Neural Networks and Ensemble Methods in Eye-State Classification

FYS-STK4155: Project 3

Ghadi Souheil Al Hajj, Maria Nareklishvili & Ulrik Seip
github.com/MariaRevili/FYS-STK4155

December 11, 2020

Abstract

This paper utilizes ensemble methods, neural networks and their combinations to predict eye-state using electroencephalography (EEG) electrodes (consisting of 14 features). In particular, we apply random forests, neural networks, boosting, combination of random forests and boosting with neural networks and compare each. We contribute the existing influential literature by learning predictions from stumps with the architecture of neural networks. The standard prediction performance measures, the ROC curve and the accuracy, show that the random forest algorithm exhibits the best prediction performance (94.8%) and is intuitive to apply. However, a remarkable performance improvement is found when combining boosted stump predictions into neural networks (91.1% improvement of the accuracy relative to the conventioonal AdaBoost algorithm with the accuracy of 77%).

I. INTRODUCTION

Software applications usually require some form of input from the user to determine the operations performed by such applications. Whether these applications are games, interactive interfaces or medical software, the user provides input through different types of devices, such as a mouse, a keyboard, a joystick, voice-command, etc. As technological advances provide us with new tools each day, we can explore new ways of interacting with software.

Using the electroencephalography, EEG, data is one of the possible options that can be exploited in brain-machine interface in order to develop novel utility applications. This can be particularly helpful for people who cannot control objects using direct touch interaction. For example, EEG data can be used to devise new control systems that operate in a touch-less manner, such as using a personal computer.

Moreover, EEG data can be analyzed to infer certain physical aspects of the human body, such as predicting the state of the human eye, whether it is open or closed. This could later be used for manipulation of certain devices through some well-defined mechanisms. The EEG data represent numerically encoded values indicating different activity levels at various regions of the brain. The outcome variable, the eye-state, to be predicted is readily available in a categorical format. This makes the analysis particularly suitable through machine learning methods.

This article exploits neural networks, ensemble methods (i.e. random forests and boosting) and their combination to predict the eye state (i.e. the eye being open). One of the major contributions of the article is to learn weak trees (i.e. stumps) through the layers of neural networks. The idea is fairly simple; predictions of each weak tree on the train data are combined together to form a matrix of new features. The new feature matrix is then used as an input matrix for neural networks for learning. This is advantageous for several reasons; weak learners carry information that can be more efficiently extracted by the architecture of neural networks in high dimensional settings with highly correlated features. In these settings the primary aim of the algorithm is to sparsify the space of features and shrink the variation that can stem from the covariates uncorrelated with the outcome variable. Even though the EEG data is not high dimensional (i.e. $n > p$) it can give useful improvement if the electroencephalography electrodes are characterized with correlated structures.

The analysis in this paper verifies that the random forest algorithm performs superior relative to the others (with 94.8% accuracy and 99% *area under the curve (AUC)*). However, the margin of the improvement is not particularly high; neural networks and learning ensemble methods through multiple layers lead to a similar accuracy (92%-94%). A remarkably interesting finding in this article is achieved by comparing the conventional boosting algorithm (using weak learners, i.e. stumps) with the combination of boosted tree predictions with neural networks. The improvement becomes sufficiently noticeable over the standard boosting scheme (i.e. 91.1% relative to the boosting accuracy of 77%). This indicates that highly weak predictions from stumps encode information that is easily learned by neural networks, but are not well reflected in the standard boosting algorithm.

The rest of the report is split into sections designed to raise subjects for discussion and lead the reader to conclusions similar to those presented. In section II, we describe the methods that we employ in the paper, section III overviews the data at hand. In section IV we present the results and finally, sections V and VI are dedicated to summarize and conclude the paper.

II. CLASSIFICATION METHODS

A. Neural Networks

Neural networks are methods that are specifically tailored to learn from the given data in a nonlinear way. The method works particularly well in a high-dimensional high-complex structures. Neural networks were in-depth described in Project 2 by Souheil Al Hajj et al. (2020), however, we will provide the essence and remind the notation. For a more detailed analysis see Souheil Al Hajj et al. (2020).

From the rich and diverse types of neural networks family, in this article we focus on Multilayer Perceptron (MLP) neural networks. The MLP neural networks are built from *layers* of connected *neurons*. In the artificial network, an input value (possibly a vector) is fed into the network model and then propagated through the layers, being processed through each neuron in turn. We will deal only with *feed forward* ANNs, meaning information always flows through the net in one direction only—essentially there are no loops. The entire ANN produces an output value (possibly a vector), which means we can think of it as a complicated function $\mathbb{R}^n \mapsto \mathbb{R}^m$.

A neuron is simply a mathematical function for propagating information through the network. Inspired by biological neurons, the artificial neuron "activates" if it is stimulated by a sufficiently strong signal. The artificial neuron receives a vector of input values $\mathbf{x}$. If the neuron is part of the very first hidden layer (denoted by l which is simply a connection of neurons), the input is simply the input (feature) value(s) to the NN. If one or more layers preceded the current one, $\mathbf{z^l}$ is a vector of outputs from the neurons in the previous layer.

