ClassifierExperiment_new

March 13, 2016

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import numpy as np
from libs import experiment
from pandas import DataFrame
from bpati import signals
from libs.datafiles import *
from libs.classifiers import *
from libs import common, features
from libs.arff import *
from datetime import datetime

datafiles = DataFiles(2)
generics, smags, sangles = datafiles.get_smag()
fn_sag_mode_events = common.get_precise_cycle_fault_signature()
```

```
Cfgmap loadad! (/mnt/qnap/BPATI_2014/2014.cmap)
Lazy Loading new-style data with 252454 minutes
No event data read 'eventdata'
PMUData SHA: b053a1359ab9e9ce1e6c43d6ca6cd9efa519d11e
```

In [2]:
```
def experimental_classifier_plot(df, classifier, xline=True, filename='', xlim=None):
    ce, results, start_i, v_ss,_,_ = classifier.scan_df(df)
    experimental_plot(df, results, classifier.signal, classifier.smags, xline, xlim, filename=fi

def experimental_plot(df, results, signal, smags, xline, xlim, filename=''):
    if results == None or all([isnan(r) for r in results]):
        print 'No signals'
        return

    result_diff_i = []
    for idx, _class in enumerate(results):
        if len(result_diff_i) == 0 or idx == len(results) - 1:
            result_diff_i.append(idx)
            continue
        if results[idx-1] == _class: continue
        result_diff_i.append(idx)

    phase_a, phase_b, phase_c = smags

    dt = df['DateTime'].iloc[0]
    pyplot.figure(figsize=(12, 4), dpi=100)
```

```
        #pyplot.subplot(211)
        #pyplot.title('%s\nStarting At: %s\nOrange: SLG, Yellow: LtL, Blue: 3P, Purple: N/A' % (sig
        pyplot.xlabel('Cycles')
        pyplot.ylabel('Voltage (kV)\nA,B,C Phases in Red,Green,Blue')
        pyplot.plot(df.index, df[phase_a], 'g', label = 'A')
        pyplot.plot(df.index, df[phase_b], 'r--', label = 'B')
        pyplot.plot(df.index, df[phase_c], 'k:', label = 'C')
        pyplot.legend(loc='best', title='Phases')
        if xlim is not None: pyplot.xlim((xlim[0], xlim[1]))

        if xline:
            for idx in xrng: pyplot.axvline(idx, c='r', linestyle='--')

        base = df.index[0]
        for idx, start_span in enumerate(result_diff_i[:-1]):
            #if start_span > 38: continue
            if (results[start_span] == 0): color = {'color':'orange', 'alpha':.5}
            elif (results[start_span] == 1): color = {'color':'yellow', 'alpha':.5}
            elif (results[start_span] == 2): color = {'color':'blue', 'alpha':.5}
            elif (np.isnan(results[start_span])): color = {'color':'purple', 'alpha':.5}
            else: continue
            pyplot.axvspan(base + start_span, base + result_diff_i[idx+1], **color)

        if filename != '': pyplot.savefig(filename, format='pdf', bbox_inches='tight')
        pyplot.show()
        pyplot.close()

    class BaseClassifyingExperimenter(object):
        class __Classification:
            def __init__(self):
                self.classification = []
                self.steady_states = []
                self.first_steady_state = {}
                self.updated_steady_state = []

            def first_update(self, experimenter, start_idx, start_i):
                self.first_steady_state[start_idx] = [experimenter.signaldata[mag] for mag in experi
                self.classification += [-1]*(start_i - start_idx)
                self.updated_steady_state += [False]*(start_i - start_idx)
                self.steady_states += [None]*(start_i - start_idx)

            def add(self, _class, ss, isupdated):
                self.classification.append(_class)
                self.steady_states.append(ss)
                self.updated_steady_state.append(isupdated)

            def __len__(self):
                return len(self.classification)

