Dempster-Shafer für Emotionserkennung

von 3711191 und 8786673

Inhaltsverzeichnis

preprocessing

April 19, 2020

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import warnings; warnings.filterwarnings('ignore')
import os
```

[2]:
```
df1 = pd.read_csv('../data/E_B02_Sequenz_1.csv', delimiter=';')
df2 = pd.read_csv('../data/E_B02_Sequenz_2.csv', delimiter=';')
df3 = pd.read_csv('../data/E_B02_Sequenz_3.csv', delimiter=';')

df = df1.append(df2).append(df3)
print(len(df), len(df1), len(df2), len(df3))


df.head()
```

```
255 87 84 84
```

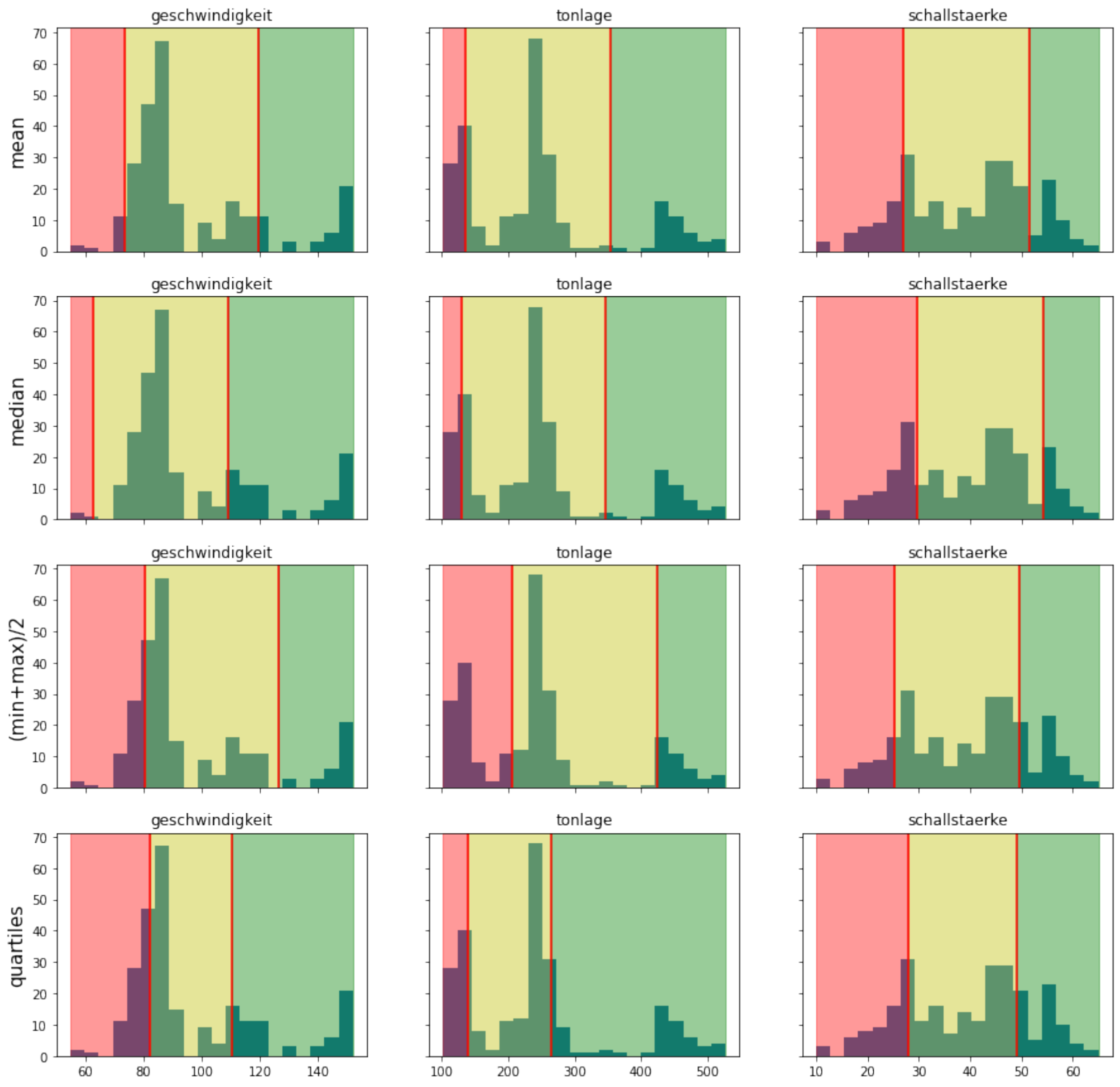
[2]:
```
   geschwindigkeit  tonlage  schallstaerke
0               87      257             33
1               84      227             33
2               82      231             34
3               79      240             37
4               76      232             41
```

0.1 Comparing Discretization Methods

What actuall *is* “low”? In order to answer that question we will compare different methods on how to define the three categories `low`, `normal` and `high` for each of the features.

Instead of taking outside sources into account we will use a data-driven method, because there exists no meta data sbout the recordings. Having a very little amount of data this naturally means that the final borders for our binning will generalize poorly to recordings from other microphones or speakers which a different demographic (gender, age, …).

[3]:
```
def mean_lower(col):
    return col.mean() - col.std()

