Seizure detection using FFT, temporal and spectral correlation coefficients, eigenvalues and Random Forest

Michael Hills

Location: Sydney, Australia
Email: mhills@gmail.com
Competition: UPenn and Mayo Clinic's Seizure Detection Challenge

1 Summary

This document describes the winning submission of UPenn and Mayo Clinic's Seizure Detection Challenge hosted on Kaggle.com.

For data preprocessing the Fast Fourier Transform is applied to each 1 second clip, taking frequency magnitudes in the range 1-47Hz and discarding phase information. Correlation coefficients and their eigenvalues are then calculated in both the time and frequency domains and are appended to the FFT data to form the feature set. This feature set is then trained on a Random Forest classifier using 3000 trees. The approach is used to train per-patient classifiers.

2 Feature selection

The features used in the model were determined through heavy experimentation. Study of the literature of prior work in the field as well as my own study of signal processing and machine learning gave inspiration for ideas to experiment with. Features were kept or discarded based on their cross-validation performance.

The first thing I experimented with was Fast Fourier Transform on the belief that the various magnitudes of different frequencies would provide a good source of features. This turned out to be correct with FFT alone providing the best single-feature model compared to all other singular pre-processing steps I tried. Combinations of multiple features eventually proved to provide a better classification score once the right features were combined.

It was shown by Schindler, Leung, Elger & Lehnertz (2007) that correlation coefficients in the time domain and their corresponding eigenvalues are effective features for seizure detection. I applied this technique not only in the time domain of the original data, but also in the frequency domain.

Three sources of features are used to form the whole feature-set:

1. FFT magnitudes in the low frequency range 1 to 47Hz

   ```
   Time-series -> FFT -> Slice(1, 47) -> Magnitude -> Log10
   ```

   Fast Fourier Transform is applied to each 1 second clip across all EEG channels, taking log10 of the magnitudes of frequencies in the range 1-47Hz. Phase information is discarded. The output of each training example in this stage is in the shape (N, 47) where N is the number of EEG channels used for a given patient.

   The range 1-47Hz was chosen through trial and error while attempting dimensionality-reduction. It turned out to give a better result than using other frequency ranges such as 1-63Hz, 1-127Hz, 1-191Hz, or the entire frequency range. Note that I also omitted the 0Hz frequency, assuming this bias to be attributed to instrument error and would contribute only noise. This was confirmed with private leaderboard scores 0.94465 for 0-47Hz and 0.94847 for 1-47Hz.

   Finally the frequency features from all of the different channels are concatenated together when used for training.

   The following is python code performing the transformation with input data in the shape (N,M) where N is number of channels and M is number of samples:

   ```
   def fft(time_data):
       return np.log10(np.absolute(np.fft.rfft(time_data, axis=1)[:,1:48]))
   ```

2. Correlation coefficients (between EEG channels) and their eigenvalues in the frequency domain

   ```
   FFT 1-47Hz -> normalization -> correlation coefficients -> eigenvalues
   ```

   The output of the FFT stage with shape (N, 47) is then normalized across frequency buckets. For example, all of the 1Hz buckets for each EEG channel are treated as a single vector, subtracting the mean and dividing by standard deviation. This is done for every frequency bucket from 1Hz up to 47Hz. The correlation coefficients (N, N) matrix is calculated from this normalized matrix of (N, 47). Real eigenvalues are calculated on this correlation coefficients matrix with complex eigenvalues made

real by taking the complex magnitude. The output of this stage is the upper right triangle of the correlation coefficients matrix (as it is symmetric there is redundant data), and the eigenvalues sorted by magnitude.

Python code for performing the transformation:

```
def freq_corr(fft_data):
    scaled = sklearn.preprocessing.scale(fft_data, axis=0)
    corr_matrix = np.corrcoef(scaled)
    eigenvalues = np.absolute(np.linalg.eig(corr_matrix)[0])
    eigenvalues.sort()
    corr_coefficients = upper_right_triangle(corr_matrix) # custom func
    return np.concatenate((corr_coefficients, eigenvalues))
```

3. Correlation coefficients (between EEG channels) and their eigenvalues in the time domain.

   ```
   Time series -> normalization -> correlation coefficients -> eigenvalues
   ```

   Schindler, Leung, Elger & Lehnertz (2007) showed this to be an effective technique. However, I believe they normalized each channel independently, using mean and standard deviation across all the time-samples within a single channel. My cross-validation showed that normalizing across samples performed the same as not applying normalization at all, and that normalizing each feature performed worse. However, leaderboard submissions showed better results normalizing each feature.

