MA4270: Term Project

Akankshita Dash (A0132788U)
Peck Kai Ting (A0130651U)

1 Introduction

In this module, a myriad of classification and regression methods have been introduced, from perceptron to ensemble methods such as Adaboost, followed by the EM algorithm and Hidden Markov models.

For our project, we apply a few different classification methods described in Section 3, and compare these methods in Section 4. The dataset selected for the project is the *Titanic* dataset, obtained from the machine learning platform *Kaggle*. On 15 April 1912, more than 1500 men, women and children perished when the *RMS Titanic* sank in the North Atlantic Ocean on its maiden voyage. Here, we make an attempt at answering the question: *Given the type of passenger, can we predict who survived the tragedy?*

To run the algorithms, we use *Python*, as it has an extensive number of libraries that can be used for machine learning purposes.

2 The Dataset

The original *Titanic* dataset consists of a training set with 891 samples (each with a label 1 (Survived = Yes) or 0 (Survive = No)), and a test set with 418 samples (whose true labels are unknown to us), each consisting of a mix of 11 numerical and categorical attributes. As certain fields are irrelevant to the problem (e.g. Name, Cabin) they were not considered in this analysis. Rows with missing values for a given attribute were filled in with the median value (for numerical data, e.g. Age, Fare) and the mode value (for categorical data, e.g. Embarked). The full set of attributes are detailed in Section 2.2.

2.1 Handling Categorical Data: Encoding

In our dataset, the categorical attributes take on both *String* and numerical values. Categorical variables typically do not contain numerical relationships amongst their values. For example, if we let (1,2,3) represent the three ports of embarkation (Cherbourg, Queenstown, Southampton), the average of Cherbourg and Southampton is Queenstown, which does not make mathematical sense. In addition, the values they take on may not be correlated. In the above example, Queenstown is not more similar to Southampton than Cherbourg, even though the value 2 is closer to 3 than 1. In order for machine learning algorithms to better understand and harness the relationship for such data, a common way is to apply *one-hot encoding* on the data.

One-hot encoding involves the mapping of a categorical variable with k values to k binary (dummy) variables. For a categorical variable $x \in \{1, \ldots, k\}$, we map this to k (new) binary variables $x_1, \ldots, x_k \in \{1, \ldots, k\}$, where $x_j = 1 \iff x = j, \forall j = 1, \ldots, k$

Applying one-hot encoding to the attributes Embarked and Sex results in feature vectors of length 10, each with a corresponding label 1 (Survived = Yes) or 0 (Survived = No). The encoding was not applied to the *Pclass* attribute as it may contain some ordinal relationships.

2.2 Data Description

The list of variables are as follows.

Variable ID	Type	Description
Age	Numeric	Age of passenger in years
Embarked_C Embarked_Q Embarked_S	Categorical	Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)
Fare	Numeric	Passenger fare
Parch	Numeric	No. of parents / children aboard the Titanic
Pclass	Categorical	Ticket class (1 = 1st, 2 = 2nd, 3 = 3rd)
Sex_M Sex_F	Categorical	Gender (male / female)
Sibsp	Numeric	No. of siblings / spouses aboard the Titanic
Survived	Categorical	whether the passenger survived (1 = Yes, 0 = No)

Table 1: Table of variables

2.3 Training, Validation and Test sets

In training the data, we typically use a 90-10 split of the training (90%) and validation (10%) samples and perform 10-fold cross-validation. In addition to the cross-validation procedure, there is a test set (where the true labels are only known to *Kaggle*) which is used for verifying the test accuracy.

3 Methods

We explored the following six methods for binary classification:

1. Logistic Regression
2. Support Vector Machine (SVM)
3. Random Forest
4. Adaboost
5. XGBoost (a variation of Gradient Tree Boosting)
6. k-Nearest Neighbours
7. Neural Networks: Multilayer Perceptron (MLP)

3.1 Logistic Regression

We learnt about logistic regression in class and found it to be a good starting point for this binary classification problem. Logistic regression assigns a probability distribution over the two different labels such that data samples further away from the decision boundary are more likely to be correct, by using the sigmoid function as its loss function [8]. More formally, for the labels 0/1, we have

$$P(y = 1|x, \theta, \theta_0) = g(\theta^T x + \theta_0) \tag{1}$$

where the sigmoid loss function is given by

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

There are quite a number of issues with logistic regression; it assumes a linear relationship between the feature vector and output, tends to overfit if there are multiple highly correlated inputs, and is also sensitive to noisy data. However, training this simple model provided us with a benchmark for comparing against more complicated methods implemented later.

