MEG Decoding with Hierarchical Combination of Logistic Regression and Random Forests

Heikki Huttunen, Oguzhan Gencoglu, Johannes Lehmusvaara, and Teemu Vartiainen

Department of Signal Processing, Tampere University of Technology, Finland
heikki.huttunen@tut.fi

1 Summary

This document describes the solution of the second place team in the DecMeg2014 brain decoding competition hosted at Kaggle.com. The model is a hierarchical combination of logistic regression and random forest. The first layer consists of a collection of 337 logistic regression classifiers, each using data either from a single sensor (31 features) or data from a single time point (306 features). The resulting probability estimates are fed to a 1000-tree random forest, which makes the final decision. In order to adjust the model to an unlabeled subject, the classifier is trained iteratively: After initial training, the model is retrained with unlabeled samples in the test set using their predicted labels from first iteration.

2 Preprocessing

The original data consists of MEG measurements with 306 channels and duration of 1.5 seconds, the stimulus shown at time 0.5 s from the beginning. Our only preprocessing operation is to decimate the data in time dimension by a factor of 8 resulting in a sampling frequency of 31.25 Hz. Moreover, the data prior to the stimulus is discarded. This decreases the amount of raw data and also removes the 50 Hz interference present in some channels. As a result, each trial is a matrix of size 306×31. We experimented with some denoising and outlier detection techniques, but eventually did not use them.

We also noticed that a solution with roughly equivalent accuracy can be achieved using only subset of the sensors. Namely, the sensors with high indices (154-306) tend to pop up in various feature selection approaches (see, e.g., Fig. 1). These correspond mostly to the back of the skull, which is where the visual cortex resides. Our second submission uses only 153 sensors as ordered by the distance from the back of the head, and the private score is only slightly less than with the full data (0.72624 compared to 0.72668 of the full model—public scores are equal at 0.72959).

The details of this document and the published code[1] correspond to the 306-sensor approach, because it seems more stable to change of parameters. Also the

[1] https://github.com/mahehu/decmeg

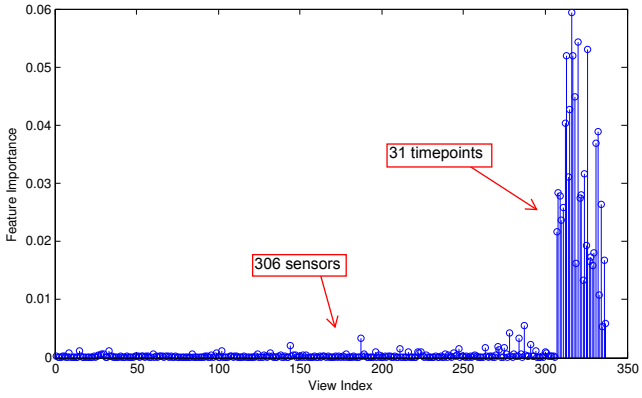

Fig. 1. The importances of features generated by the first layer as estimated by the random forest on the second layer.

variance of the prediction accuracy between subjects is typically smaller: with 306 sensors the cross-validated accuracy ranges between 0.5898 for subject 16 and 0.8266 for subject 4 (mean: 0.7314 ± 0.0609), while using 153 sensors results in 0.5610 and 0.8468 fur subjects 4 and 16, respectively (0.7311 ± 0.0664).

3 Prediction Model

The model is a two-layer hierarchical combination of logistic regression (LR) and random forest (RF) classifiers, which is illustrated in Figure 3. The first layer consists of a collection of 337 logistic regression classifiers, each using data either from a single sensor (31 features) or from a single time point (306 features). The resulting 337 probability estimates are fed to a 1000-tree random forest, which makes the final decision. The first layer features are illustrated in Figure 1, which plots the feature importances of the random forest for each first layer classifier.

The training procedure fits each LR classifier to the training data with the true labels as targets. After this, the class membership probability estimates of each LR classifier are fed as input to the RF, which is also trained with the true labels as targets.