def mean_upper(col):
    return col.mean() + col.std()

def median_lower(col):
    return col.median() - col.std()

def median_upper(col):
    return col.median() + col.std()

def half_min_max_lower(col):
    return (col.max() + col.min()) / 2 - col.std()

def half_min_max_upper(col):
    return (col.max() + col.min()) / 2 + col.std()

def quartile_lower(col):
    return col.quantile(0.25)

def quartile_upper(col):
    return col.quantile(0.75)


lower_funs = (mean_lower, median_lower, half_min_max_lower, quartile_lower)
upper_funs = (mean_upper, median_upper, half_min_max_upper, quartile_upper)
fun_names = ("mean", "median", "(min+max)/2", "quartiles")

fig, axes = plt.subplots(nrows=len(lower_funs), ncols=3, figsize=(15,15),
                        sharex='col', sharey='row')

for lower_fun, upper_fun, name, row_axes in zip(lower_funs, upper_funs,␣
 ↪fun_names, axes):

    for col_name, ax in zip(df, row_axes):
        col = df[col_name]

        col.plot.hist(bins=20, ax=ax)

        ax.axvline(lower_fun(col), color='r')
        ax.axvline(upper_fun(col), color='r')

        ax.axvspan(col.min(), lower_fun(col), alpha=0.4, color='r')
        ax.axvspan(lower_fun(col), upper_fun(col), alpha=0.4, color='y')
        ax.axvspan(upper_fun(col), col.max(), alpha=0.4, color='g')
        ax.set_title(col_name)

    row_axes[0].set_ylabel(name, size=15)
```



Using the mean or median and a 1-sigma interval will lead to barely any velocity being regarded as “low”. Using quartiles seems like a robust measure which whill include some, but not too much of the heavy left tail in `tonlage`. It will also gives us a uniform distribution in the discrete data.

```
[4]: def _stringify(x, lower, upper):
         if x <= lower: return "low"
         if x >= upper: return "high"
         return "normal"

     def discretize_using_fixed_borders(df):
         borders = {
             'geschwindigkeit': (80, 126),
             'tonlage': (180, 423),
             'schallstaerke': (25, 49)
```

```
    }

    for col in df:
        lower_border, upper_border = borders[col]

        df[col] = df[col].map(lambda x: _stringify(x, lower_border,␣
 ↪upper_border))

    return df

def discretize_using_quartiles(df):
    for col in df:
        df[col] = pd.qcut(df[col],
                          q=(0, .25, .75, 1),
                          labels=('low', 'normal', 'high'))
    return df

def discretize_window(window):
    lower_quartiles = df.quantile(0.25)
    upper_quartiles = df.quantile(0.75)

    for col in window:
        window[col] = window[col].map(lambda entry: _stringify(entry,
                                                       lower_quartiles[col],
                                                       upper_quartiles[col]))
    return window