   The features from this stage are calculated the same as stage 2, but the input source of data is the original time-series source data instead of the FFT output. Like the previous stage, both the upper triangle of the correlation coefficients and the sorted eigenvalues are used as features.

   Python code for performing the transformation:

   ```
   def time_corr(time_data):
       resampled = scipy.signal.resample(data, 400, axis=1) \
               if data.shape[-1] > 400 else data
       scaled = sklearn.preprocessing.scale(resampled, axis=0)
       corr_matrix = np.corrcoef(scaled)
       eigenvalues = np.absolute(np.linalg.eig(corr_matrix)[0])
       corr_coefficients = upper_right_triangle(corr_matrix) # custom func
       return np.concatenate((corr_coefficients, eigenvalues))
   ```

The output of all 3 stages are combined together to form the feature set.

```
def transform(data):
    fft_out = fft(data)
    freq_corr_out = freq_corr(fft_out)
    time_corr_out = time_corr(data)
    return np.concatenate((fft_out.ravel(), freq_corr_out, time_corr_out))
```

The peculiarity in the solution is that normalizing across features (axis=0) instead of on samples performed better on the leaderboard, but worse in cross-validation. In cross-validation better performance was obtained normalizing across samples (axis=1) or by not normalizing at all.

3 Modelling techniques and training

My background is Computer Engineering with self-study of signal processing and machine learning. My focus was therefore on writing performant code that would allow me to systematically experiment with different classifiers and different mathematical transforms to get quick feedback about what works and what doesn't. It was a greedy brute force approach, trying the most promising techniques first but otherwise trying every technique I could find. To facilitate this heavy experimentation I wrote a rudimentary framework to allow quick cross-validation of many different combinations of data processing techniques and classifiers.

3.1 Experimentation framework

Like the feature selection, the chosen model for classification was determined through experimentation. Each run gives a cross-validation score matching the scoring criteria used by the competition leaderboard (ROC AUC). Combinations of feature set and classifier that gave higher scores were explored further. A list of results obtained by using different feature sets can be found in Section 9.

This approach was made easy using the scikit-learn python machine learning library which offers many different classifiers that can be easily substituted in the code. Many different classifiers were tried in succession from logistic regression to decision trees or support vector machines. There are too many in scikit-learn to list here. However Random Forest offered not only the best performance but also consistent performance. The performance of SVM for example changed wildly with only small changes to the tuning parameters. Some manual experimentation was then done to determine good parameters to maximise the Random Forest performance.

The following python scikit-learn classifier was used to train the winning submission:

```
RandomForestClassifier(n_estimators=3000, min_samples_split=1,
        bootstrap=False, random_state=0)
```

3.2 Cross-validation

Initially I split the training set and cross-validation randomly. However scores achieved in cross-validation varied a significant amount from leaderboard scores suggesting this was not a good approach. Following advice on the forums from Kaggle user Alexandre, I split the ictal training data based on whole seizures. For example for a ratio of 0.25, if there were 4 seizures then 1 entire seizure was split out leaving the other 3 to train on. Some seizures for each patient were of different length. I sorted the seizures by length, then took the cross validation set from the middle, and trained on the shorter/longer seizures.

This gave cross-validation scores that matched the leaderboard very closely. However once cross-validation scores reached the 0.96-0.97 mark it was no longer sufficient to reliably rank models against each other and I fell back to submitting models based on intuition and checking performance on the leaderboard.

For example, my first submission of FFT combined with correlation data increased my score significantly on the public leaderboard from 0.95748 to 0.96961, an increase of 0.01213. However in cross-validation I saw an order of magnitude smaller increase of approximately 0.001. This indicates that there is perhaps room to improve in my cross-validation setup. These last numbers were taken with a cross-validation split of 50%.

Other splits to try might be to train on the medium-length seizures, and cross-validate on the longer/shorter ones. Another option is to use k-fold cross-validation, however that takes much longer training time which I did not wish to wait for.

3.3 Ictal training data generation

This was not used in the winning submission, but private leaderboard results showed this would have improved on the winning submission (no-gen 0.96288 vs with-gen 0.96331) although it showed worse performance in cross-validation (no-gen 0.96147 vs with-gen 0.95886). For this reason it deserves a mention here.