The neuron is connected to the previous layers' neurons, and the strength of the connection is represented by a vector of weights, $\mathbf{w}$. Let us now consider a neuron which we will label by the index k. The output from neuron i (of the preceding layer), z_i, is multiplied by the weight corresponding to the i—k connection, w_i. The combined weight vector multiplied by the input vector gives part of the total activation of the neuron,

$$\begin{aligned}
\text{First layer: } & \sum_{i=1}^{N^1} w_i x_i = \mathbf{w}^T\mathbf{x} = z^1 \qquad (1)\\
\text{Second layer: } & \sum_{i=1}^{N^2} w_i z_i^1 = \mathbf{w}^T\mathbf{z^1} = z^2\\
& \vdots\\
\text{Output layer: } & \sum_{i=1}^{N^{l-1}} w_i z_i^{l-1} = \mathbf{w}^T\mathbf{z^{l-1}} = a
\end{aligned}$$

Note that here a is the final predicted output of a particular neuron. This notation is introduced for the convenience. Also the number of neurons can differ in each subsequent layer, therefore, the total number of neurons —N^l is indexed by a corresponding layer l. The remaining part is known as the bias, b_k. This is a single real number. There is one for each neuron, and it acts as modifier making the neuron more or less likely to activate independently of the input.

The total input is passed to an activation (or transfer) function, which transforms it in some specified way, yielding the neuron *output* $\hat{z}_k$. This in turn becomes input for the neurons in a subsequent layer.

Various different activation functions f are used for different purposes. The function may be linear or non-linear, but should vanish for small inputs and *saturate* for large inputs. Numerous different transfer functions are in popular use today, such as, sigmoid, tanh, rectifiers (ReLU, Elu, Selu etc.) (for a more detailed description of these functions see Souheil Al Hajj et al. (2020)).

In total, the action of a single neuron k in layer l can be written as follows:

$$\text{input } \rightarrow f\left(\mathbf{w}^T\mathbf{z} + b\right) = z_k^l \rightarrow \text{ output.} \quad (2)$$

Figure 1 depicts a neural network architecture consisting of $n \times p$ matrix of features. The sum of these features multiplied by the weights and bias are sent into the second layer. The process can continue up to infinity, until the final output layer prediction. The final output is denoted by a. The weights are updated by the backpropagation algorithm (see Souheil Al Hajj et al. (2020)).

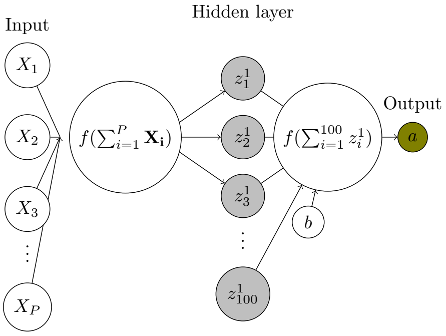

FIG. 1. Neural Networks

B. Decision Trees

To compare the performance of neural networks with well-known ensemble methods, we start out by describing simple trees. Tree-based methods partition the feature space into a set of rectangles, and then fit a simple model (like a constant) in each one. These methods are conceptually simple yet powerful. Growing a simple tree requires recursive binary partitions of the feature space. We first split the space into two regions and evaluate the average value of the outcome variable. Then one or both regions are further split into two or more regions, and this process continues until some stopping criterion is applied.

Regression Tree

The given data consists of p input variables and a target variable. The pair $(\mathbf{x}_i, y_i)$ represents a particular row in a data set for each item $i = 1, ..., N$, with $x_i = (x_{i,1}, x_{i,2}, ..., x_{i,p})$. The primary task is to split the region and compute a constant in these regions. Starting with the full data, consider a splitting variable j and the split point s. Then define the pair of half-planes:

$$R_1(j,s) = \{X|X_j \leq s\} \text{ and}$$
$$R_2(j,s) = \{X|X_j \leq s\}$$

Finding the best binary partition in terms of minimum sum of squares is computationally infeasible. The procedure therefore "boils down" to a greedy algorithm. We seek the pair of splitting variable and split point (j, s) that solves the following minimization problem:

$$\min_{j,s}\Bigg[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 \quad (3)$$
$$+ \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2\Bigg]$$

where c_1 and c_2 are constants to be determined from the inner minimization. Clearly, solving for the inner minimzation verifies that $c_1 = \text{ave}\left(y_i|x_i \in R_1\left(j,s\right)\right)$ and $c_2 = \text{ave}\left(y_i|x_i \in R_2\left(j,s\right)\right)$.

Based on the described procedure, we partition the region of the data space into two parts. The splitting process continues then on each of those two regions and subsequently onward on the resulting sub-regions using the same mantra (see Tibshirani (1996)).

To intuitively describe the idea, imagine the full space of the data can be fit into the area of the rectangle. Assume the height is some vector of outcome variable $\mathbf{y}$ and the length and width are represented by two variables: $\mathbf{x_1}$ and $\mathbf{x_2}$. A 3-dimensional example is given on Figure 2 [1].

[1] *Source:* https://en.wikipedia.org/wiki/K-d_tree

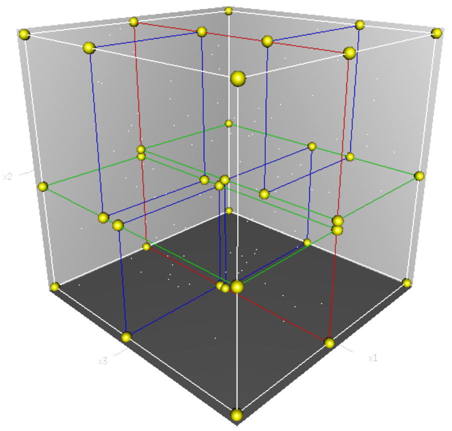

FIG. 2. The process of partitioning of the data space.