discrete_df = discretize_using_quartiles(df.copy())
discrete_df.head()
```

[4]:
```
  geschwindigkeit tonlage schallstaerke
0          normal  normal        normal
1          normal  normal        normal
2             low  normal        normal
3             low  normal        normal
4             low  normal        normal
```

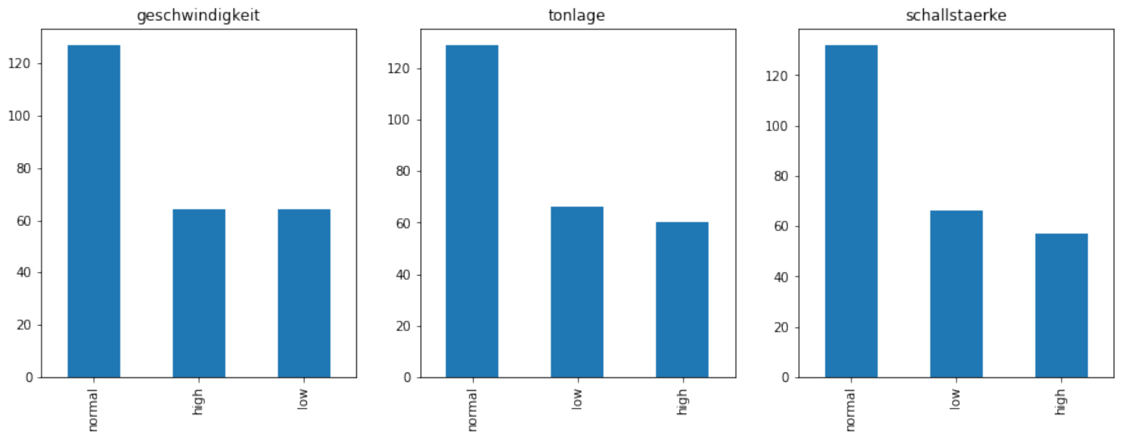
[5]:
```
# Print the global quartiles to export them into the script
dict(df.quantile(0.25)), dict(df.quantile(0.75))
```

[5]:
```
({'geschwindigkeit': 82.5, 'tonlage': 140.0, 'schallstaerke': 28.0},
 {'geschwindigkeit': 110.5, 'tonlage': 265.0, 'schallstaerke': 49.0})
```

[6]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15,5))

for col_name, ax in zip(df, axes):
        col = discrete_df[col_name]

        col.value_counts().plot(kind='bar', ax=ax)

        ax.set_title(col_name)

    row_axes[0].set_ylabel(name, size=15)
```

[6]:

```
Text(22.200000000000017, 0.5, 'quartiles')
```



0.2 Normalization

In the end we want to be able to say how much confidence we have in the statement *"the intensity was low in the last ten seconds"*. We need to be able to define a basic measure like this:

`m1(Traurigkeit) = 0.7`

In order to map a windows concrete values to a percentage value, we calculate the discrete distribution in the last ten seconds:

For each feature we calculate how many of the last then steps were `low`, `normal` or `high`.

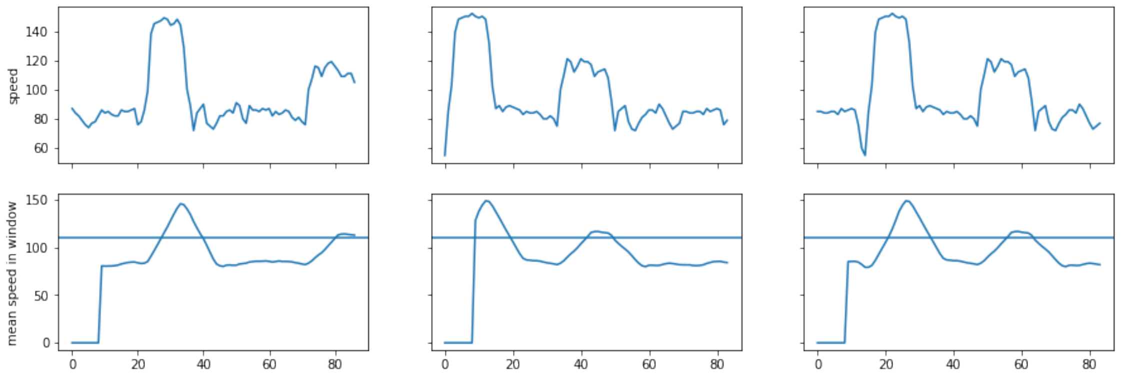
[7]:

```python
def discrete_distribution(discrete_window):
    dist = {}

    for col in discrete_window:
        window_counts = dict(discrete_window[col].value_counts())
        dist[col] = tuple(window_counts.get(level, 0) / len(discrete_window)
                          for level in ('low', 'normal', 'high'))

    return dist
```

