COSC 522: Machine Learning - Final Project - Milestone 3

Aaron Wilson, Adrian Cross, and Xuesong Fan

Preprocessing

The training and testing data sets contain 8,712 and 20,336 samples with 800,000 data points per sample, respectively. Each training/testing sample represents one cycle of a 50 Hz voltage waveform, sampled at 40 megasamples-per-second (Msps). Using techniques from [1], each waveform was filtered using a 10^{th}-order high-pass Butterworth filter with a low cut-off frequency of 10 kHz, and denoised using the discrete wavelet transform (DWT) (Figure 1). From the denoised signal, 12 features were extracted: signal mean, signal standard deviation, signal skewness, signal kurtosis, number of negative peaks, number of positive peaks, mean peak width, mean peak height, max peak width, max peak height, min peak width, and min peak height.

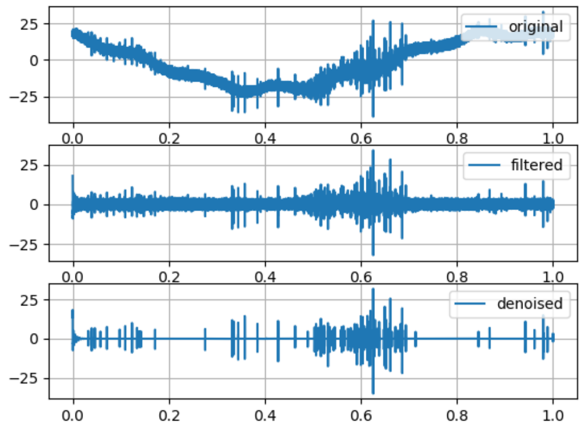

Figure 1 Denoising Process

Classification – Minimum Distance Classifier (Case 1)

The given testing set was not labeled. Therefore, a "pseudo-labeling" procedure was performed to predict pseudo-labels for the testing set, which were then used to re-train the model.

From the training data, $P(\omega_0) = 0.94$ and $P(\omega_1) = 0.06$, where ω_0 represents the class in which partial discharge is not present and ω_1 represents the class where partial discharge is present. Normalization was first performed on the training and testing datasets. Principal component analysis (PCA) was then performed on both the training and testing sets based on the features of the training dataset, allowing for an error rate of $\varepsilon = 0.1$. This reduced the dimension of the data from 12 features to 8. Minimum distance classifier (Case I) was used for the classification. The trained model was used to generate the pseudo-labels on the testing dataset. After concatenating the training and the testing datasets, the data was used to re-train the model and predict the labels. The performance evaluation of this classifier, using the testing set with the pseudo-labels, is given in Table 1.

Table 1. Case 1 Classifier Performance Evaluation

Accuracy (%)	97.99
# TP	116
# TN	28349
# FP	26
# FN	557
Sensitivity (%)	17.24
Specificity (%)	99.91

References

[1] T. Vantuch, "Analysis of Time Series Data", Ph.D Disssertation, Dept. Comp. Sci., VŠB – Technical University of Ostrava