In mathematical terms, denote the the input data as $\mathbf{X} \in \mathbf{R}^{N \times P \times T}$, and corresponding class labels as $\mathbf{y} \in \{0,1\}^N$. Then, the first layer consists of 306 classifiers F_j ($j = 1, 2, \ldots, 306$) in sensor dimension (receptive fields marked in red in Figure 3) and 31 classifiers G_k ($k = 1, 2, \ldots, 31$) in time dimension (receptive fields marked in blue in Figure 3). The two groups of logistic regression classifiers are trained by maximizing the L_2 penalized log-likelihoods [**?**]

$$\max_{\boldsymbol{\beta}, \beta_0} \left\{ \sum_{n=1}^{N} \mathbf{y}[n] \left(\beta_0 + \boldsymbol{\beta}^T \mathbf{X}[n, j, :] \right) - \log \left(1 + \exp \left(\beta_0 + \boldsymbol{\beta}^T \mathbf{X}[n, j, :] \right) \right) - \lambda \|\boldsymbol{\beta}\|_2^2 \right\}$$

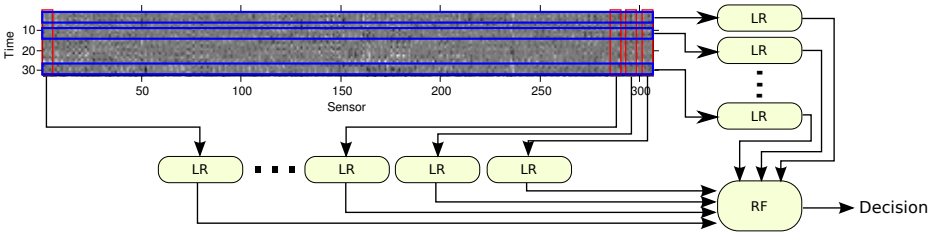

Fig. 2. Hierarchical prediction model.

for each $j = 1, 2, \ldots, 306$. Thus, each classifier will see one vertical slice of the data $\mathbf{X}[:, j, :]$ and maximize the likelihood of the model for the data. Similarly, the 31 sensor-dimension classifiers maximize the L_2 penalized log-likelihoods

$$\max_{\boldsymbol{\beta}, \beta_0} \left\{ \sum_{n=1}^{N} \mathbf{y}[n] \left(\beta_0 + \boldsymbol{\beta}^T \mathbf{X}[n, :, k] \right) - \log \left(1 + \exp \left(\beta_0 + \boldsymbol{\beta}^T \mathbf{X}[n, :, k] \right) \right) - \lambda \|\boldsymbol{\beta}\|_2^2 \right\}$$

for each $k = 1, 2, \ldots, 31$. Note that in both cases the target variable is the class label for the sample, which will be used as the target on the second layer, as well. This idea traces back to the *stacked generalizers* of Wolpert in 1992 [**?**].

After the first layer has been trained, the predicted class membership probabilities

$$\hat{\mathbf{y}}_j = F_j \left(\mathbf{X}[n, j, :] \right) \text{ and } \hat{\mathbf{z}}_k = G_k \left(\mathbf{X}[n, :, k] \right)$$

for $j = 1, 2, \ldots, 306$ and $k = 1, 2, \ldots, 31$ are calculated. The design matrix for the second layer is then constructed by catenating the predictions to a matrix:

$$\hat{\mathbf{Y}} = [\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_{306}, \hat{\mathbf{z}}_1, \hat{\mathbf{z}}_2, \ldots, \hat{\mathbf{z}}_{31}] \in \mathbf{R}^{N \times 337}.$$

The final step is then to train a random forest that predicts $\mathbf{y}$ from $\hat{\mathbf{Y}}$.