The competition training data for ictal segments were given in sequential 1 second clips, each marked with a latency representing the sequence. Overlapping segments can be created by taking the second half of clip 0, with the first half of clip 1, and labelling it 0.5. This gave clips with latency 0.5, 1.5, 2.5, ..., 13.5, 15.5, 16.5 and so on. The generated clip for 14.5 is omitted as the second prediction to be made is whether the test clip falls within latencies 0 to 15, or greater than 15. A clip at 14.5 exists halfway in both early and late classes, hence it was omitted in attempt to keep the threshold clear to classifier as to what is early and late.

4 Code description

The code can be found at `https://github.com/MichaelHills/seizure-detection`

There is too much code to to consider writing the documentation in this document, however a brief overview is possible.

- **common/**
  - **data.py**: Data caching interface, will attempt to dump to hickle format but falls back to pickle if it fails
  - **pipeline.py**: Implementation of the Pipeline class, contains whether ictal training data should be generated and what transforms to use
  - **io.py**: Helper methods for saving and loading both hickle and pickle files
  - **time.py**: Helper methods for getting a unix timestamp in seconds
- **seizure/**
  - **tasks.py**: This file contains the Task framework, defining units of computation with cacheable output (LoadIctalDataTask, LoadInterictalDataTask, LoadTestDataTask, TrainingDataTask, CrossValidationScoreTask, TrainClassifierTask, MakePredictionsTask)
  - **transforms.py**: This file contains all of the various transforms such as FFT, Magnitude, Log10, etc
- **train.py**: Train the classifiers and write them to the data cache
- **predict.py**: Make predictions on the test data (sourced from the competition data directory)
- **cross_validation.py**: Run cross-validation across selected pipelines and classifiers
- **seizure_detection.py**: The main source file called by the above 3 files setting up and running tasks with lists of pipelines and classifiers
- **SETTINGS.json**: configuration keys
  - **competition-data-dir**: directory containing the downloaded competition data
  - **data-cache-dir**: directory the task framework will store cached data
  - **submission-dir**: directory submissions are written to

5 Dependencies

5.1 Required

- Python 2.7
- scikit_learn-0.14.1
- numpy-1.8.1
- pandas-0.14.0
- scipy
- hickle (plus h5py and hdf5, see `https://github.com/telegraphic/hickle` for installation details)

5.2 Optional (to try out various data transforms)

- pywt (for Daubechies wavelet)
- scikits talkbox (for MFCC)

6 How to generate the solution

The **README.md** at `https://github.com/MichaelHills/seizure-detection` contains instructions for running the code. However to simply generate the winning submission follow these steps:

1. Setup python environment with all the necessary dependencies
2. Place the competition data in the project base directory under *seizure-data/* such that this forms paths like *seizure-data/Dog_1*
3. Run ./predict.py
4. Find the submission file under *submissions/*

7 Additional comments and observations

There are so many different mathematical techniques that can be applied to pre-process the data. Similarly there are so many different classifiers that can be used for training. Scikit-learn provides a common-interface for trying many different classifiers very easily.

However I have yet to come across a library that makes it very easy to try different pre-processing steps. With a lot of computational power and optimised code one can quickly iterate over many different mathematical techniques and over many different classifiers.

To implement data pre-processing steps I wrote a Pipeline framework, specifying single transform steps to execute on the data. This made it easy to try out small variations such as whether to use log10 on the FFT magnitudes or to try using the phases. Still I had to implement these transform steps 1 by 1, often fitting library functions into my Pipeline transforms and less often writing custom implementations.

```
# an example pipeline
Pipeline(gen_ictal=False, pipeline=[FFT(), Slice(1, 48), Magnitude(), Log10()])
```

Common patterns have emerged for good choices of machine learning classifiers and have been turned into excellent libraries such as scikit-learn. I feel the area of data preprocessing should have this too. A library which lets you easily try out best practice data preprocessing would be very valuable.

In software development, the development cycle is writing code and then running the code to see if it works whether through automated testing or manual testing. Having a fast development cycle means you can see results faster, fix issues sooner, and not lose focus while waiting for the compiler or some other slow tool. This also applies to machine learning where the cycle is training your model and then checking your cross-validation score. The faster this can be done the sooner you can discard bad models as well as find better models.

I started working on this competition with a from-scratch codebase. Immediately I ran into problems with preprocessing data and training models being slow. I put a lot of effort into optimising the code, writing a Task framework to re-use previously-computed data. With this in place many different classifiers could be run on the same data without having to recompute the data every time.