Appendix A – Preprocessing Code

```
import numpy as np
import scipy.signal as sig
from scipy.signal import find_peaks, peak_widths, peak_prominences, butter
from scipy import signal, stats
import pywt, time, pandas as pd
import csv
import pyarrow.parquet as pq

# NOTE - THIS CODE TAKES SEVERAL HOURS TO RUN!

# "Most" of the following code taken from https://www.kaggle.com/jackvial/dwt-signal-
denoising
# Original functions marked with "#AJW" comments next to the function name


def maddest(d, axis=None):
    """
    Mean Absolute Deviation
    """
    return np.mean(np.absolute(d - np.mean(d, axis)), axis)


def high_pass_filter(x, low_cutoff=1000, sample_rate=40e6):
    """
    From @randxie https://github.com/randxie/Kaggle-VSB-
Baseline/blob/master/src/utils/util_signal.py
    Modified to work with scipy version 1.1.0 which does not have the fs parameter
    """

    # nyquist frequency is half the sample rate
https://en.wikipedia.org/wiki/Nyquist_frequency
    nyquist = 0.5 * sample_rate
    norm_low_cutoff = low_cutoff / nyquist

    # Fault pattern usually exists in high frequency band. According to literature,
the pattern is visible above 10^4 Hz.
    # scipy version 1.2.0
    # sos = butter(10, low_freq, btype='hp', fs=sample_fs, output='sos')

    # scipy version 1.1.0
    sos = butter(10, Wn=[norm_low_cutoff], btype='highpass', output='sos')
    filtered_sig = signal.sosfilt(sos, x, axis=0)

    return filtered_sig


def denoise_signal(x, wavelet='db4', level=1):
    """
    1. Adapted from waveletSmooth function found here:
    http://connor-johnson.com/2016/01/24/using-pywavelets-to-remove-high-frequency-
noise/
    2. Threshold equation and using hard mode in threshold as mentioned
    in section '3.2 denoising based on optimized singular values' from paper by Tomas
```

```
Vantuch:

http://dspace.vsb.cz/bitstream/handle/10084/133114/VAN431_FEI_P1807_1801V001_2018.pdf
    """

    # Decompose to get the wavelet coefficients
    coeff = pywt.wavedec(x, wavelet, mode="per")

    # Calculate sigma for threshold as defined in
http://dspace.vsb.cz/bitstream/handle/10084/133114/VAN431_FEI_P1807_1801V001_2018.pdf
    # As noted by @harshit92 MAD referred to in the paper is Mean Absolute Deviation
not Median Absolute Deviation
    sigma = (1 / 0.6745) * maddest(coeff[-level])

    # Calculte the univeral threshold
    uthresh = sigma * np.sqrt(2 * np.log(len(x)))
    coeff[1:] = (pywt.threshold(i, value=uthresh, mode='hard') for i in coeff[1:])

    # Reconstruct the signal using the thresholded coefficients
    return pywt.waverec(coeff, wavelet, mode='per')


### THE FOLLOWING 3 FUNCTIONS TAKEN FROM https://www.kaggle.com/c/vsb-power-line-
fault-detection/discussion/86616#latest-501584
def remove_false_peak(signal, p1, p2, maxDistance=10):
    peak_diff = np.diff(p2)
    if len(peak_diff) == 0:
        return p1
    ticks = []
    for i, d in enumerate(peak_diff):
        ratio = signal[p2[i+1]]/signal[p2[i]]
        if d < maxDistance and -0.25 > ratio and ratio > -4:
            ticks.append((p2[i], p2[i+1]))
    mask = np.array([True]*len(p1))
    for i, j in ticks:
        mask = mask & ((p1 < i) | (p1 > 500+j))
    return p1[mask]


def get_peaks(signal):
    p1_1, _ = find_peaks(signal, height=[5, 100])
    p1_2, _ = find_peaks(-signal, height=[5, 100])
    p1 = np.union1d(p1_1, p1_2)
    n_peaks, _ = find_peaks(-signal, height=[10, 100])
    p_peaks, _ = find_peaks(signal, height=[10, 100])
    p2 = np.union1d(n_peaks, p_peaks)
    p = remove_false_peak(signal, p1, p2, maxDistance=10)