4 Iterative Training

A key challenge in MEG decoding is the high intersubject variance. For a single subject, it is easy to reach over 80% accuracy in cross validation tests, but the accuracy clearly decreases when testing the model with another subject. Therefore, is is necessary to transfer the trained model to adapt to the test subject, as well. These techniques are often called *transductive learning*, *semisupervised learning* or *transfer learning*.

Our approach for the transduction is to iterate training by augmenting training data with samples from the unlabeled test set. More specifically, on the 0'th iteration, we train a model with all training data, and use it for predicting labels for the unlabeled test samples. These labels are then used as ground truth on the 1'st iteration and the model is retrained with original training data and the

test samples with predicted labels. The iteration can be continued, as the model hopefully becomes more accurate in each iteration and the test labels become more and more correct.

The iterative approach typically improves the prediction accuracy by 2-3 %. In our experiments, we discovered that also the predicted probabilities are useful for screening the test samples for augmentation. It is expected that the samples with predicted probability far from 0.5 are probably more reliable. Therefore, we decided to include samples whose probability is over 0.6 (as faces) or below 0.4 (as scrambled faces).

We also investigated duplicating the augmented samples. Namely, there are altogether about 600 trials for each test subject, which is insignificant compared to the training data with almost 10 000 trials. Therefore, we studied different multipliers for duplicating each sample many times during augmentation. Increasing the weight of test samples seems to increase the accuracy, so we decided to discard the original training data completely in retraining (which is equivalent to infinite weight), and train with the test samples and predicted labels only.

As a final note, we realized that the test data should be collected from one test subject at a time. In predicting the labels for subject S, we train a model iteratively with training data and test samples from subject S only. Including samples from all subjects for augmentation produces clearly inferior results compared to this approach. This is quite natural to expect, as it is difficult to adjust the model to multiple subjects simultaneously due to the high intersubject variance.

5 Code Description

The implementation of our method is available at the repository `https://github.com/mahehu/decmeg`. The code is written in Python, and depends on the *scikit-learn, SciPy and NumPy* packages. The implementation consists of four Python files:

- `train.py`: The main training file. To train a model, execute: `python train.py`.
- `predict.py`: The file to predict the labels to test data. Execute by calling: `python predict.py`.
- `LrCollection.py`: A class containing the hierarchical model of Section 3.
- `IterativeTrainer.py`: A class containing the iterative training method of Section 4.

Moreover, there is a settings file `SETTINGS.json`, which defines the data path and the model path. Each function of the implementation is described in detail in Appendix A.

The implementation was tested on Linux Ubuntu 14.04 LTS, but should work on any platform, where the required packages are available. The code runs with 8 GB RAM, but probably also a smaller amount is enough.

6 Generating the Submission File

The model is trained by running `python predict.py`. Training assumes that the data is in the subdirectory specified in `SETTINGS.json` (key `"TRAIN_DATA_PATH"`, which defaults to `./data/`). This will create 7 models, one for each test subject. The models will be placed in the subdirectory specified in `SETTINGS.json` (key `"MODEL_PATH"`, which defaults to `./models/`). The models are serialized in Python *pickle* format as files `model17.pkl,...,model23.pkl`. Training takes about 20-30 minutes (3-5 minutes per test subject). The main body of this file defines additional parameters for the training. Among them is the parameter `estimateCvScore`, which toggles leave-one-subject-out cross-validation (LOSO-CV).

A submission file is created by running `python predict.py`. Prediction loads data from the subdirectory defined in `SETTINGS.json` (key `"TEST_DATA_PATH"`, which defaults to `./data/`) and prepares a submission file `submission.csv` to directory specified in `SETTINGS.json` (key `"SUBMISSION_PATH"`, which defaults to `./submissions/`). The prediction of 7 subjects takes about 30 seconds.

With the default parameters, the above procedure will train model with LOSO-CV score 0.7290 ± 0.0682, public leaderboard score 0.72732 and private score 0.71796. This is slightly different from our original submission due to different random seed.