The cube can be partitioned first into two smaller 'sub-cubes'. If we do not define the stopping criterion these cubes will be partitioned fully, until we end up with one single observation in each cube. [2] For each partition, we end up with the average value of the outcome variable. That can be illustrated with a simple example in Figure 3[3]:

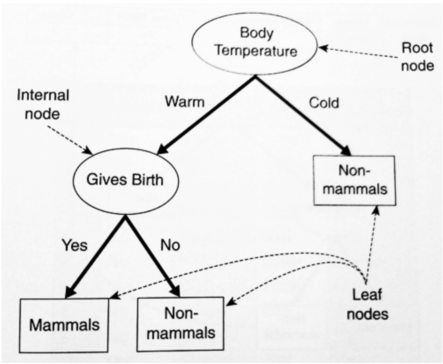

FIG. 3. The decision tree.

The starting variable (in Figure 3 Body Temperature) is called the *root node*. Then the branches (given by arrows) lead to the subsequent nodes, a.k.a *internal nodes*. Finally, the end nodes are called *leaf nodes*. The size of the tree is a tuning parameter and it should adaptively be chosen from the data by minimizing the *cost-complexity criterion*. In the regression setup this criterion is given as follows:

$$C_\alpha(T) = \sum_{m=1}^{T} N_m Q_m(t) + \alpha|T| \tag{4}$$

where $C_\alpha(T)$ denotes the cost function. N_m represents the number of observations in region m. $Q_m(t) = \frac{1}{N_m}\sum_{x_i \in R_m}(y_i - ave(\mathbf{y}))$ is the squared loss for each region. T is any sub-tree that can be obtained by pruning some tree T_0. The tuning parameter α governs the tradeoff between tree size and its goodness of fit to the data. Increasing values of α result in smaller trees, and conversely for decreasing values of α.

Classification Trees

When the outcome of interest consists of classes taking values $1, 2, ...K$, the only change needed for constructing the tree is the criterion for splitting nodes and pruning the tree. In node m of region R_m with N_m observations, define the probability of class k.

$$\hat{p_{mk}} = \sum_{x \in R_m} I(y_i = k) \tag{5}$$

where $I(y_i = k)$ is the binary indicator for a class k. Then the uppermost goal is to maximize the probability of observing class k, i.e. $k(m) = \text{argmax}_k \hat{p_{mk}}$. Instead of the cross-complexity criterion, we introduce the node *impurity measure.*

Misclassification error: (6)

$$\left(\frac{1}{N_m}\sum_{i \in R_m} I(y_i \neq k(m))\right) = 1 - \hat{p}_{mk(m)}$$

Gini index : $\sum_{k \neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k \neq k'} \hat{p}_{mk}(1 - \hat{p}_{mk})$

Cross entropy: $-\sum_{k=1}^{K} \hat{p}_{mk}\log(\hat{p}_{mk})$ (7)

Node impurity measures inform about how much additional information each feature can add to the tree. Unlike misclassification error, cross-entropy and Gini index are differentiable in their domain, hence more amenable to numerical optimization. Gini index is simply the sum of variances (i.e. $\hat{p}_{mk}(1 - \hat{p}_{mk})$) for each existing class k. Intuitively, when the impurity indices increase by adding new features to the split, that indicates that the variability on the training error increases. That by itself entails

[2] Note that in this case random forests will be analogous to 1-nearest-neighbor approach.

[3] *Source:* http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/sdaugherty/decisiontree.html

disregarding the variables that increase the error.

In this paper we only utilize classification setup and grow the trees using the Gini index in sci-kit learn library.

Several notable issues mark the simple tree:

- **Overfitting**: In decision trees, over-fitting occurs when the tree is designed so as to perfectly fit all samples in the training data set. Thus it ends up with branches with strict rules of sparse data. This effects the accuracy when predicting samples that are not part of the training set.
- **Instability of the trees:** Trees are characterized with high variance because of their hierarchical nature. Often a small change in the data will induce very different series of splits.
- **Difficulty in capturing the additive structure:** Since the procedure is recursive (and plausibly binary), there is no guarantee that the tree will capture the additive structure, even with sufficient data.

To overcome the issues mentioned above (and also increase the prediction performance) we introduce bagging and random forests in the subsequent sections.

C. Bagging

Intuitively, one would argue that instead of using one tree, we can build more than one tree and use the collective vote of all the trees as a prediction. This follows from the law of large numbers, which states that as the sample size (the number of trees, in this case) increases, we get closer to the real value of the quantity of interest. This approach is exactly what bagging, short for bootstrap aggregating, does.

In this method, we perform several bootstraps of the original data set, and we build a new tree each time. Importantly, each tree is built independently of the other trees in the sense that the performance of one tree does not affect how other trees are built. Additionally, trees are not constrained to shallow depths, as compared to the AdaBoost algorithm (to be discussed later in section F) while, of course, taking over-fitting into consideration. Upon classifying a given input example, each tree gets an equal say in the classification process, i.e. democratic voting. The example is classified by the majority vote according to the following equation:

$$\tilde{y_i} = \arg\max_c (n_1, .., n_c, .., n_C) \tag{8}$$

where $\hat{y_i}$ is the predicted class of the i^{th} example, n_c is the number of trees that have voted for the c^{th} class, and C is the total number of classes.

In the binary case, this becomes:

$$\tilde{y_i} = \delta(n_1 > n_0) \tag{9}$$

In other words, a new test example is assigned the class that gets the majority of the votes from the trees.