    return np.intersect1d(p1_1, p), np.intersect1d(p1_2, p)


def extract_peak_feature(signal):
    p_peaks, n_peaks = get_peaks(signal)

    num_p, num_n = len(p_peaks), len(n_peaks)

    sig_peak_width = np.concatenate(
        [peak_widths(signal, p_peaks)[0], peak_widths(-signal, n_peaks)[0]])
    sig_peak_height = abs(signal[np.concatenate([p_peaks, n_peaks])])

    if num_n or num_p:
        height_mean = sig_peak_height.mean()
        height_max = sig_peak_height.max()
        height_min = sig_peak_height.min()
```

```python
        height_median = np.median(sig_peak_height)

        width_mean = sig_peak_width.mean()
        width_max = sig_peak_width.max()
        width_min = sig_peak_width.min()
        width_median = np.median(sig_peak_width)

        return np.array([num_n, num_p, width_mean, height_mean,
                         width_max, height_max, width_min, height_min])
    else:
        return np.zeros(8)


def extract_features(denoised_signal):  #AJW

    # Taken from Tomas Vantuch's PhD thesis "Analysis of Time Series Data"

    # Make sure signal is a pandas series type, for the entropy calculation.
    if type(denoise_signal) != pd.core.series.Series:
        denoised_signal = pd.Series(denoised_signal)


    # Mean
    sig_mean = np.mean(denoised_signal)

    # Standard deviation
    sig_std = np.std(denoised_signal)

    # Skewness
    sig_skw = stats.skew(denoised_signal)

    # Kurtosis
    sig_kur = stats.kurtosis(denoised_signal)

    # Peak features
    pk_features = extract_peak_feature(denoised_signal)

    return np.append([sig_mean, sig_std, sig_skw, sig_kur], pk_features)


def main():  #AJW

    t = np.linspace(0, 1, 800000)  # Only for plotting purposes

    y_train = list(pd.read_csv('input/metadata_train.csv')['target'])
    # y_test = list(pd.read_csv('input/metadata_test.csv')['target'])

    with open('training_data_new.csv', 'w', newline='') as outcsv_train:
        writer = csv.writer(outcsv_train)
        writer.writerow(['mean', 'std', 'skw', 'kur', 'ent', 'num_n_pks', 'num_p_pks',
'mean_pk_width',
                         'mean_pk_height', 'max_pk_width', 'max_pk_height',
'min_pk_width', 'min_pk_height'])
        outcsv_train.flush()

        for sample in range(0, len(y_train)):
            train_data = pq.read_pandas('input/train.parquet',
columns=[str(sample)]).to_pandas()
            filt_sig = high_pass_filter(train_data, low_cutoff=10000,
sample_rate=40e6)
            wavelet_sig = denoise_signal(filt_sig[:, 0], wavelet='db4', level=1)
            features = extract_features(wavelet_sig)
            writer.writerow(features)
```

```
            outcsv_train.flush()

    print('Training data finished!')

    with open('test_data_new.csv', 'w', newline='') as outcsv_test:
        writer = csv.writer(outcsv_test)
        writer.writerow(['mean', 'std', 'skw', 'kur', 'num_n_pks', 'num_p_pks',
'mean_pk_width',
                         'mean_pk_height', 'max_pk_width', 'max_pk_height',
'min_pk_width', 'min_pk_height'])
        outcsv_test.flush()

        for sample in range(8712, 29048):
            test_data = pq.read_pandas('input/test.parquet',
columns=[str(sample)]).to_pandas()
            filt_sig = high_pass_filter(test_data, low_cutoff=10000, sample_rate=40e6)
            wavelet_sig = denoise_signal(filt_sig[:, 0], wavelet='db4', level=1)
            features = extract_features(wavelet_sig)
            writer.writerow(features)
            outcsv_test.flush()

        print('Testing data finished!')


if __name__ == "__main__":
    main()
```