7 Discussion

The method described was inspired by our initial experiments with convolutional neural networks (CNN). We discovered that they are effective but unstable, and minor changes in their parameters or initialization may change the accuracy a lot. Moreover, there was a large bias in the predictions: The predicted classes were not evenly distributed. In particular subject 20 was particular in this sense—up to 84% of the trials were predicted as faces. This imbalance was corrected by adjusting the selection threshold, but the results were still not very satisfactory.

During our experimentation with CNN's we realized that the snapshots of brain activity at one time instant (blue rectangles in Figure 3) were good and relatively stable predictors. We tried to reproduce the CNN behavior with various methods; including unsupervised ones (PCA, ICA, dictionary learning), but none of these were particularly successful.

Luckily one of the forum threads discusses stacked generalization. We decided to try this approach as well, and the results were good. Apart from the final model with logistic regression and random forest layers (LR-RF model), we tested other combinations, as well (other *scikit-learn* classifiers are still supported). To our slight surprise the L2 regularized LR was more accurate than the L1 regularized one.

The problem at hand was a difficult one. The best accuracies achieved were not nearly at the level of most Kaggle competitions. Even for MEG decoding

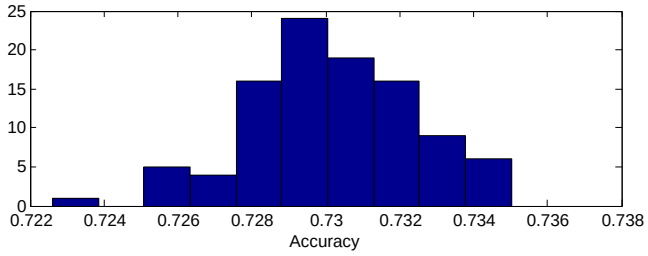

Fig. 3. LOSO-CV scores for the classifier with different random seeds.

problem the intersubject setup complicates the setup significantly. For comparison, our winning submission to the 2011 ICANN MEG Mind Reading competition[2] scored 68% for a 5-class problem, but this competition used within-subject data from two days [**?**,**?**,**?**].

As a result of this difficulty, most top submissions are within a fraction of a percent from each other. Our guiding principle was to maximize the stability of the model, such that the LOSO-CV and leaderboard scores are coherent with each other, and changes in parameters change the results in a consistent manner. Still, given more time and resources, there is still room for improvement in this respect. Figure 3 plots the LOSO-CV accuracy of the classifier when initialized with 100 different random seeds. It can be clearly seen that the distribution is wide and spans an interval of one percent. The variance could be decreased by adding trees to the random forest, which was not done due to computational reasons. However, this does not remove the uncertainty of predicting new test subjects.

[2] `http://www.cis.hut.fi/icann2011/mindreading.php`

A Description of the Code

A.1 File `train.py`

Function: `loadData`
Description: Load, downsample and normalize the data from one test subject.
Args:

- `filename`: input mat file name
- `downsample`: downsampling factor
- `start`: first time index in the result array (in samples)
- `stop`: last time index in the result array (in samples)

Returns:

- `X:` the 3-dimensional input data array
- `y:` class labels (None if not available)
- `ids:` the sample Id's of the samples, e.g., 17000, 17001, ... (None if not available)

Function: `run`
Description: Run training and save the model to disk.
Args:

- `datapath:` Directory where the training .mat files are located.
- `C:` Regularization parameter for logistic regression
- `numTrees:` Number of trees in random forest
- `downsample:` Downsampling factor in preprocessing
- `start:` First time index in the result array (in samples)
- `stop:` Last time index in the result array (in samples)
- `relabelThr:` Threshold for accepting predicted test samples in second iteration (only used if substitute = False)
- `relabelWeight:` Duplication factor of included test samples (only used if substitute = False)
- `substitute:` If True, original training samples are discarded on second training iteration. Otherwise test samples are appended to training data.
- `estimateCvScore:` If True, we do a full 16-fold CV for each training subject. Otherwise only final submission is created.