3.2 Support Vector Machine (SVM)

In class, we learnt about different types of SVMs (with or without offset and/or slack) and kernel functions. Here, we apply the C-SVM (with offset and slack), and experiment with different parameter settings (C, γ) for the Radial Basis Function kernel using the Bayesian optimization method as detailed in Section 3.8. Note that the Radial Basis Function kernel transforms input vectors $\mathbf{x}$ to $\phi(\mathbf{x}) = \exp(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2)$. We seek to solve the following problem (in primal form):

$$\begin{aligned} \min_{\theta,\theta_0} \frac{1}{2}\theta^T\theta + C\sum_{t=1}^{n}\xi_t \\ \text{subject to } y_t(\theta^T\phi(\mathbf{x_t}) + \theta_0) \geq 1 - \xi_t \quad \forall t = 1,\ldots,n \\ \xi_t \geq 0 \quad \forall t = 1,\ldots,n \end{aligned} \tag{3}$$

3.3 Random Forest

The random forest method is an ensemble-based method which uses a collection of decision trees to make classifications. A typical decision tree algorithm is a recursive algorithm and works as follows:

1. [Training] Given a set of examples E and attributes A at a node, if all examples have the same labels, return the classification as a leaf node.
   Else, if some terminating condition is met (e.g. no remaining examples), return a computed classification (e.g. by majority voting).Otherwise, select an attribute a to split on. Usually, this is done based on an information measure such as information gain or gini impurity.
   Suppose $a \in \{a_1, ...a_k\}$. Partition E into k groups $\{E_1, ...E_k\}$, based on the value of a the examples take on. For each attribute value a_k, use the set of examples E_k and remaining set of attributes $A\backslash\{a\}$ to (recursively) build a subtree, and attach this subtree onto the tree rooted at a. The algorithm is also illustrated in Algorithm 1.

2. [Prediction] Given an input $\mathbf{x}$, propagate it down the tree, checking for the matching attribute value at each branch of the tree. Upon reaching a leaf node, output the classification at that node.

Algorithm 1: Decision Tree Algorithm

```
Function Decision-Tree(examples, attributes, parent_examples)
    if if terminating conditions satisfied
        return classification;
    else
        A ← attribute to split on (based on a specified criterion e.g. information gain or gini impurity);
        tree ← make a new decision tree with root A;
        for each value a_j of a do
            curr_examples ← examples under this node, with attribute value a = a_j;
            subtree ← Decision-Tree(curr_examples, remaining attributes without a (A\a), examples);
            attach subtree to tree;
        end
        return tree
```

A random forest is a classifier consisting of a collection of independent and identically distributed (i.i.d) decision trees. Given an input $\mathbf{x}$, each tree is able to predict a class for $\mathbf{x}$, and casts a unit vote for that class. The final output for $\mathbf{x}$ for the random forest is the most popular class, decided by majority voting. In training each tree, samples are drawn with replacement, which adds elements of randomness into the model.

Random forests have been shown to obtain comparable results to other boosting methods such as AdaBoost, and are fairly robust to noise. In [1], a theoretical proof was also given for a rather useful result indicating that random forests do not overfit as the number of trees increases. Rather, generalization error converges to a limiting value. In our experiment, we attempt to build random forests with a varying number of trees to see if this is indeed empirically true.

3.4 Adaboost

We covered AdaBoost in class. Like random forest, it is also an ensemble-based method. The algorithm for binary classification is presented in pseudocode below.

Algorithm 2: AdaBoost Algorithm (dataset $D = \{(x_t, y_t)\}_{t=1}^{n}$, M base learners)

Initialize weights $W_0(t) = \tilde{W}_0(t) = \frac{1}{n} \quad \forall t = 1, \ldots, n$;
for *m = 1 to M* **do**
  Find a base learner $\hat{h}_m$ that minimises the weighted training error (using 0-1 loss)ϵ_m;
  Compute the optimized weighted training error $\hat{\epsilon}_m = \sum_{t=1}^{n} \tilde{W}_{m-1}(t) \mathbb{1}\{y_t \neq \hat{h}_m(x_t)\}$;
  Compute the vote for $\hat{h}_m$, $\hat{\alpha}_m = 0.5 \log \frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$;
  Update the weights $W_m(t) = W_{m-1}(t) \times \exp(-y_t \hat{\alpha}_m \hat{h}_m(x_t)) \quad \forall t = 1, \ldots, n$;
  Compute normalized weights $\tilde{W}_m(t) = \frac{W_m(t)}{\sum_{t'=1}^{n} W_m(t')} \quad \forall t = 1, \ldots, n$;
end
return *final classifier* $H(x) = \sum_{j=1}^{M} \hat{\alpha}_j \hat{h}_m(x)$

As mentioned in class, AdaBoost is resistant to overfitting, and the generalization error is also rather robust to the number of base learners used [6]. Note that we also had similar results for random forests in Section 3.3. In fact, Breiman presented an interesting conjecture about the two methods - that AdaBoost (a deterministic algorithm) is equivalent to a random forest (a random algorithm). Although the two are intrinsically different, it was also shown empirically that they achieved close error rates on a variety of datasets.