D. Random Forests

One of the major problems with bagging is the positive correlation of each pair of tree. This in turn limits the benefits of aggregating them. The issue can be overcome by introducing randomness with respect to the choice of the predictor variables. We outline the algorithm for growing random forests in the following steps:

```
1) For b = 1 to B:
    a) Draw a bootstrap sample $\mathbf{Z}^*$ of size $N$ from the training data.
    b) Grow a tree on each bootstrapped sample recursively by repeating the following steps:
        * Select m variables at random from the $p$ features.
        * Pick the best split-point among the chosen $m$.
        * Split the node into two subsequent (i.e. daughter) nodes.
2) Output the ensemble of trees
```

Simply put, random forests is a collection of trees just like bagging, but the tree is grown by randomly choosing m variables at each split. A collection of such trees creates a "forest". Visually a forest is depicted in Figure 4. Aggregating these "weak learners" is motivated by the majority aggregating rule. That is, a combination of weak judgements can lead to a jointly improved judgement. [4]

[4] Note that trees can be grown up to infinity, i.e. with no pre-defined number of leaf nodes (this is a default choice in scikit-learn library).

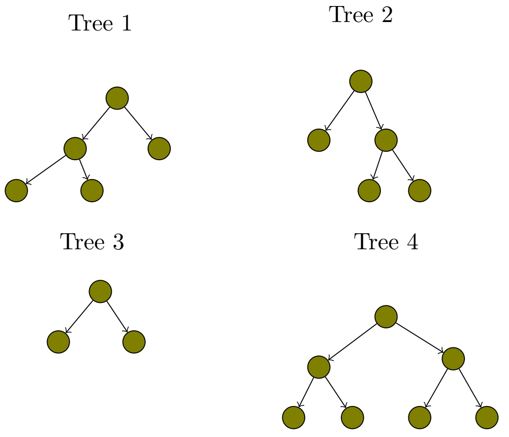

FIG. 4. Random Forest with four trees.

E. Deep Forest Learning

Following Kong and Yu (2018), this paper, in addition to the aforementioned methods, presents a combination of random forests with neural networks. First, we fit random forest trees on the train data and predict the output of each weak learner on both, train and test data sets. Then we “feed” the matrix of these outputs into neural networks instead of the original covariates. This approach is known as *stacking*. Stacking has been actively used in the field of machine learning (see e.g. Sridhar et al. (1996)). However, learning weak predictions by the neural network architecture is preferable for several reasons: first, sparsifying the data and disregarding the redundant information (useful in high-dimensional data). Second, the approach can improve the prediction accuracy. The algorithm is outlined as follows:

1. Fit random forest trees on the train data set and predict output of each tree on both train and test data sets (denoted as $\mathbf{T} = \left[\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_B}\right]$ where B is the number of trees).

2. Use predicted outcome matrix T as an input for the first layer l^1 in Neural networks instead of the original feature matrix X.

2. Compute the prediction performance on the test data.

Figure 5 graphically illustrates the deep forest learning method. The sum of the predictions from the trees is transferred to the hidden layer and the neural network framework outputs the final prediction.

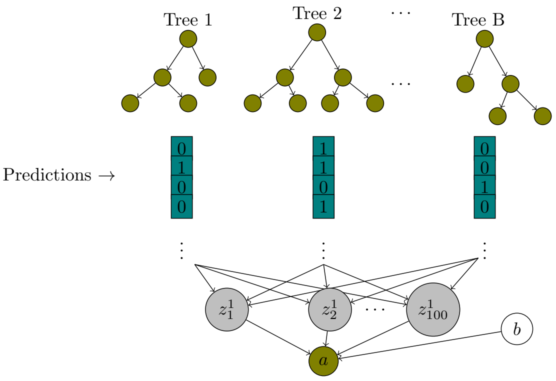

FIG. 5. Visual exposition of the deep forest learning method

F. AdaBoost

Instead of following a simple voting scheme, as the one used in bagging or random forests, a boosting method called AdaBoost approaches this problem differently. This approach has several differences from the straightforward voting scheme. Specifically, in AdaBoost each tree gets a different say in the final voting depending on its performance, and thus some trees may contribute more to the final decision than other trees. More importantly, instead of building trees independently of each other, as in bagging, in AdaBoost trees are built in relation to the previous trees, so the order in which the trees are built is very important. Concretely, the error that one tree makes, influences how the next tree is build in the sense that the new tree strives to overcome the shortcommings of the previous trees. Additionally, classifiers are usually weak classifiers called stumps, which are basically trees with one root node and two leaves each. The last point implies that instead of using many features, whether a random subset of them as in random forests or all of them as with standard decision trees, in order to build multistaged trees, only one feature is used to build a given stump.

Since order is important, boosting is technically an iterative algorithm that starts with one weak classifier built on the original data set and adds a new hypothesis at each iteration.

Converse to bagging, AdaBoost does not use the same data set to build different stumps. Rather, at each iteration it builds a new data set so as to over-represent the examples that were misclassified in the previous iteration. Technically speaking, the algorithm starts with first assigning an equal weight to each example. The weight of any example in the first iteration, $w_{i,0}$, is equal to:

$$w_{i,0} = \frac{1}{n},\ for\ i = 0, 1, ..., n-1 \tag{10}$$

where n is the total number of samples in the training set.

After that, we build the best weak classifier on the data set. To do that, different stumps are built, each using one of the available features, and the one with the best classification performance, measured using an appropriate index, is selected as the stump for that iteration. Then, the error that a stump makes at a the m^{th} iteration is calculated as:

$$\varepsilon_m = \frac{\sum_{y\neq\hat{y}} w_{i,m}}{\sum_{i=0}^{n-1} w_{i,m}} \tag{11}$$

The error is simply the sum of weights associated the incorrectly classified examples at the m^{th} iteration.

