**COSC 522: Machine Learning - Final Project - Milestone 3**

**Aaron Wilson, Adrian Cross, and Xuesong Fan**

**Preprocessing**

A screenshot of a video game

Description automatically generatedThe 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 10th-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, and , where represents the class in which partial discharge is not present and 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 . 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()  
 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))  
  
  
**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  
  
  
**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)  
 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()