# AMATH 582 Final Project

Tyrone DeSilva

February 21, 2020

**Abstract**

Partial discharge is a type of fault in electric transmission lines that can cause damage to equipment. Detecting partial discharge is necessary identify faulty equipment before failure occurs. We apply Multi-Resolution Analysis(MRA) to the problem of classifying faults from power line Voltage measurements.

## 1 Introduction and Overview

This data set and problem are from a data science competition [2]. The data set contains 8,712 labeled training samples and 20,337 unlabeled test samples. Predictions on the unlabeled test samples are used to score submissions. The scoring metric used in this competition is the Matthew's correlation coefficient(MCC), which penalizes both poor recall and poor precision.

One measurement consists of 3 samples, each measured from a phase of the 3-phase power delivery scheme. In both the training and test set, each signal has an ordinal label denoting it's phase. This is the breakdown of number of faults per measurement:

| # Faults | Measurment Count |
|---|---|
| 0 | 2710 |
| 1 | 19 |
| 2 | 19 |
| 3 | 156 |

It is clear that the binary classification problem has a class imbalance; just 6% of the signals are faulty. It is also clear that one phase displaying a fault is related to whether the other 2 phases are also labelled faults. All 3 signals being labelled faults occurs much more frequently than just 1 or 2 phases having faults.

Each signal has 800,000 Voltage measurements taken over 20 milliseconds. The sampling frequency $F_s$ is 40,000,000. See Figure 1 and Figure 2 for examples of the signals in the time domain.

In this case, there are two challenges: i) reducing the dimensionality of each signal, ii) dealing with class imbalance. There's also the issue of encoding information about the other phases, but for now we will treat each signal as independent, even though that is as simplification.

## 2 Theoretical Background

To preprocess our signals for classification, we used Principal Component Analysis(PCA) and the Continuous Wavelet Transform(CWT).

PCA is a way to decompose data into the components which maximize variance, with the aim of reducing the dimensionality of the data. This is done by finding a diagonalization of the data. One method for PCA is diagonalizing using Singular Value Decomposition(SVD), which is the decomposition of an arbitrary matrix $A$ into the components: $U\Sigma V^*$ where $\Sigma$ is a diagonal matrix, and $U$, $V$ are unitary matrices. The variance of the $i$th component is proportional to $\sigma_i^2$ where $\sigma_i$ is the $i$th diagonal element of $\Sigma_i$. We can choose how many components of the projected matrix to keep by calculating how many of the principal components are needed to account for a given proportion of the total variance.

The CWT is given as

$$W_\psi[f](a,b) = (f, \psi_{a,b})$$

where

$$\psi_{a,b} = \frac{1}{\sqrt{a}}\psi(\frac{t-b}{a})$$

$a$ and $b$ are the dilation and translation terms. The implementation of the CWT we used to get our scalogram features are just a discretization of this equation. Instead of $a$, $b$ being continuous we have

$$\psi_{n,m} = \frac{1}{2^{\frac{n}{v}}}\psi(\frac{t}{2^{\frac{n}{v}}} - mF_s) \quad \text{for} \quad m, n, v \in \mathbb{Z}^+$$

CWT offers us the ability to capture both high frequency behavior with good time resolution and lower frequency behavior with less time resolution. MRA is how we apply the wavelet basis to get a representation of our signal at varying time and frequency resolution.

## 3 Algorithm Implementation and Development

In addition to preprocessing our data to reduce dimensionality, we applied Synthetic Minority Over-sampling Technique(SMOTE) to create a balanced training set. In order to choose a model, we compared model performance on MCC score using 5-fold cross validation. This is the algorithm we used to evaluate each model.

```
Algorithm 1: Model Selection
 Input: $X$, $y$: training signals along with binary labels
 Output: Cross validation Score
   $X \leftarrow$ Scalograms of $X$
   Split $X$ and $y$ into 5 folds.
   for Each fold do
      Hold that fold out as training fold, the other 4 are training folds.
      $P \leftarrow$Principal Component Vectors of training set
      $X \leftarrow$ Projection of all scalograms onto $P$
      Resample training folds.
      Fit model on training folds of $X$, $y$
      Score model on test fold of $X$, $y$
   end for
   return  Cross Validation Scores
```