This error, in turn, is used to calculate the importance of that stump's vote at the final voting stage. Intuitively, the higher the error a tree makes, the less important that tree's vote would be at the final stage, and vice versa. The "say" of the stump built at the m^{th} iteration is calculated using the following equation:

$$\beta_m = \frac{1}{2}\ ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right) \tag{12}$$

Figure 6 shows how the importance of the vote of a given stump, β, changes with respect to the value of the error made by that stump, ε. At low errors, the stump gets a high value of β, thereby, increasing its share in the final say on a new example. On the other hand, for high error rates, the stump's influence on the final say decreases. Alternatively, if the stump is more or less a random classifier, it gets a importance of 0.5

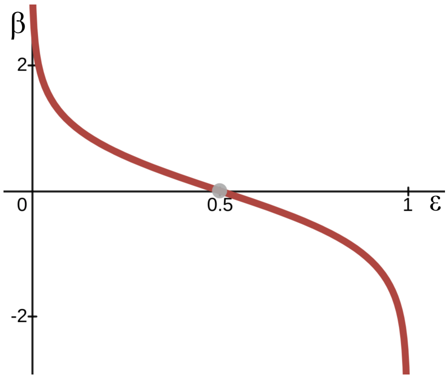

FIG. 6. The importance of vote, β, in terms of the error made by the stump, ε. *Source: generated using* https://www.desmos.com/calculator

As mentioned earlier, after each iteration, we build a new data set that emphasizes the examples that were incorrectly classified in the previous iteration. This ensures that the next stump will pay more attention to the incorrectly classified examples from the previous iteration. Before we can do that, the weight of each sample is first adjusted depending on whether that sample was correctly classified in the previous iteration. If the sample was correctly classified in the previous iteration, its weight gets updated by the following equation:

$$w_{i,m+1} = w_{i,m} * e^{-\beta_\mathrm{m}} \tag{13}$$

Otherwise, the weight gets updated as:

$$w_{i,m+1} = w_{i,m} * e^{\beta_\mathrm{m}} \tag{14}$$

These two equations ensure that the incorrectly classified samples will get a higher weight in the next iteration, as their weight is multiplied by a factor of e^{β_m}, whereas those at were correctly classified will get a lower weight, as their weight is multiplied by a factor of $e^{-\beta_m}$. One important step to take before repeating the same process again is to normalize the new weights to sum to one, using the following equation:

$$w^*_{i,m+1} = \frac{w_{i,m+1}}{\sum_{i=0}^{n-1} w_{i,m+1}} \tag{15}$$

Now, in order to build a new data set, we use a random number generator that outputs a number from a uniform distribution $\sim U(0,1)$, and then select the sample, which the number falls into its range. This range is equal to the difference of a sample's weight and the weight of the sample preceding it. Intuitively, the weight of a highly weighted sample will be farther away from the other weights. For example, if three samples have weights 0.1, 0.7, and 0.2, then the difference between the second weight and the previous one is the largest weight difference, $0.7 - 0.1 = 0.6$, thus giving the second weight a larger distance from the preceding sample's weight. Therefore, a random number generated from the previously mentioned distribution is more likely to fall within the range between 0.1 and 0.7, than between 0 and 0.1, or 0.7 and 1. Therefore the higher weighted samples will get a higher chance of being included in the next data set. This process is repeated until we build a new data set of the same size of the original set, i.e. n.

The whole process of building a new data set, selecting the best stump, and updating and normalizing the weights, is repeated until a predetermined number, M, of iterations is performed.

Finally, when it comes to classifying a new test sample, the class is predicted using the following equation:

$$\tilde{y}_i = \delta\Big(\sum_{\tilde{y}_{i,m}=1} \beta_\mathrm{m} > \sum_{\tilde{y}_{i,m}=0} \beta_\mathrm{m}\Big) \tag{16}$$

In plain words, this compares the sum of the says, β, of the stumps that predicted class "1" to the sum of says of the stumps that predicted class 0. Then, the final vote would go the class that got the higher sum of votes.

G. Boosting in Neural Networks

To make a parallel with the deep forest learning (DFL) method, we combine stumps learned by boosting using neural networks. The idea is essentially the same, however instead of fully grown trees, we use predictions from boosting as the input matrix into the framework of neural networks.

Figure 7 conveys the idea in its simplest form. B is the number of grown stumps. The predictions from fitted boosted trees on the train data are sent to the hidden layer with a given number of neurons. Neural networks then output the prediction from these weak trees on the test data, denoted as a.

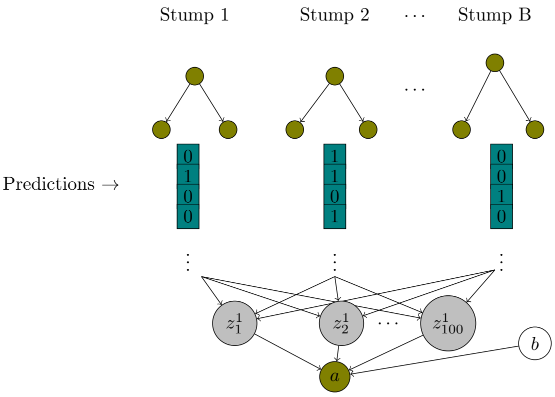

FIG. 7. Boosting in Neural Networks. B is the number of stumps (weak learners), b is the bias for the first layer and a stands for the output.

H. Performance

Before presenting the data and the prediction results, we define the standard measures (i.e. Receiver Operating Characteristic (ROC) and the accuracy) used to evaluate the performance of each method. In its simplest form, they both describe the rate of correct predictions. As elaborated on in Project 2 by Souheil Al Hajj et al. (2020) the accuracy can be expressed as follows:

$$\text{Accuracy} = \frac{\#\text{ Correctly classified data points}}{\#\text{ All data points}}$$

Which in our case can further be specified as:

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}} = \frac{\text{TP+TN}}{\text{N}}$$

where TP and TN represent true positives and negative prediction counts (i.e. here the count of truly predicted 1s (eye being open) and 0s (eye being closed). By comparison, FP and FN represent counts for false positive and false negative predictions. N is simply the total number of predictions.