        def __init__(self, signal, smags, sangles, classifier_pickle):
            self.signal = signal
            self.smags = smags[signal]
            self.is2014data = common.check_2014_data(signal)
```

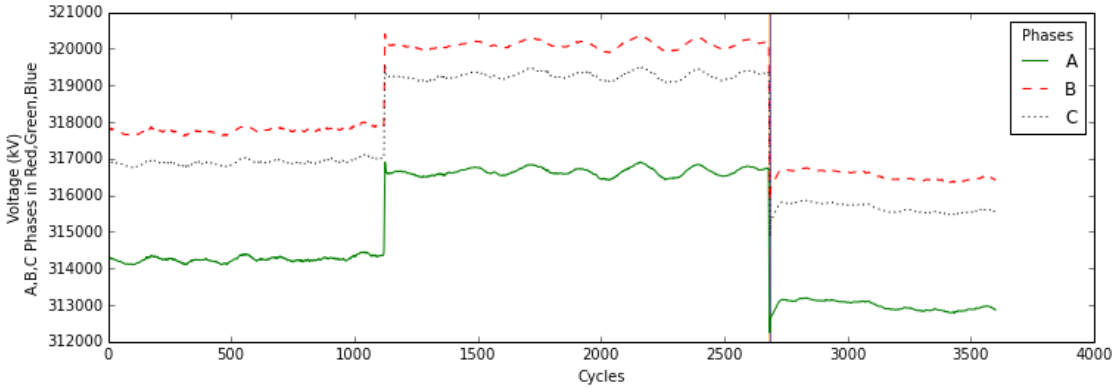
```
        if not self.is2014data: self.sangles = sangles[signal.replace('Mag', 'Ang')]
        else: self.sangles = sangles[signal.replace('MAG', 'ANG')]
        self.nominal = signals.nominalvoltage(signal)

        self.signaldata = None
        self.datapoint = None
        self.first_signal_data = None

        self.classifier = EventClassifier.LoadPickle(classifier_pickle)

        self.last_datapoint_scan_datetime = None
        pass

    # the function is to convert 2014 -> 2013 format
    def convert_2014_data_(self, scan_df):
        df = {col: common.convert_2014_smag(scan_df[col].values) for col in self.smags}
        df.update({col: common.convert_2014_angs(scan_df[col].values) for col in self.sangles})
        df.update({'DateTime': scan_df['DateTime'].values})
        return DataFrame(data=df, index=scan_df.index.values)

    #the function return the tuple: success, normal_voltages, starting point and ending point
    def steady_state_(self, arr, extent = 10):
        assert self.signaldata is not None or len(arr) > extent, 'Scan window is too narrow'
        starti = 0
        for starti in range(0, len(arr) - extent):
            window = arr[starti:starti + extent]
            #print 'check nan or window <=25', any(isnan(window)) or any(window <= 25)
            #print 'come heh', any(isnan(window)), any(window <= 25)
            if any(isnan(window)) or any(window <= 25): continue
            if self.is_steady_state_(window, True): return True, starti, starti + extent
        return False, starti, len(arr)

    def is_steady_state_(self, voltages, edge_test):
        sagthresh=.9
        edgethresh=.01
        outofservice_threshold = 0.8

        w_mean, w_min = mean(voltages), min(voltages)
        edgemin = (1.0 - edgethresh) * w_mean
        edgemax = (1.0 + edgethresh) * w_mean
        return w_min > sagthresh*w_mean and w_mean >= outofservice_threshold * self.nominal and
                (not edge_test or (voltages[0] > edgemin and voltages[-1] > edgemin))

    def update_first_window_(self, df, steady_states, base_idx, starti, extent_data_point = 1):
        self.signaldata = {key: array(df[key].loc[v_ss[1]+base_idx:v_ss[2]+base_idx-1])
                               for key, v_ss in zip(self.smags, steady_states)}
        self.signaldata.update({key: array(df[key].loc[v_ss[1]+base_idx:v_ss[2]+base_idx-1])
                                for key, v_ss in zip(self.sangles, steady_states)})
        self.datapoint = {key: array(df[key].loc[starti-extent_data_point:starti-1])
                              for key in self.signaldata}

    def update_data_window(self, df, idx):
        for key in self.signaldata:
            self.signaldata[key] = append(delete(self.signaldata[key], 0), df[key][idx])
```

```
def reused_first_window_(self, dt):
    limitted_n_cycle_from_lastpoint = 2
    in_out_date = self.last_datapoint_scan_datetime is not None and \
            0 <= (dt - self.last_datapoint_scan_datetime).total_seconds() < 0.02 * limitted
    return in_out_date

def compute_starting_point_(self, df, base_idx):
    if self.reused_first_window_(df['DateTime'][base_idx]): return base_idx
    if any([col not in df for col in self.smags+self.sangles]): return -1