## 4 Computational Results

Previously, we've used time-frequency analysis along with SVD to analyze musical scores[1]. This was initially our plan for this problem, but we ran into some short-comings of using Short Time Fourier Transform(STFT) for time-frequency analysis. Figure 1 shows how STFT fails to encode both the low and high frequency parts of the signal. We tried tuning the STFT parameters to get better time localization at high frequency, but were never able to get a transform that kept both good time and frequency resolution. We wanted to also capture high frequency behavior with good time resolution, so we decided to use MRA.

Figure 3 shows some of the modes that were most significant in our training set. It seems like the low frequency behavior is captured in the first few modes. The first117 modes captured 99.5% of the variance, which is a big reduction from the 15,000 dimension scalogram.

MCC score is a correlation score, so it takes on values between -1 and 1. A MCC score of 0 is equivalent to random guessing. A score of 1 is a perfect classifier, and a score of -1 means the predictions are always wrong Our final model had an average cross-validation MCC score of 0.210. To get the score on the test set, we had to submit the predictions to the Kaggle website. They have a public score(57% of test data that is available to be scored during the competition) and a private score(based on the held out 43% of the data that is held out until the competition deadline). Oddly our model did well on the private dataset, which is the one that counts, with a MCC score of 0.216. We scored poorly on the public set, with a MCC score of 0.056.

## 5 Summary and Conclusions

Our final model a gradient boosting model with SMOTE built using scalogram features whose dimension was reduced using PCA We used cross validation to try a lot of different models with and without resampling techniques. In addition to SMOTE, we also tried undersampling. In some cases, random undersampling was pretty close to SMOTE in terms of performance. Without resampling, the classifier performance was poor–almost as bad as random guessing. The other models evaluated were logistic regression and Support Vector Classifier(SVC). Both of these under-performed the gradient boosting model.

| # Model | Cross-val MCC Score |
|---|---|
| Logistic Regression | 0.033 |
| Logistic Regression with SMOTE | 0.003 |
| Logistic Regression with Undersampling | 0.013 |
| SVC | 0.0 |
| SVC with SMOTE | 0.182 |
| Gradient Boosting | 0.085 |
| Gradient Boosting with SMOTE | 0.210 |
| Gradient Boosting with Undersampling | 0.166 |

The vast majority of our time spent on this project was on preprocessing. Initially we were planning on using the same technique as [1], STFT and PCA. It seemed like there were some high frequency, time localized behavior happening in the signals. Without having much background knowledge on the problem domain, it was impossible to say for certain that we didn't need good time localization of the high frequency signals. Downsampling the signals before applying the wavelet transform was necessary just to get everything to fit in memory.

One alternative we were looking at, and which was mentioned in [3], is to use the Discrete Wavelet Transform(DWT) instead of CWT. DWT is more efficient, as the CWT contains a lot of redundant information. There's also more easily available libraries to both apply DWT efficiently. However DWT doesn't produce a nice scalogram.

Finally, it's still unclear why performance on the public and private test sets are so different. Test set performance is very similar to cross validation performance. There may be pathological examples added into the public test set, in order to throw people off who are trying to game the public test set for information about the private test set.