0.2.1 Example

As an example: The last ten seconds of the first sequence look like they have a comparitively high speed.

```
[8]: dfs = (df1, df2, df3)
     vals = tuple(map(lambda df: df['geschwindigkeit'], dfs))
     windows = tuple(map(lambda df: df.rolling(window=10)['geschwindigkeit'],
                         dfs))

     fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15,5),
                              sharey='row', sharex='col')
     val_axes, window_axes = axes

     for val, ax in zip(vals, val_axes):
         val.plot(ax=ax)
         ax.set_ylabel('speed')

     for window, ax in zip(windows, window_axes):
         window.mean().fillna(0).plot(ax=ax)
         ax.axhline(df['geschwindigkeit'].quantile(0.75))
         ax.set_ylabel('mean speed in window')
```



```
[9]: last_window = df1[-10:]
     window = last_window

     discrete_window = discretize_window(window.copy())
     discrete_distribution(discrete_window)
```

```
[9]: {'geschwindigkeit': (0.0, 0.3, 0.7),
      'tonlage': (0.0, 0.0, 1.0),
      'schallstaerke': (0.0, 0.8, 0.2)}
```

This gives us three basic measures:

```
# high , low, normal
mSpeed(A, Ü, W, F) = 0.7, mSpeed(F, E) = 0, mSpeed(Omega) = 0.3
mPitch(A, Ü, W, F) = 1.0, mPitch(E, T) = 0, mPitch(Omega) = 0
mIntensity(Ü, W, F) = 0.2, mIntensity(T) = 0, mIntensity(Omega) = 0.8
```

0.3 Intensity: low deviations

We know about one more feature that has to be determined differently from the others:

The fact that a high deviation in the intensity can indicate sadness.

How do we map the intensity in a window to a confidence $c \in \{0, 1\}$ that we can use in a basic measure? Using Min-Max-Scaling we can determine to confidence for the current window w':

$$c_{w'} = \frac{\sigma_{w'} - \min_{w \subset W} \sigma_w}{\max_{w \subset W} \sigma_w - \min_{w \subset W} \sigma_w}$$

where W is the set of all windows.

[10]:
```
win_stds = df.rolling(window=10)['schallstaerke'].std()
global_min = win_stds.min()
global_max = win_stds.max()
global_min, global_max
```

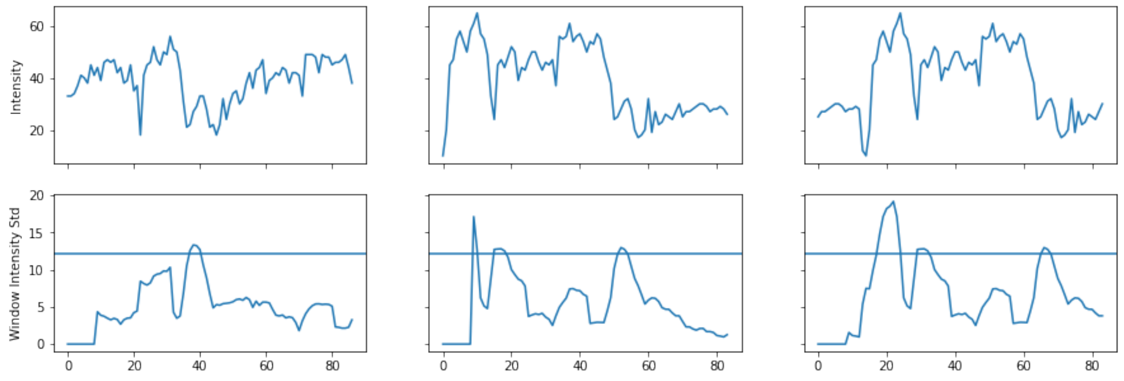
[10]:
```
(0.9660917830794542, 19.203298327804692)
```

[11]:
```
def normalized_intensity_std(window):
    def _min_max_normalize(x, _min, _max):
        numerator = win_std - _min
        denominator = _max - _min
        return numerator / denominator

    win_std = window['schallstaerke'].std()

    global_max = 19.3
    global_min = 0.95

    return _min_max_normalize(win_std, global_min, global_max)
```

0.3.1 Example

[12]:
```
dfs = (df1, df2, df3)
vals = tuple(map(lambda df: df['schallstaerke'], dfs))
windows = tuple(map(lambda df: df.rolling(window=10)['schallstaerke'],
                    dfs))

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15,5),
                         sharey='row', sharex='col')
val_axes, window_axes = axes
```

```
val_axes[0].set_ylabel('Intensity')
for val, ax in zip(vals, val_axes):
    val.plot(ax=ax)

for window, ax in zip(windows, window_axes):
    window.std().fillna(0).plot(ax=ax)
    ax.axhline(df['schallstaerke'].std())
    ax.set_ylabel('Window Intensity Std')
```



