ax[1].plot(time, baby_data2['d_'+name].values, label='derivative')
    ax[1].plot(time, baby_data2[name].values, label='data 2')
    ax[1].legend(fontsize= FONTSIZE)


    ax[1].set_ylabel('response')
```

```
    ax[2].plot(time[:-1], baby_data1['d_'+name].values[:-1], label='data 1␣
↪derivative')
    ax[2].plot(time[:-1], baby_data2['d_'+name].values[:-1], label='data 2␣
↪derivative')
    ax[2].legend(fontsize= FONTSIZE)

    ax[2].set_ylabel('velocity (m/s)')
    ax[2].set_xlim([8,12])
```




left_wrist_y
response
derivative
data 1
response
derivative
data 2
velocity (m/s)
data 1 derivative
data 2 derivative


left_wrist_z
response
derivative
data 1
response
derivative
data 2
velocity (m/s)
data 1 derivative
data 2 derivative