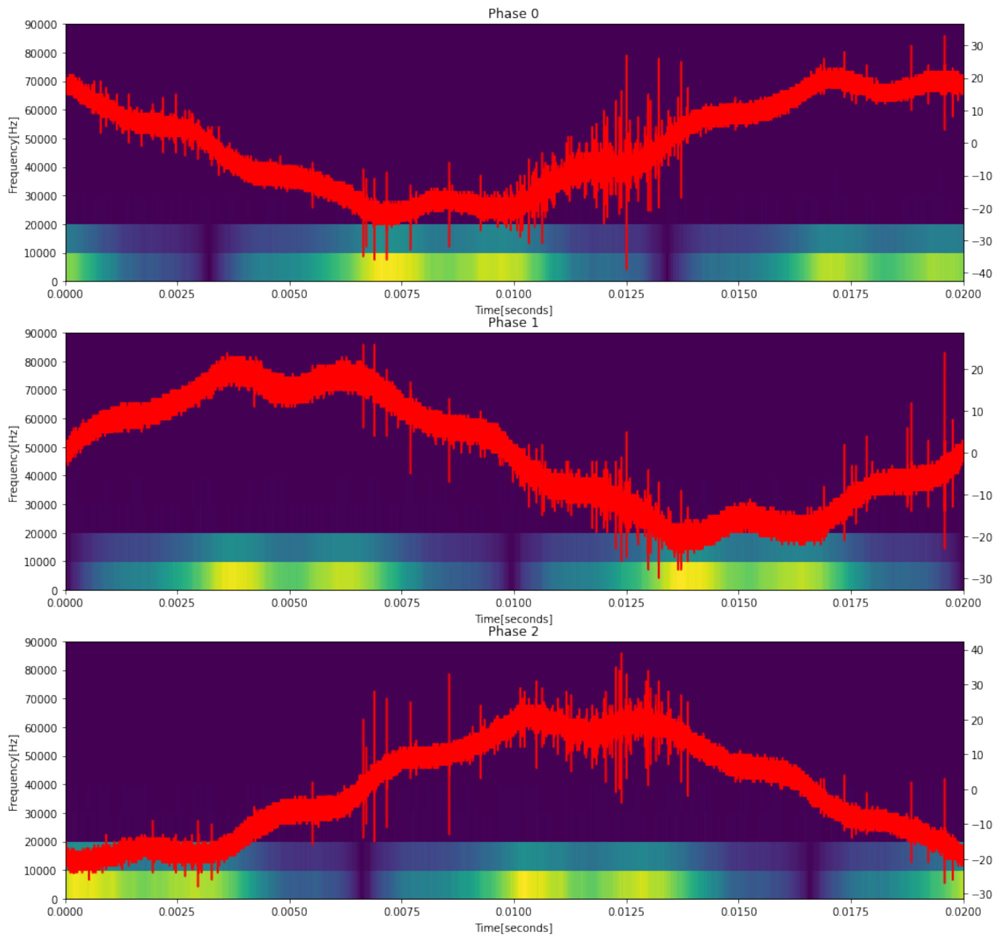

Figure 1: Periodograms for three phases along with original signals.

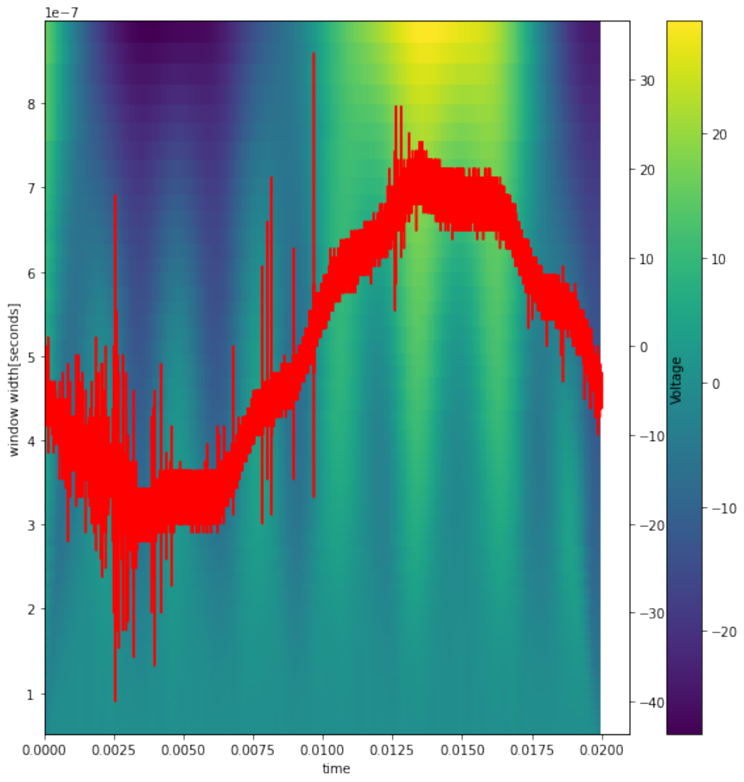

Figure 2: Scalogram with original signal.