3.5 XGBoost

XGBoost, which stands for **Extreme Gradient Boosting**, is a gradient tree boosting technique used in many Kaggle competitions. Similar to AdaBoost, it combines weak learners into a single strong learner in an iterative fashion, but instead of using the exponential loss function, it uses an arbitrary (differentiable) loss function instead, which gives it an advantage.

The basic principle behind Gradient Tree Boosting is as follows: For a given data set, a tree ensemble model uses K additive functions to predict the output, given by

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i) \qquad , f_k \in \mathcal{F} \tag{4}$$

where $\mathcal{F}$ is the space of regression trees [3]. In order to learn the trees f_k, the regularized objective function that is minimized is given by

$$\mathcal{L}(\phi) = \sum_i l(\hat{y_i}, y_i) + \sum_k \Omega(f_k) \tag{5}$$

where

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda ||\omega||^2 \tag{6}$$

and γ and λ are parameters for the regularization term, ω is the weight of the leaves, and T is the number of leaves in the tree.

The model is then trained in an iterative manner - the f_t that most improves the objective is added to the predictive model. This *greedy* approach to selecting the ensemble may be computationally intensive, as it must enumerate splits over all the features. However, in XGBoost, there are two approaches to make this process more efficient - either the data is first sorted according to feature values, after which all the splits are enumerated, or the enumeration is only done over a proposed set of splits, which is obtained by considering percentiles of feature distributions.

The size of the resultant tree and the magnitude of the weights are controlled by regularization parameters, making the model resistant to overfitting.

3.6 k-Nearest Neighbours

The k-Nearest Neighbours algorithm is a popular non-parametric method used for classification problems. Its advantage lies in the fact that it assumes no underlying distribution of the dataset. A sample is classified by

a majority vote of its k neighbouring (nearest) samples (the most common distance metric used to calculate the neighbours is Euclidean distance, but one can also use Manhattan, Chebyshev and Hamming distance) i.e. the sample gets assigned the class which is most common among its k nearest neighbors. If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
The kNN algorithm is as follows [9] :

Algorithm 3: kNN (classify (X,Y,x), X: dataset of size N, Y: class labels, x: unknown sample)

for *i = 1 to N* **do**
  Compute the distance d(X_i,x)
end
Compute set I containing indices for the k smallest distances d(X_i,X)
return *majority label for Y_i, $i \in I$*

kNN has only one parameter that needs to be considered, and is thus easy to implement. Moreover, optimization of the hyperparamter k can be performed with the use of cross-validation.

3.7 Neural Networks

We considered Artificial Neural Networks for this problem and used neural_network libraries from **scikit-learn** to implement it.

3.7.1 Multilayer Perceptron

A multilayer perceptron (MLP) is a class of feedforward artificial neural network, inspired by biological processes (neurons) in the brain. It utilizes backpropagation and non-linear activation functions for training, often with one or more hidden layers. The backpropagation algorithm corrects the errors made by the neural network during training, while the non-linear activation functions (the more common ones are sigmoid, Rectified Linear Unit (ReLu) and tanh) attempt to classify non-linear data by introducing non-linearity into the output of a single neuron [5]. We illustrate the idea using a basic version of a neuron, known as the perceptron unit.

Figure 1 shows the structure of a perceptron unit. Given an input $\mathbf{x}$, the weighted sum is computed, and the resulting value is passed through an activation function to generate an output o. Note that here, $x_0 w_0 = 1 \times w_0 = w_0$ represents the bias.

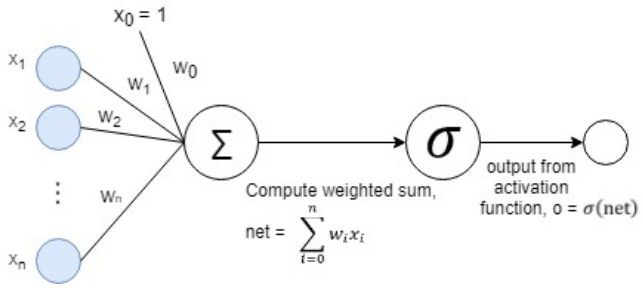

Figure 1: A single perceptron unit

The ReLu and sigmoid activation functions have equations as described in Equations 7 and 8.

$$f(x) = \max\{0, x\} \tag{7}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