Appendix B – Training and Classification Code

```
"""
COSC 522
Final Project - Milestone 3
Adrian Cross, Xuesong Fan, and Aaron Wilson
"""

import numpy as np
import sys


def load_training(file):
    """load training data from file"""
    data = np.loadtxt(file, delimiter=',', skiprows=1)
    X = data[:, :-1]
    y = data[:, -1].astype(int)
    return X, y


def load_testing(file):
    """load testing data from file"""
    data = np.loadtxt(file, delimiter=',', skiprows=1)
    return data


def euc2(a, b):
    """euclidean distance square"""
    return np.dot(np.transpose(a - b), (a - b))
```

```python
def mah2(a, b, sigma):
    """mahalanobis distance square"""
    return np.dot(np.transpose(a - b), np.dot(np.linalg.inv(sigma), (a - b)))


def norm(Tr, Te):
    """normalize the data"""
    m_ = np.mean(Tr, axis=0)
    sigma_ = np.std(Tr, axis=0)
    nTr = (Tr - m_) / sigma_
    nTe = (Te - m_) / sigma_
    return nTr, nTe


def pca(Tr, Te, err):
    """PCA"""
    Tr_cov = np.cov(np.transpose(Tr))
    eigval, eigvec = np.linalg.eig(Tr_cov)
    sort_eigval = eigval[np.argsort(-eigval)]
    sort_eigvec = eigvec[np.argsort(-eigval)]
    tot_ = np.sum(sort_eigval)
    sum_ = 0.0
    for i in range(len(sort_eigval)):
        sum_ += sort_eigval[i]
        err_ = 1 - sum_ / tot_
        if err_ <= err:
            break
    print(i + 1, 'features were kept with the error rate of', "%.2f" %(err_ * 100),
'%')
    P_ = sort_eigvec[:i + 1]
    pTr = Tr.dot(np.transpose(P_))
    pTe = Te.dot(np.transpose(P_))
    return pTr, pTe


def fld(Tr, y, Te):
    """FLD"""
    covs_, means_, n_, S_ = {}, {}, {}, {}
    Sw_ = None
    classes_ = np.unique(y)
    for c in classes_:
        arr = Tr[y == c]
        covs_[c] = np.cov(np.transpose(arr))
        means_[c] = np.mean(arr, axis=0)  # mean along rows
        n_[c] = len(arr)
        if Sw_ is None:
            Sw_ = (n_[c] - 1) * covs_[c]
        else:
            Sw_ += (n_[c] - 1) * covs_[c]
    w_ = np.dot(np.linalg.inv(Sw_), means_[0]-means_[1])
    fTr = Tr.dot(np.transpose(w_))
    fTe = Te.dot(np.transpose(w_))
    return fTr, fTe
```

```python
def eva(y, y_model):
    """ return accuracy score """
    assert len(y) == len(y_model)
    accu = np.count_nonzero(y == y_model) / len(y)
    TP = TN = FP = FN = 0
    for i in range(len(y)):
        if y_model[i] == y[i] == 1:
            TP += 1
        if y_model[i] == y[i] == 0:
            TN += 1
        if y_model[i] == 1 and y_model[i] != y[i]:
            FP += 1
        if y_model[i] == 0 and y_model[i] != y[i]:
            FN += 1
    sens = TP / (TP + FN)
    spec = TN / (TN + FP)
    print('accuracy = ', "%.2f" %(accu * 100), '%')
    print('TP = ', TP)
    print('TN = ', TN)
    print('FP = ', FP)
    print('FN = ', FN)
    print('sensitivity = ', "%.2f" %(sens * 100), '%')
    print('specificity = ', "%.2f" %(spec * 100), '%')
    return None


class mpp:
    def __init__(self, case=1):
        self.case_ = case

    def fit(self, Tr, y):
        # derive the model
        self.covs_, self.means_, self.pw_ = {}, {}, {}
        self.covsum_ = None

        self.classes_ = np.unique(y)  # get unique labels as dictionary items
        self.classn_ = len(self.classes_)

        for c in self.classes_:
            arr = Tr[y == c]
            self.covs_[c] = np.cov(np.transpose(arr))
            self.means_[c] = np.mean(arr, axis=0)  # mean along rows
            if self.covsum_ is None:
                self.covsum_ = self.covs_[c].copy()
            else:
                self.covsum_ += self.covs_[c]
            self.pw_[c] = len(arr) / len(y)

        # used by case II
        self.covavg_ = self.covsum_ / self.classn_

        # used by case I
        if type(self.covavg_) != np.ndarray:
            self.varavg_ = self.covavg_.copy()
```

```
        else:
            self.varavg_ = np.sum(np.diagonal(self.covavg_)) / len(self.covavg_)

        return None

    def disc(self, Te):
        # eval all data
        y = []
        disc = np.zeros(self.classn_)
        ne = len(Te)

        if type(self.covavg_) != np.ndarray:
            for i in range(ne):
                for c in self.classes_:
                    if self.case_ == 1:
                        edist2 = (Te[i] - self.means_[c]) ** 2
                        disc[c] = -edist2 / (2 * self.varavg_) + np.log(self.pw_[c])
                    elif self.case_ == 2:
                        mdist2 = ((Te[i] - self.means_[c]) ** 2) / self.covavg_
                        disc[c] = -mdist2 / 2 + np.log(self.pw_[c])
                    elif self.case_ == 3:
                        mdist2 = ((Te[i] - self.means_[c]) ** 2) / self.covs_[c]
                        disc[c] = -mdist2 / 2 - np.log(self.covs_[c]) / 2 +
np.log(self.pw_[c])
                    else:
                        print("Can only handle case numbers 1, 2, 3.")
                        sys.exit(1)
                y.append(disc.argmax())
        else:
            for i in range(ne):
                for c in self.classes_:
                    if self.case_ == 1:
                        edist2 = euc2(self.means_[c], Te[i])
                        disc[c] = -edist2 / (2 * self.varavg_) + np.log(self.pw_[c])
                    elif self.case_ == 2:
                        mdist2 = mah2(self.means_[c], Te[i], self.covavg_)
                        disc[c] = -mdist2 / 2 + np.log(self.pw_[c])
                    elif self.case_ == 3:
                        mdist2 = mah2(self.means_[c], Te[i], self.covs_[c])
                        disc[c] = -mdist2 / 2 - np.log(np.linalg.det(self.covs_[c]))
/ 2 \
                                  + np.log(self.pw_[c])
                    else:
                        print("Can only handle case numbers 1, 2, 3.")
                        sys.exit(1)
                y.append(disc.argmax())

        return y


def main():
    Xtrain, ytrain = load_training('training_data_new.csv')
    Xtest = load_testing('test_data_new.csv')

    nXtrain, nXtest = norm(Xtrain, Xtest)
```

```python
    pXtrain, pXtest = pca(nXtrain, nXtest, 0.1)

    model = mpp()
    model.fit(pXtrain, ytrain)
    y_pseudo = model.disc(pXtest)

    pX = np.concatenate((pXtrain, pXtest))
    y = np.concatenate((ytrain, y_pseudo))

    model_whole = mpp()
    model_whole.fit(pX, y)
    y_model = model_whole.disc(pX)
    eva(y, y_model)


if __name__ == "__main__":
    main()
```