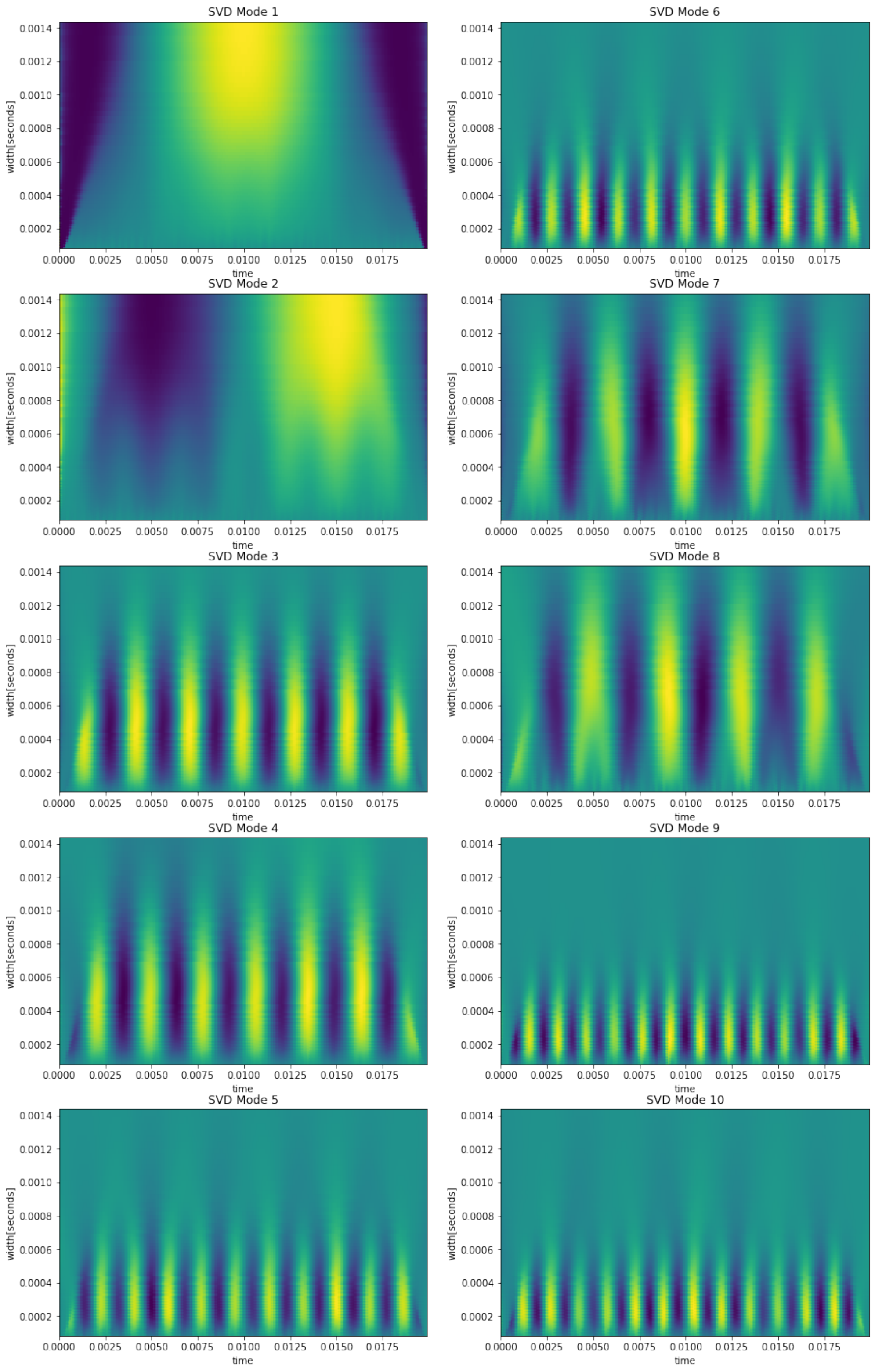

Figure 3: PCA Modes

## References

[1] Tyrone DeSilva. "AMATH582 HW4a". In: (2020). URL: https://github.com/tqhdesilva/AMATH582HW04/blob/master/music/report.pdf.

[2] VSB - T.U. of Ostrava Kaggle Enet Centre. *VSB Power Line Fault Detection*. 2014. URL: https://www.kaggle.com/c/vsb-power-line-fault-detection/data.

[3] Manohar Singh, Bijaya Panigrahi, and R.P. Maheshwari. "Transmission line fault detection and classification". In: *2011 International Conference on Emerging Trends in Electrical and Computer Technology, ICETECT 2011* (Mar. 2011). DOI: 10.1109/ICETECT.2011.5760084.

## Appendix A Python Functions

- `scipy.signal.cwt` Apply discretized CWT to signal.
- `sklearn.ensemble.GradientBoostingClassifier` Iterative ensemble learner for classification.
- `sklearn.model_selection.cross_validation` Fit and score pipeline using k-fold cross-validation.
- `imblearn.oversample.SMOTE` Apply SMOTE to generate more instances of non-majority class.

## Appendix B Python Code

### B.1 preprocess.py

```
from scipy import signal
import load
import numpy as np
from tqdm import tqdm
import argparse


Fs = 40000000
n = int(Fs * 20e-3)
k = 24


def wavelet(s):
    downsampled = signal.resample(s, n // 2 // 1600)
    widths = [2 ** (j / k) for j in range(1, 101)]
    z = signal.cwt(downsampled, signal.ricker, widths)
    return z


def preprocess(loader, output):
    signals, meta = loader()
    result = np.zeros((int(25000), signals.shape[1]), dtype=np.float32)
    for i in tqdm(range(signals.shape[1])):
        z = wavelet(signals.iloc[:, i])