At around 12 to 22 seconds in sequence 3:

```
[13]: window = df3[12:22]
      window['schallstaerke'].std()
```

```
[13]: 18.56789343643125
```

```
[14]: normalized_intensity_std(window)
```

```
[14]: 0.960103184546662
```

Compared to the end of sequence 1:

```
[15]: window = df1[-10:]
      normalized_intensity_std(window)
```

```
[15]: 0.1262117887580874
```

Which gives us a forth basic measure:

```
mIntensityStd(T) = 0.126, mIntensityStd(Omega) = (1 - 0.126)
```

dempster_shafer.basic_measure

index
/Users/hegele/Nextcloud/Study/s6/ki/prog/dempster_shafer/basic_measure.py

```
General functions for Dempster-Shafer theory

Classes:
    BasicMeasure: also called "mass" (dt. Basismaß)
            can be used to determine a belief, plausibility
            can be accumulated with other basic measures

Functions:
    accumulate: create a new basic measure from two basic measures
        using Dempster's rule of combination
```

Classes

builtins.object
    BasicMeasure

class **BasicMeasure**(builtins.object)

```
BasicMeasure(entry_domain)

A class that can be used to define a basic measure / mass (dt. Basismaß)
    e.g. m1(Alive) = 0.6, m1(Dead) = 0.3, m1(Omega) = 0.1

Attributes:
    entry_domain (frozenset): the domain of possible entries,
        where each entry is a string,
        e.g. {'Alive', 'Dead'}
    measures (dict): A dictionary mapping entry sets to measures / mass,
        e.g. {{'Alive', 'Dead'}: 0.1, ...}
```

Methods defined here:

__init__(self, entry_domain)

```
Constructor of BasicMeasure.
Initializes the measures for the power set of all possible values.

Parameters:
    entry_domain (iterable): all possible entries
```

add_entry(self, entry, value: float)

```
Add a single measure for an entry.
The Omega entry is maintained automatically
and should not be set.

Parameters:
    entry: single string or iterable of unique strings
    value (float): mass to assign to the entry set

Example:
    m1 = BasicMeasure({'Alive', 'Dead'})
    m1.add_entry('Alive', 0.6)
    m1.add_entry({'Dead'}, 0.3)

Negative Example:
    m1 = BasicMeasure({'Alive', 'Dead'})
    m1.add_entry({'Alive', 'Dead'}, 0.1)
```

args_to_set(self, *args)
```
Convenience function to enable getters to accept multiple
string arguments and to interpret "Omega" as the entry set
containing the entry domain.
    e.g. a = m1.get_measure('Alive', 'Dead')
         b = m1.get_measure(Omega)
         return a == b
```

get_belief(self, *args)
```
Return the belief for a single entry set
```

get_doubt(self, *args)
```
Return the doubt for a single entry set
```

get_measure(self, *args)
```
Return the mass for a single entry set
```

get_measures(self)
```
Return a dictionary mapping all entries to their measures
```

get_plausibility(self, *args)
```
Return the plausibility for a single entry set
```

Data descriptors defined here:

__dict__
```
dictionary for instance variables (if defined)
```

__weakref__
```
list of weak references to the object (if defined)
```

Functions

accumulate(m1: dempster_shafer.basic_measure.BasicMeasure, m2: dempster_shafer.basic_measure.BasicMeasure)
```
Accumulate to measures using Dempster's rule of combination.
Applies conflict resolution. For example usage best regard the tests.

Parameters:
    m1 (BasicMeasure): a basic measure of a certain entry domain
    m2 (BasicMeasure): a basic measure of the same domain

Returns:
    m12 (BasicMeasure): a basic measure containing the accumulated values
```

powerset(iterable)
```
Returns the power set of values for any iterable.
e.g. powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
```

process_set(in_set)
```
Check an incoming set, list or tuple of values,
and try to convert it to a frozenset.
```

emotion

index
/Users/hegele/Nextcloud/Study/s6/ki/prog/emotion.py

```
Emotion module

Determines plausibility and belief for all emotions in each 10 second
interval within the given CSV.

Expects a strings indicating the path to the CSV with the acoustic data.
Expects the CSV to be of format "geschwindigkeit;tonlage;schallstaerke".

Writes the output into a new CSV matching the file name in the 'results' folder.

Example Invocation:
        python emotion.py data/E_B02_Sequenz_1.csv
```