Before optimisation, the data preprocessing steps would take minutes to run. After optimisation it would only take seconds. Additionally my laptop would no longer crash as I took great care in only holding in RAM what was needed, leaving everything else cached on disk. With more streamlined performance and memory usage, I could train many models simultaneously and compare their outputs. A full cross-validation run across all patients using FFT and a reduced number of trees in the Random Forest (150 instead of 3000) can be done in under 4 minutes. Each run can also specify many pipelines and many classifiers, queueing them all up to run and overnight I would have a new batch of results.

For this competition I used a MacBook Pro with 2.7GHz Core i7, 16GB RAM, 512GB SSD. Having a powerful machine helped greatly in being able to handle the size of the input data, as well as quickly loading cached data to and from disk.

The framework I have written is by no means perfect and is missing the important feature of concatenating features from different pipelines without having to recompute

the original pipelines. If I continue to develop the framework then this will be the first feature I add, allowing arbitrary pipelines and combinations of data with zero redundant computations.

8 Simple features and methods

In cross-validation the log10 magnitude of FFT buckets in the frequency range 1Hz to 47Hz provided excellent baseline performance when trained with Random Forest, more than any other single technique.

9 Cross-validation results

The following is an example of the kind if output (although some data removed to make it easier to read in this document) I could get after 1 hour of trying different data processing steps. This includes trying various transforms such as FFT, frequency correlation, time correlation, MFCC, Daubechies wavelet and just basic stats of the time-series data. Note this is not an exhaustive list of the methods I tried.

The tags *us*, *usf* and *none* refer to the normalization option used with *us* referring to normalizing across all samples within a channel, and *usf* referring to normalizing each feature across all channels one by one.

```
0.96750 fft-with-time-freq-corr-1-48-r400-none_rf150mss1Bfrs0
0.96750 fft-with-time-freq-corr-1-48-r400-us_rf150mss1Bfrs0
0.96623 fft_slice1-48_mag_log10_rf150mss1Bfrs0
0.96496 fft_slice1-128_mag_log10_rf150mss1Bfrs0
0.96315 dwtdb4stats_rf150mss1Bfrs0
0.96257 fft_slice1-96_mag_log10_rf150mss1Bfrs0
0.96237 gen_fft-with-time-freq-corr-1-48-r400-none_rf150mss1Bfrs0
0.96237 gen_fft-with-time-freq-corr-1-48-r400-us_rf150mss1Bfrs0
0.96188 resample400_dwtdb4stats_rf150mss1Bfrs0
0.96151 fft_slice1-160_mag_log10_rf150mss1Bfrs0
0.96147 fft-with-time-freq-corr-1-48-r400-usf_rf150mss1Bfrs0
0.96069 fft_mag_log10_rf150mss1Bfrs0
0.95886 gen_fft-with-time-freq-corr-1-48-r400-usf_rf150mss1Bfrs0
0.95818 fft_slice1-64_mag_log10_rf150mss1Bfrs0
0.95462 resample400_mfcc_rf150mss1Bfrs0
0.95253 time-freq-correlation-1-48-r400-none_rf150mss1Bfrs0
0.95253 time-freq-correlation-1-48-r400-us_rf150mss1Bfrs0
0.93334 time-freq-correlation-1-48-r400-usf_rf150mss1Bfrs0
```

```
0.93158 stats_rf150mss1Bfrs0
0.92159 freq-correlation-1-48-nofft-none_rf150mss1Bfrs0
0.92159 freq-correlation-1-48-nofft-us_rf150mss1Bfrs0
0.92043 freq-correlation-1-48-nofft-us-noeig_rf150mss1Bfrs0
0.91571 time-correlation-r400-us-noeig_rf150mss1Bfrs0
0.90778 time-correlation-r400-none_rf150mss1Bfrs0
0.90778 time-correlation-r400-us_rf150mss1Bfrs0
0.90577 time-correlation-r400-usf_rf150mss1Bfrs0
0.90374 freq-correlation-1-48-nofft-usf_rf150mss1Bfrs0
0.83134 time-correlation-r400-us-nocorr_rf150mss1Bfrs0
0.81425 freq-correlation-1-48-nofft-us-nocorr_rf150mss1Bfrs0
```

References

[1] K Schindler, H Leung, CE Elger, K Lehnertz (2007) Assessing seizure dynamics by analysing the correlation structure of multichannel intracranial EEG. Brain 130(Pt 1):65-77