        z = np.ravel(z)
        result[:, i] = z.astype(np.float32)
    np.save(output, result)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--train", action="store_true", default=False)
    parser.add_argument("--test", action="store_true", default=False)
    args = parser.parse_args()
    if args.train:
        preprocess(load.load_train, "data/preprocessed/train.npy")
    if args.test:
        preprocess(load.load_test, "data/preprocessed/test.npy")
```

## B.2 load.py

```
import pandas as pd
import pyarrow.parquet as pq
import os


THIS_FILE_DIR = os.path.dirname(os.path.abspath(__file__))
DATA_DIR = os.path.join(THIS_FILE_DIR, "data/vsb-power-line-fault-detection")


def load_train(n_columns: int = None) -> (pd.DataFrame, pd.DataFrame):
    columns = None
    if n_columns:
        columns = [str(i) for i in range(n_columns)]
    train_data = pq.read_pandas(
        os.path.join(DATA_DIR, "train.parquet"), columns=columns
    ).to_pandas()
    train_meta = pd.read_csv(
        os.path.join(DATA_DIR, "metadata_train.csv"),
        index_col="signal_id",
        nrows=n_columns,
    )
    return train_data, train_meta


def load_test(n_columns: int = None) -> (pd.DataFrame, pd.DataFrame):
    columns = None
    if n_columns:
        columns = [str(i) for i in range(n_columns)]
    test_data = pq.read_pandas(
        os.path.join(DATA_DIR, "test.parquet"), columns=columns
    ).to_pandas()
    test_meta = pd.read_csv(
        os.path.join(DATA_DIR, "metadata_test.csv"),
        index_col="signal_id",
        nrows=n_columns,
    )
    return test_data, test_meta
```