    # no scan before
    v_ss_info = [self.steady_state_(df[s].loc[base_idx:].values) for s in self.smags]
    starti = base_idx + max([v_ss[2] for v_ss in v_ss_info])
    if all([v_ss[0] for v_ss in v_ss_info]):
        self.update_first_window_(df, v_ss_info, base_idx, starti)
    return starti

def get_exporting_event_(self, df, idx, verbose=False):
    for s in self.signaldata:
        self.datapoint[s] = append(delete(self.datapoint[s], 0), df[s][idx])
    if verbose:
        print '\tDatapoint'
        for k in self.datapoint: print '\t\t', k, self.datapoint[k]
    return ExportingEvent(self.signal, self.smags, self.sangles, self.datapoint, self.signal

# the function is a combination of logics to determine that the moment of time is appropria
# in this experiment, I would let the movement forward if the voltage is not less than 90%
# the sag threshod is equal to the one used in steady_state_
def is_safe_to_slide_(self, predict, v_a, v_b, v_c, verbose):
    ss_flags = [self.is_steady_state_(append(delete(self.signaldata[s], 0), v), False)
                    for v, s in zip([v_a, v_b, v_c], self.smags)]
    if verbose:
        print '\tSteady state flags:'
        for s, f in zip(self.smags, ss_flags): print '\t\t', s, f
    # if the last outcome is noFault, or is steady_state check is ok
    return predict == 3.0 and all(ss_flags)

def is_valid_ss_window_(self, dt):
    return True

# the function is an iteration if scan that should start at start_idx and break
#if ss window is out date time or until reach to end
def __scan_df_(self, df, start_idx, classification, verbose):
    start_i = self.compute_starting_point_(df, start_idx)

    if verbose: print 'Staring point: ', start_i
    if self.signaldata is None : return -1

    classification.first_update(self, start_idx, start_i)

    idx = start_i
    for idx in df.index[df.index >= start_i]:
        if not self.is_valid_ss_window_(df['DateTime'][idx]): break
```

```
            ss_a, ss_b, ss_c = [mean(self.signaldata[s]) for s in self.smags]
            if verbose:
                print 'At cycle', idx
                print '\tSteady state phase A', ss_a
                print '\tSteady state phase B', ss_b
                print '\tSteady state phase C', ss_c

            ee = self.get_exporting_event_(df, idx, verbose)
            self.last_datapoint_scan_datetime = df['DateTime'][idx]

            if ee.error:
                if verbose: print '\tSkip since the signal is unvailable'
                classification.add(nan, (ss_a, ss_b, ss_c), False)
                #print idx, len(results)
                continue

            # all success for ss computation
            x = {field.split()[0]: lambda_fn(ee)
                      for field, lambda_fn
                      in defaultfields if field.split()[0] in self.classifier.features()}
            if verbose:
                print '\tComputed features:'
                for k in x: print '\t\t', k, x[k]
            outcome_predict = self.classifier.predict_instance(x)
            update_ss = False
            # check to move
            if self.is_safe_to_slide_(outcome_predict, # recently predict outcome, test to bypa
                                      df[self.smags[0]][idx], # v_a
                                      df[self.smags[1]][idx], # v_b
                                      df[self.smags[2]][idx],
                                      verbose = verbose): # v_c
                if verbose: print '\tUpdating steady state', idx
                self.update_data_window(df, idx)
                update_ss = True
            else:
                if verbose: print '\tSkipping updating steady state', idx

            classification.add(outcome_predict, (ss_a, ss_b, ss_c), update_ss)
            if verbose: print '\tPrediction outcome:', outcome_predict
        return idx

    # return a tuple as
    #    self: the object experiment which contains up to date signal data
    #    results: the predict outcome
    #    start_i: the start index of dataframe in which corespondents to the first predict outc
    def scan_df(self, df, verbose = False):
        if self.is2014data:
            df = self.convert_2014_data_(df)
        df = df[self.smags + self.sangles + ['DateTime']]
        start_idx = df.index.values[0]
        result = self.__Classification()
        while(len(result.classification) < df.shape[0]):
            idx = self.__scan_df_(df, start_idx, result, verbose)
            if idx == -1: return None, None, None, None
```

```
                start_idx = idx
            return  result.classification, result.steady_states, result.first_steady_state, result.