Modules

os　　pandas　　sys

Functions

discretize_cont_df(cont_df)

```
Discretizes the values into bins of low, normal and high

Parameters:
    cont_df (pandas.DataFrame): set of continous values for
        'geschwindigkeit', 'tonlage' and 'schallstaerke'
Returns:
    disc_df (pandas.DataFrame): set of discrete values
        each one of 'low', 'normal' or 'high'
```

distribution_of_bins(discrete_window)

```
Determines the window's ratios for 'low', 'normal' and 'high'

Parameters:
    discrete_window (pandas.DataFrame): a time frame containing
        multiple rows of binned values
Returns:
    dist (dict): maps the each col to a 3-tuple indicating
        the ratios for 'low', 'normal' and 'high',
        e.g. 'tonlage': (0.3, 0.5, 0.2)
```

main(csv_file)

```
Main procedure: discretize values, and for each window
determine the distribution of bins
as well the degree of standard deviation in the voice intensity.

Then, construct basic measures out of the determined values,
and accumulate them into single basic measure
to determine the belief and plausibility for each emotion in the window.

Finally, write the output into a 'csv' file in the 'results' folder.

Parameters:
    csv_file (str): path to the CSV file containing continous values
        for each of the voice parameters
        'geschwindigkeit', 'tonlage' and 'schallstaerke'

Returns:
    None
```

normalized_intensity_std(window)

```
Normalizes the intensity's standard deviation to [0, 1]

Parameters:
    window (pandas.DataFrame): a time frame containing
        multiple rows of continuous values
Returns:
    win_std (float): a continous number in between 0 and 1
        indicating to which degree the intensity was high
        in this window
```

evaluation

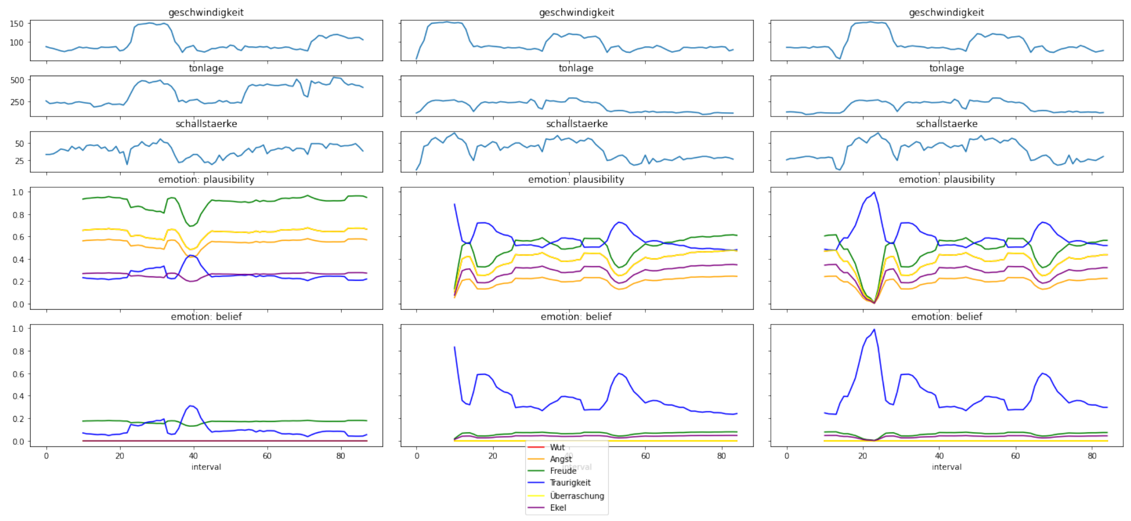
April 19, 2020

0.1 Evaluation

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import warnings; warnings.filterwarnings('ignore')
import os
```

[3]:
```
res1, res2, res3 = (pd.read_csv(os.path.join('results',
                                             f'E_B02_Sequenz_{seq}.csv'),
                                index_col=['filename', 'window', 'metric',␣
 ↪'emotion'])
                    for seq in (1, 2, 3))