## B.3 pca_modes.py

```
import numpy as np
import matplotlib.pyplot as plt
```

```
data = np.load("../data/preprocessed/train.npy")
means = np.reshape(np.mean(data, axis=1), (-1, 1))
centered = data - means
std = np.reshape(np.std(centered, axis=1), (-1, 1))
scaled = data / std
u, s, vh = np.linalg.svd(scaled, full_matrices=False)
fig, ax = plt.subplots(5, 2, figsize=(15, 25))
widths = [2 ** (j / 24) for j in range(1, 101)]
dt = 20e-3 / 250
y = np.array(widths) * dt
x = np.array([j * dt for j in range(250)])
for j in range(10):
    pos = (j % 5, j // 5)
    a = ax[pos[0]][pos[1]]
    a.pcolormesh(x, y, np.reshape(u[:, j], (100, 250)))
    a.set_title(f"SVD Mode {j + 1}")
    a.set_xlabel("time")
    a.set_ylabel("width[seconds]")
```

## B.4 pipeline.py

```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_validate
from imblearn.pipeline import make_pipeline
from sklearn.metrics import (
    matthews_corrcoef,
    precision_score,
    recall_score,
    make_scorer,
)
from imblearn.over_sampling import SMOTE
from joblib import dump
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import GradientBoostingClassifier

train_wavelets = np.load("../data/preprocessed/train.npy")
train_meta = pd.read_csv(
    "../data/vsb-power-line-fault-detection/metadata_train.csv", index_col="signal_id",
)

x = train_wavelets.T
y = train_meta.target.values


gb_smote_pipe = make_pipeline(
    StandardScaler(), PCA(n_components=117), SMOTE(), GradientBoostingClassifier(),
)
scores = cross_validate(
    gb_smote_pipe,
    x,
    y,
    cv=5,
    scoring={
        "mcc": make_scorer(matthews_corrcoef),
        "precision": make_scorer(precision_score),
        "recall": make_scorer(recall_score),
    },
    return_train_score=True,
    n_jobs=-1,
)

print(scores)
gb_smote_pipe.fit(x, y)
dump(gb_smote_pipe, "../data/models/gb_smote.joblib")
```

## B.5 plots.py

```
from scipy.signal import stft
from scipy import ndimage
from scipy import signal
import matplotlib.pyplot as plt
import numpy as np
from load import load_train

train, train_meta = load_train(6)
train.info()
train_meta.info()

Fs = 40000000
n = int(Fs / 10000)
overlap = None

fig, ax = plt.subplots(3, 1, figsize=(15, 15))
for i in range(3):
    f, t, z = stft(train.iloc[:, i].values, fs=Fs, nperseg=n, noverlap=overlap)
    ax[i].pcolormesh(t, f[:10], np.abs(z)[:10, :], vmin=0)
    ax[i].set_xlabel("Time[seconds]")
    ax[i].set_ylabel("Frequency[Hz]")
    ax[i].set_title(f"Phase {i}")
    ax2 = ax[i].twinx()
    ax2.plot([j / Fs for j in range(800000)], train.iloc[:, i], c="r")


downsampled = signal.resample(train.iloc[:, 4], 800000 // 2 // 1600)
k = 24
widths = [2 ** (j / k) for j in range(1, 101)]
z = signal.cwt(downsampled, signal.ricker, widths)
z_filt = z
plt.figure(figsize=(10, 10))
plt.pcolormesh(
    [1600 * j / (Fs / 2) for j in range(250)],
    [1 / (Fs / 2) * (j) for j in widths],
    z_filt,
)
plt.ylabel("window width[seconds]")
plt.xlabel("time")
plt.colorbar()
ax2 = plt.twinx()
ax2.plot([j / Fs for j in range(800000)], train.iloc[:, 4], c="r")
ax2.set_ylabel("Voltage")
```