    class ClassifyingExperimenter(BaseClassifyingExperimenter):
        pass

    class ConfigurableClassifyingExperimenter(BaseClassifyingExperimenter):
        def __init__(self, signal, smags, sangles, classifier_pickle):
            super(ConfigurableClassifyingExperimenter, self).__init__(signal, smags, sangles, classi
            self.last_ss_scan_datetime = None

        def reused_first_window_(self, dt):
            return self.is_valid_ss_window_(dt)

        def update_data_window(self, df, idx):
            super(ConfigurableClassifyingExperimenter, self).update_data_window(df, idx)
            self.last_ss_scan_datetime = df['DateTime'][idx]

        def update_first_window_(self, df, steady_states, base_idx, starti):
            super(ConfigurableClassifyingExperimenter, self).update_first_window_(df, steady_states
            self.last_ss_scan_datetime = df['DateTime'][starti-1]

        # test the scan should be within datetime of steady state
        def is_valid_ss_window_(self, dt):
            limitted_n_cycle_from_ss = 3
            in_out_date = self.last_ss_scan_datetime is not None and \
                    0 <= (dt - self.last_ss_scan_datetime).total_seconds() < 0.02 * limitted_n_cycle
            return in_out_date

    class ContinuousClassifyingExperimenter(BaseClassifyingExperimenter):
        #the function return the tuple: success, normal_voltages, starting point and ending point
        def steady_state_(self, arr):
            return super(ContinuousClassifyingExperimenter, self).steady_state_(arr, 30)

        def is_safe_to_slide_(self, predict, v_a, v_b, v_c, verbose):
            return True
```

In [3]:
```
signal = '' # signal name
classifier = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pick
df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[:]

results, _, _, _ = classifier.scan_df(df, False)
experimental_plot(df, results, classifier.signal, classifier.smags, False, None, '')
print len(results)
```



3600

In [4]:
```
# test function
signal = '' # signal name
classifier = ClassifyingExperimenter(signal, smags, sangles, 'txt/EventClassifier_20160101.pickl
df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)
results, v_ss, first_ss, updated_ss = classifier.scan_df(df, False)
assert all([x == -1 for x in results[0:10]]), 'first ten cycles are equal to 10'
assert all([x != -1 for x in results[10:]]), 'The ss window should not reset'
assert len(results) == 3600 and len(updated_ss) == 3600, 'Failed to reach the end of window'
assert len(classifier.signaldata[smags[signal][0]]) == 10, 'Over flow signal data'
assert len(classifier.datapoint[smags[signal][0]]) == 1, 'Over flow signal data'

# Test of resuing ss window
df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 31), 0, 60)
results, v_ss, _, _ = classifier.scan_df(df, False)
assert results[0] != -1, 'Failed to reuse the window steady state'

# Test of resuing ss window
df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[1000:2000]
results, v_ss, _, _ = classifier.scan_df(df, False)
assert len(results) == 1000, 'Failed to reach the end of window'
assert results[0] == -1, 'Begining of scan'

df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[2002:3002]
results, v_ss, _, _ = classifier.scan_df(df, False)
assert results[0] == -1, 'Failed since reusing the window steady state'
assert len(results) == 1000, 'Failed to reach the end of window'

# Test of not resuing ss window
df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[1000:2000]
results, v_ss, _, _ = classifier.scan_df(df, False)
assert len(results) == 1000, 'Failed to reach the end of window'

df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[2011:3011]
results, v_ss, _, _ = classifier.scan_df(df, False)
assert results[0] == -1, 'Failed to reuse the window steady state'
assert len(results) == 1000, 'Failed to reach the end of window'
```

```
        print 'Pass'
```

```
Pass
```

In [5]:
```
        # test function
        signal = '' # signal name
        classifier = ClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pickle')
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)
        results, v_ss, first_ss, updated = classifier.scan_df(df, False)
        assert len(results) == 3600, 'Failed to reach the end of window'

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 31), 0, 60)
        results, v_ss, first_ss, updated = classifier.scan_df(df, False)
        assert results[0] != -1, 'Failed to reuse the window steady state'

        print 'Pass'
```

```
Pass
```

In [6]:
```
        #test function
        signal = '' # signal name
        classifier = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pick
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)
        results, v_ss, _, _ = classifier.scan_df(df, False)
        assert all([x == -1 for x in results[2684:2684+10]]), 'start of reset scan should equal to -1'
        assert all([x == -1 for x in results[0:10]]), 'start of scan should equal to -1'
        assert all([x != -1 for x in results[10:2684]]), 'The other should not be -1'
        assert all([x != -1 for x in results[2684+10:]]), 'The other should not be -1'