Accuracy is a standard measure for making an initial assessment of a model, but it is not sufficient to fully assess the quality of the method at hand. Imagine a data set with 90% positives and 10% negatives. If the model predicted all data cases as positives it would be 90% accurate, which at first glance seems like a good score. To obtain a better understanding of our models performance we introduce the receiver operating characteristic curve (ROC).

The ROC of a model is a graphic representation of the models performance, comparing the False Positive Rate (FPR) to the True Positive Rate (TPR), as can be seen in figures 9 through 13.

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}} \tag{17}$$

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{N}} \tag{18}$$

A useful quantity related to the ROC is the area under the curve (AUC). If the ROCs AUC is 1, that means we have a perfect model, with 100% correct predictions, regardless of how strict we are with the false positive rate. As the AUC approaches 0.5 the model approaches complete randomness, and if the AUC decreases further it suggests that our model is wrong more often than not.

III. DATA

The data set used in this project is a set of 14 different measurement regions in the human brain, captured using electroencephalography (EEG), along with their corresponding eye-state. This data represents the brain activity at each of these regions. Figure 8 shows these regions, designated as AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, and AF4. Specifically, Fp stands for Frontopolar, F for Frontal, C for Central, T for Temporal, P for Parietal, and O for Occipital, with AF representing the region between Fp and F, and FC representing the region between F and C. As for the numbers, even numbers denote the right hemisphere locations, while odd numbers denote the left hemisphere locations.

The data set contains a total of 14977 instances, in their corresponding chronological order, from a single EEG recording that is around 117 seconds long. The eye-state was recorded via a video camera shooting at 32 frames per second, along with an analog to digital converter (ADC) sampling at four times the frame rate. An "open" or "partially open" eye was encoded as "1", whereas only a completely closed eye was encoded as 0. Both classes are represented almost equally with 55.12% of the instances belonging to the positive class, 1, and 44.88% belonging to the negative class, 0.

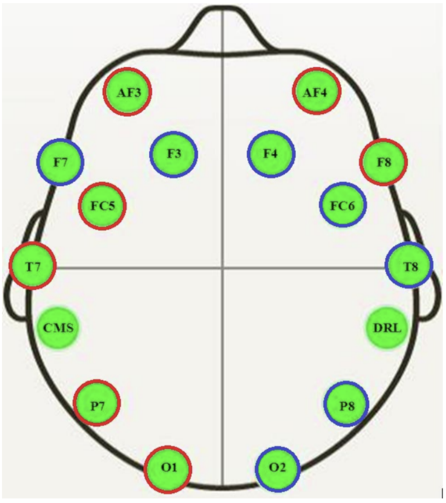

FIG. 8. The placement of the different EEG electrodes on the human skull. *Source:* Roesler and Suendermann (2013)

To get a better idea about the distribution of the features, Table I summarizes the mean, the minimum, and the maximum value of each feature for, both, the open and closed eye status.

TABLE I. The mean and the range of each feature for each of the eye states. *Source:* Roesler and Suendermann (2013)

Eye	closed			open		
State	min	mean	max	min	mean	max
AF3	4198	4305	4445	1030	4297	4504
F7	3905	4005	4138	3924	4013	7804
F3	4212	4265	4367	4197	4263	5762
FC5	4058	4121	4214	2453	4123	4250
T7	4309	4341	4435	2089	4341	4463
P7	4574	4618	4708	2768	4620	4756
O1	4026	4073	4167	3581	4071	4178
O2	4567	4616	4695	4567	4615	7264
P8	4147	4202	4287	4152	4200	4586
T8	4174	4233	4323	4152	4229	6674
FC6	4130	4204	4319	4100	4200	5170
F4	4225	4281	4368	4201	4277	7002
F8	4510	4610	4811	86	4601	4833
AF4	4246	4367	4552	1366	4356	4573

The implementation of the methods in this paper is carried out in scikit-learn and tensorflow libraries using Python.

IV. RESULTS

I. Neural Networks

The complex architecture of neural networks requires tuning the parameters in order to achieve the desirable prediction performance. We tune learning rate, gradient descent decay, hidden layers and the batch size. The process was in-depth analyzed in Project 2 by Souheil Al Hajj et al. (2020).

We build neural networks using 1 hidden layer and ReLU activation function. We use ReLU as it successfully overcomes the saturation problem (see e.g. Yu et al. (2001)). The hidden layer is defined by 100 neurons. As a gradient descent method, we employ Adam with 1200 epochs. Adam leads to the improved accuracy relative to the standard SGD algorithm.

Neural networks yield the 92.9% accuracy on the test data. That means, EEG electrodes explain 92.9% of the eye-state (i.e. closed or open) based on neural networks. High reflection of features into the outcome variable is perhaps not surprising as the brain signalling largely determines the state of the eye.

Figure 9 shows the corresponding ROC curve. The method as seen from the Figure captures 98% of the area under the curve (AUC). That indicates that the false positive rates (i.e. "false alarm") is relatively small.