Returns:

- Nothing.

The main body of this file will call the function `run`.

A.2 File `predict.py`

Function: `run`
Description: Predict classes for test samples and prepare submission file.
Args:

- `datapath:` Directory where the .mat files are located.
- `downsample:` Downsampling factor in preprocessing
- `start:` First time index in the result array (in samples)
- `stop:` Last time index in the result array (in samples)

Returns:

- Nothing.

A.3 File `LrCollection.py`

The file defines a class called `LrCollection`, which holds containing the hierarchical model of Section 3.

Function: `LrCollection.__init__`
Description: Initialize class instance.
Args:

- `clf1:` First layer classifier. Can also be a list of classifiers.
- `clf2:` Second layer classifier. Can also be a list of classifiers. In this case, the output will be averaged over the predictions of the list.
- **useCols:** If true, train a predictor for each sensor
- **useRows:** If true, train a predictor for each timepoint

Returns:

- self

Function: `LrCollection.getView`
Description: Extract data from a single row or column in the data matrix. Can also span multiple rows/columns.
Args:

- `X:` Input data array of shape `(n, p, t)`
- `idx:` A three element list of requested slice coordinates. The element `idx[0]` is the list of trials to extract. Second element `idx[1]` is a two element vector with start and end indices in the sensor space (e.g., `idx[1] = [0, 306]`). The third element `idx[1]` is a two element vector with start and end indices in the time dimension (e.g., `idx[1] = [0, 31]`).

Returns:

– self

Function: `LrCollection.fit`
Description: Train the hierarchical classification model.
Args:

– `X`: Input data array of shape `(n, p, t)`
– `y`: Training class labels.

Returns:

– self

Function: `LrCollection.predict_proba_l1`
Description: Predict class probabilities for the test data using all 1st layer classifiers.
Args:

– `X`: Input test data array of shape `(n_t, p, t)`

Returns:

– Numpy array of probabilities (shape: `(n_t, num_classifiers)`)

Function: `LrCollection.predict_proba`
Description: Predict class probabilities for the test data. **Args:**

– `X`: Input test data array of shape `(n_t, p, t)`

Returns:

– Numpy array of probabilities (shape: `(n_t, num_classes)`)

Function: `LrCollection.predict`
Description: Predict class labels for the test data. **Args:**

– `X`: Input test data array of shape `(n_t, p, t)`

Returns:

– Numpy array of class labels (shape: `(n_t, )`)

A.4 File `IterativeTrainer.py`

The file defines a class called `IterativeTrainer`, which iterates the training by augmenting the data as described in Section 4.

Function: `IterativeTrainer.__init__`
Description: Initialize the IterativeTrainer. **Args:**

- `clf`: The classifier to iterate
- `iters`: Number of training/augmentation iterations
- `relabelWeight`: Duplication factor of included test samples (only used if substitute = False)
- `relabelThr`: Threshold for accepting predicted test samples in second iteration (only used if substitute = False) substitute: If True, original training samples are discarded on second training iteration. Otherwise test samples are appended to training data.

Returns:

- self

Function: `IterativeTrainer.fit`
Description: Train a classifier using the iterative semisupervised approach. **Args:**

- `X`: Input data array of shape `(n, p, t)`
- `y`: Training class labels (shape `(n,)`)
- `X_test`: Test samples (shape `(n_t, p, t)`)

Returns:

- self

Function: `IterativeTrainer.predict_proba`
Description: Predict class probabilities for the test data. **Args:**

- `X`: Input test data array of shape `(n_t, p, t)`

Returns:

- Numpy array of probabilities (shape: `(n_t, num_classes)`)

Function: `IterativeTrainer.predict`
Description: Predict class labels for the test data. **Args:**

- `X`: Input test data array of shape `(n_t, p, t)`

Returns:

- Numpy array of class labels (shape: `(n_t, )`)