df1, df2, df3 = (pd.read_csv(res.index[0][0], delimiter=';')
            for res in (res1, res2, res3))

res1, res2, res3 = (res.reset_index(level=0).drop('filename', axis=1)
                    for res in (res1, res2, res3))

dfs = (df1, df2, df3)
res_dfs = (res1, res2, res3)
```

[9]:
```
from matplotlib.gridspec import GridSpec

fig = plt.figure(figsize=(20,10))
gs = GridSpec(5, 3, figure=fig,
              height_ratios=[1, 1, 1, 3, 3],
              wspace=0.05)

axes = [[None for x in range(5)] for y in range(len(dfs))]
for seq, (res, df) in enumerate(zip(res_dfs, dfs)):

    for i, col in enumerate(df):
        ax = fig.add_subplot(gs[i, seq],
                             sharex=axes[seq][i-1],
                             sharey=axes[seq-1][i])
        axes[seq][i] = ax
        df[col].plot(title=col, ax=ax)
    colors = {'Wut':'red',
              'Angst':'orange',
              'Freude':'green',
              'Traurigkeit':'blue',
              'Überraschung': 'yellow',
              'Ekel': 'purple'
             }

    for j, metric in enumerate(('plausibility', 'belief')):
        j = i + j + 1
        ax = fig.add_subplot(gs[j, seq],
                              sharex=axes[seq][j-1],
                              sharey=axes[seq-1][j])
        axes[seq][j] = ax
        for emotion in colors:
            emotions = res.loc(axis=0)[:,metric, emotion]
            emotions['window_shifted'] = emotions.index.levels[0] + 10
            emotions.plot(ax=ax,
                            x='window_shifted',
                            y='value',
                            label=emotion,
                            color=colors[emotion],
                            title=f'emotion: {metric}',
                            legend = False
                         )
            ax.set_xlabel("interval")
            handles, labels = ax.get_legend_handles_labels()

fig.legend(handles, labels, loc='lower center')
fig.savefig('doc/evaluation.png')
```



Allgemein:

Wut wird von Überraschung überlagert – deren Basismaß-Definitionen sind äquivalent.

Da Traurigkeit über eine zusätzliche Evidenz (Abweichung in der Schallstärke) unterstützt wird, in der die Traurigkeit das einzige Subjekt ist, hat es global einen höheren Belief als die anderen Emotionen. Es ist wahrscheinlicher Teilmengen $X \subset E$ für $E =$ Traurigkeit mit $m(X) > 0$ zu finden, als für andere E. Deswegen sollte hier die Plausibilität als Metrik bevorzugt werden.

Sequenz 1:

Bis auf den Einbruch in der Tonlage und Schallstärke in der Mitte der Sequenz scheint Freude hier die dominante Emotion zu sein. Signifikant ist hier die Höhe des Beliefs für die Freude – trotz der beschriebenen Tendenz.

Sequenz 2 und 3:

Sind ein und dieselbe Tonaufnahme um etwa 15 Sekunden verschoben. Dementsprechend herrscht hier Ähnlichkeit in der Klassifikation.

Die Traurigkeit erscheint als plausibel nicht nur zu Anfang, wenn die Abweichung in der Schallstärke hoch ist, sondern auch später bei tiefer Tonlage und Schallstärke. Da die Schallstärke zu Ende der zweiten Sequenz wieder steigt, steigt hier auch die Plausibilität der anderen Emotionen.

```
[ ]:
```

Emotion Classification using Dempster-Shafer Theory

Classify the emotions in a time series of voice acoustics.

Installation

Requirements:

- Python 3.7
- `pandas` and its dependencies

```
pip install -r requirements.txt
```

Docker

If this does not work for you consider building using the Docker image. This will build an run on all CSV files in the `data/` directory.

```
docker build -t dempster-shafer .
docker run -it dempster-shafer
```

Or enter it with an interactive bash:

```
docker run -it --rm dempster-shafer /bin/bash
```

Test

```
make test
```

Run

The Run will produce results as CSV files in `results/*.csv`.

You can run on single CSV files:

```
python emotion.py data/E_B02_Sequenz_1.csv
```

Or run on all CSV files in the `data/` directory.

```
make
```

For an evaluation of the results see `evaluation.ipynb`.

Documentation

See the `doc/` directory for code documentation as well as explanations for our procedure. It is all summarized in the `doc/documentation.pdf`.