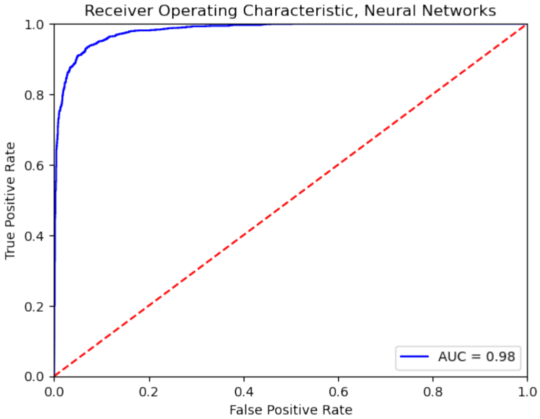

FIG. 9. ROC for Neural Networks. Learning rate (η) is 0.15 and the gradient descent decay (λ) is 0.

J. Random Forests

Neural networks capture most of the relevant information in the data, however because of its complex structure, it might not necessarily be the most optimal choice. Ensemble methods are one of the most common methods applied in prediction settings. We start out by describing the results using random forests in this section.

We build 500 trees on the train data and test the prediction performance on the test data. The accuracy of random forests is 94.8%. The method performs slightly better than neural networks. Figure 10 depicts the ROC for random forests. AUC is 99%, indicating that the method has particularly low false positive rates.

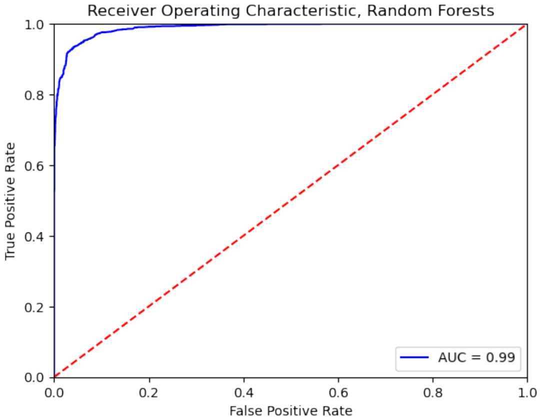

FIG. 10. ROC for Random Forests. Random forests consist of 500 trees on the original data without bootstrapping.

The size of the tree (i.e the number of end node leaves) is not restricted in this paper.

K. Deep Forest Learning

As discussed, in certain settings combination of random forests with neural networks yields a more robust estimation results and an improved accuracy rate (see Kong and Yu (2018)).

The accuracy of deep forest learning on the eye-state data is 0.93%. This is not much different either from random forests or neural networks. Figure 11 depicts the ROC curve for the deep forest learning. The area captured under the curve is 98%. The prediction performance is the same as in neural networks.

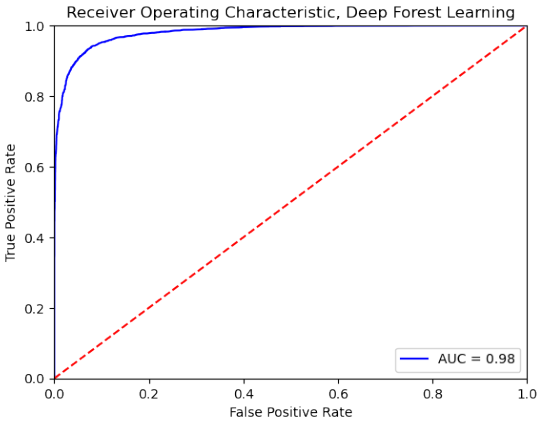

FIG. 11. ROC for the Deep Forest Learning. The forest includes 500 trees. Neural networks consists of 1 layer, 100 neurons, ReLU activation function in the first layer, 500 epochs with 100 batch size. Learning rate is 1 and the gradient descent decay is 0.003.

In the eye-state data classification, the combination of random forests with neural networks does not improve upon greatly over the other methods. Various reasons can underline this result; trees grown in random forests do not have a predefined number of leaves in the end nodes (i.e. we do not restrict the length or the width of the tree, the accuracy of each predicted tree is roughly 40%). Therefore, prediction of each fully grown tree entails sufficient information and this can be accurately learned from the forest aggregating rule itself. If we reduce the length of the tree the accuracy of the prediction by each separate tree decreases. In that case combining weak learner predictions into neural networks yields a higher performance improvement.

The second reason is the lack of correlation of the features and low dimension. This method has been proven to be fairly well-behaved in high dimensional data (i.e. when $p \geq n$). However, the data set at hand includes only 14 features. Hence, there is no substantial need to reduce dimensionality or extract the relevant features.

L. Boosting

Comparing the results of boosting to those of any of the previously discussed methods, AdaBoost performs poorly even with the same number of trees, although in this method, trees are weak learners, particularly stupms, instead of fully grown trees. Figure 12 shows the results of AdaBoost with an AUC score equal to 0.85 with an accuracy of 0.77.

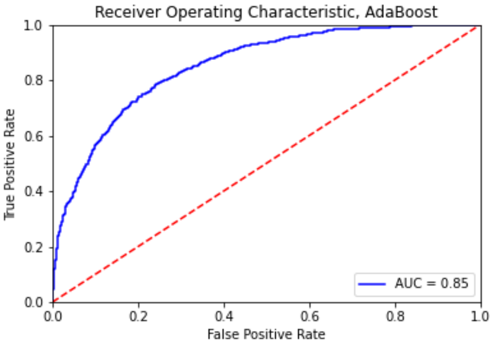

FIG. 12. ROC for the AdaBoost method. The Boosting process built 500 stumps.

Not surprisingly, an AdaBoost model with trees allowed to grow to a maximum depth of, say, three, would result in a better performance, as can be seen in figure 13. Under these settings, the model achieves an AUC score of 0.97 and an accuracy of 0.92.