        #test function
        signal = '' # signal name
        classifier = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pick
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 2, 30), 0, 60)[:]

        results, v_ss, first_ss, updated = classifier.scan_df(df, False)
        assert results is None, 'No data'
        assert v_ss is None, 'No data'
        assert first_ss is None, 'No data'
        assert updated is None, 'No data'
        print 'PASS'
```

```
PASS
```

In [7]:
```
        #test function
        signal = '' # signal name
        experimenter = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.p
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 56), 0, 1*60)[1950:1985]
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        #experimental_plot(df, classification, signal, smags[signal], False, filename='', xlim=None)
        assert len(classification) == 35, 'Data failed'

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 56), 0, 1*60)[1985:]
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        #experimental_plot(df, classification, signal, smags[signal], False, filename='', xlim=None)
```

```
        assert classification[0] == -1, 'reestablish ss window'

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 56), 0, 1*60)[3000:]
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        #experimental_plot(df, classification, signal, smags[signal], False, filename='', xlim=None)
        assert classification[0] == -1, 'reestablish ss window'

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 56), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        #experimental_plot(df, classification, signal, smags[signal], False, filename='', xlim=None)
        assert classification[0] == -1, 'reestablish ss window'
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 57), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert classification[0] != -1, 'reuse ss window'

        print 'PASS'
```

```
PASS
```

```
In [8]: signal = '' # signal name
        experimenter = ContinuousClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pic
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 56), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert len(classification) == 3600, 'Failed since not reaching end of window'
        assert len([x for x in classification if x == -1]) == 30, 'Failed since not reaching end of win
        assert len([x for x in classification if x >= 0]) == 3600-30, 'Failed since not reaching end of

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 23, 57), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert len(classification) == 3600, 'Failed since not reaching end of window'
        assert len([x for x in classification if x == -1]) == 0, 'Failed since not reaching end of wind
        assert len([x for x in classification if x >= 0]) == 3600, 'Failed since not reaching end of wi

        print 'PASS'
```

```
PASS
```

```
In [9]: signal = '' # signal name
        experimenter = ContinuousClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pic

        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 3, 58), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert len(classification) == 3600, 'Failed since not reaching end of window'
        assert len([x for x in classification if x == -1]) == 30, 'Failed since not reaching end of win
        assert all([np.isnan(x) for x in classification[1500:1559]]), 'Failed since not reaching end of

        experimenter = ClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.pickle')
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 3, 58), 0, 1*60)
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert len(classification) == 3600, 'Failed since not reaching end of window'
        assert len([x for x in classification[0:10] if x == -1]) == 10, 'Failed since not reaching end
        assert all([np.isnan(x) for x in classification[1500:1559]]), 'Failed since not reaching end of

        experimenter = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/J48Classifier.p
        df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 3, 58), 0, 1*60)
```

```
        classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df(
        assert len(classification) == 3600, 'Failed since not reaching end of window'
        assert len([x for x in classification[0:10] if x == -1]) == 10, 'Failed since not reaching end
        assert all([np.isnan(x) for x in classification[1500:1503]]), 'Failed since not reaching end of
        assert all([x == -1 for x in classification[1503:1559]]), 'Failed since not reaching end of win

        print 'PASS'
```

```
PASS
```

```
In [10]: signal = '' # signal name
         experimenter = ConfigurableClassifyingExperimenter(signal, smags, sangles, 'txt/one_vs_one_cla
         df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 0, 22), 0, 1*60)
         classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df
         assert len(classification) == 3600, 'Failed since not reaching end of window'

         df = datafiles.pmudata.dataframe(datetime(2014, 10, 5, 0, 23), 0, 1*60)
         classification, steady_states, first_steady_state, updated_steady_state = experimenter.scan_df
         assert len(classification) == 3600, 'Failed since not reaching end of window'
         assert len(steady_states) == 3600, 'Failed since not reaching end of window'
         assert len([x for x in classification if x == -1]) == len([x for x in steady_states if x is No

         print 'PASS'
```

```
PASS
```

```
In [ ]:
```