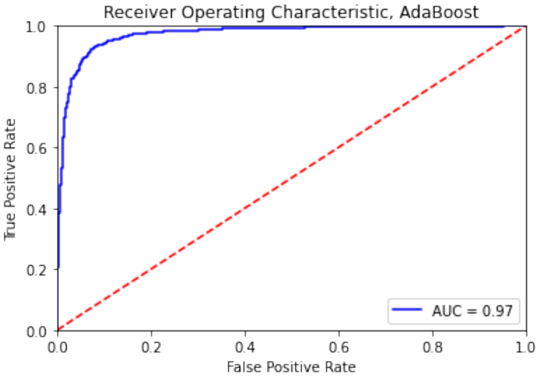

FIG. 13. ROC for the AdaBoost method with trees of maximum depth equal to three. Again, the Boosting process used 500 trees.

M. Boosting in Neural Networks

In the previous section we provided the results from the deep forest learning method. We saw that the improvement was not much evident in this setting. To make a parallel, we combine fitted stumps through boosting into neural networks and dedicate this section to the results.

The accuracy from combining the boosted stump predictions improves substantially over the boosting method solely. Boosting as seen predicts the eye-state on the test data with 77% accuracy, however predictions from each separate weak tree learned by neural networks exhibit 91.1% accuracy. This improvement over the standard AdaBoost with stumps can be seen by comparing Figure 12 with Figure 14. The area covered by boosting in neural networks is roughly 97%.

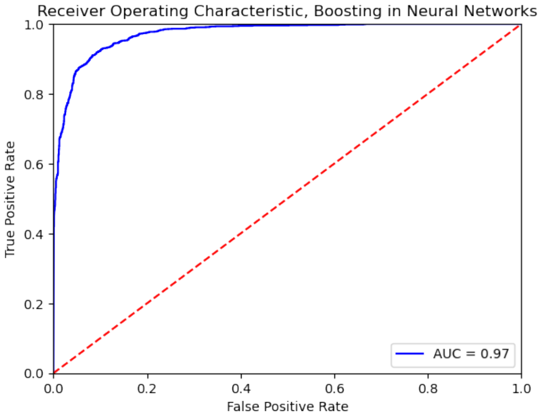

FIG. 14. ROC for Boosting in Neural Networks. The Boosting process used 500 stumps (weak trees). The learning rate is 1 and the gradient descent decay is 1e-5. For the details regarding other parameters, see section I.

V. SUMMARY OF THE RESULTS

Table II shows that random forests, neural networks and deep forest learning exhibit similar prediction performance (i.e. accuracy) in the range of 92% - 95%. Boosting with weak trees of only two leaf nodes (i.e. stumps) performs worst of all methods (77% accuracy on the test data). However, a remarkably improved result is achieved when combining predictions of each weak boosted tree with neural networks. Boosting predictions learned with neural networks achieves 91.1% accuracy.

Deep forest learning and boosting in neural networks do not achieve much improvement over the standard neural networks or random forests with fully grown trees. However, the prediction performance improvement is notably large when we are given with vague and completely uninformative trees, such as predictions from each stump, or bootstrapped trees with restricted number of leaf nodes.

In a nutshell, the choice of the method depends on the applications at hand. In this setting it is easiest to use random forests. If the dimension or the correlation of features increases, then we are better off using the combination of ensemble methods with neural networks.

VI. CONCLUSION

In this paper we used neural networks, random forests, boosting, deep forest learning and boosting in neural networks to predict eye-state movements with electroencephalography (EEG) electrodes. The ROC curve and the accuracy represent prediction performance measures for each method.

The results in the paper indicate that random forests is the easiest and the most efficient choice in this specific setting (with 94.8% accuracy). However, an interesting improvement was found by combining boosted stump predictions into neural networks over the conventional boosting method (from 77% to 91.1% accuracy on the test data).

We hypothesize that combination of ensemble methods with neural networks gives higher prediction performance in high dimensional data and (or) highly correlated features. Overall, each method exhibits similar prediction performance in the range of 91%-95% except the standard AdaBoost method (77% accuracy).

The scope of this article does not go beyond the setup where the number of items is strictly greater than the number of features (i.e. n > p). A plausible extension of this article is to apply the suggested methods in high dimensional data (i.e. n < p) with correlated features. To enrich the methodological contribution, the article can be extended to a heterogeneous setup: the influence of a feature on the target variable might vary based on particular subgroups of the feature space. A data-driven way to find these subgroups is to use heterogeneous effects using generalized random forests (see Wager and Athey (2018)).

TABLE II. Accuracy of the Methods (%)

Method	Test Data
Neural Networks	92.8
Random Forests	94.8
Boosting (stumps)	77.0
Boosting (stumps with maximum depth 3)	92.9
Deep Forest Learing	93
Boosting in Neural Networks (stumps)	91.1
Boosting in Neural Networks (stumps with maximum depth 3)	93.5

REFERENCES

Kong, Y. and Yu, T. (2018). A deep neural network model using random forest to extract feature representation for gene expression data classification. *Scientific reports*, 8(1):1–9.

Roesler, O. and Suendermann, D. (2013). A first step towards eye state prediction using eeg.

Souheil Al Hajj, G., Nareklishvili, M., and Seip, U. (2020). Artificial neural networks.

Sridhar, D. V., Seagrave, R. C., and Bartlett, E. B. (1996). Process modeling using stacked neural networks. *AIChE Journal*, 42(9):2529–2539.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Wager, S. and Athey, S. (2018). Estimation and inference of heterogeneous treatment effects using random forests. *Journal of the American Statistical Association*, 113(523):1228–1242.

Yu, D. C., Cummins, J. C., Wang, Z., Yoon, H.-J., and Kojovic, L. A. (2001). Correction of current transformer distorted secondary currents due to saturation using artificial neural networks. *IEEE Transactions on Power Delivery*, 16(2):189–194.