An MLP consists of one or more hidden layers, each layer made up of such units. Each unit, like the one in 1 has its own set of weights. The output from each unit is passed on to the next layer, and so on, until a final output is generated in the output layer. In Figure 2, we have a simple MLP, with the following properties[1]

- 5 input units
- 1 hidden layer consisting of 3 hidden units

[1] for simplicity purposes, we do not display the bias term w_0 explicitly, but this is implied from the unit structure

- 1 output unit in the output layer

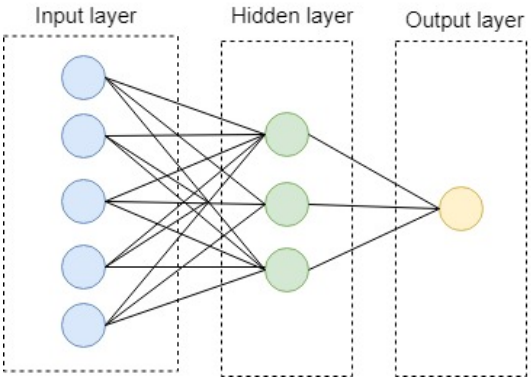

Figure 2: MLP with one hidden layer

We illustrate the basic idea for a backpropagation algorithm in Algorithm 4. With varying activation functions and other parameter settings, the computed errors and updates (by applying chain rule and working backwards through the layers) may also vary. Note that steps 5 and 6 are the backpropagation steps. The errors are propagated backwards and used to compute the weight updates for each unit.

Algorithm 4: Backpropagation Algorithm

input : Training data $\mathcal{D} = \{\mathbf{x}_t, y_t\}_{t=1}^{n}$
1 Initialize $\mathbf{w}$ to random values ;
2 **for each** *training example* $\boldsymbol{x}$ **do**
3 Input $\mathbf{x}$ through the MLP, and compute the output of every unit in the hidden and output layers;
4 For each hidden unit k and output unit h, compute the error δ_k and δ_h;
5 Update the set of weights for each hidden unit, where the update is a function of δ_k;
6 Update the set of weights for each output unit, where the update is a function of δ_h;
7 **end**

3.8 Bayesian Optimisation

In most of the machine algorithms used, there are hyperparameters to be optimized; for instance, in the case of SVMs, the C term, and for XGBoost, *min_child_weight*, *colsample_bytree*, *max_depth*, *subsample*, *gamma* and *alpha*. Some common ways of optimizing for these parameters include GridSearch, RandomSearch and Bayesian optimization.

Bayesian optimization typically works by assuming a Gaussian Process Prior for the function $f(\mathbf{x})$ that we seek to maximize (in our case the accuracy score), and maintain a posterior distribution so that information from previous evaluations of $f(\mathbf{x})$ are used to find the next best point (parameter setting). More precisely, the posterior distribution is characterized by mean and variance $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$. The optimization procedure uses an *acquisition function* to select the next point $\mathbf{x}$ to evaluate. Amongst other such possible functions, two commonly used ones are the Expected Improvement (EI) and the Upper Confidence Bound (UCB).

Expected Improvement (EI): The improvement function is defined as

$$I(x) = \max\{0, f_{t+1}(\mathbf{x}) - f(\mathbf{x}^+)\}, \quad \text{where } f(\mathbf{x}^+) = \max_{x_i \in x_{1:t}} f(\mathbf{x}_i) \quad \text{(the best evaluation of } f(\mathbf{x}_i) \text{ thus far)} \tag{9}$$

The algorithm seeks to find $\mathbf{x}$ by maximizing the EI, i.e. $\mathbf{x} = \arg\max \mathbb{E}(I(\mathbf{x})|D_t)$, where D_t is the set of observations that have been evaluated. The EI is defined in Equation 10:

$$EI(\mathbf{x}) = \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+))\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0 \end{cases} \tag{10}$$

where $\Phi(\cdot)$ = c.d.f of the standard normal, and $Z = \frac{\mu(\mathbf{x} - f(\mathbf{x}_+)}{\sigma(\mathbf{x})}$

This expression in Equation 10 involves $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ in place of $f_{t+1}(\mathbf{x})$, and allows us to balance the trade-off between choosing $\mathbf{x}$ with high $\mu(\mathbf{x})$ (exploitation) versus high $\sigma^2(\mathbf{x})$ (exploration) [7, 2]. Alternatively, incorporating the ideas of balancing exploitation and exploration, one can also optimize what is known as the *Upper Confidence Bound* (UCB):

UCB: The function is given by

$$UCB(\mathbf{x}) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}) \tag{11}$$

where κ is a tunable parameter to balance exploitation and exploration.

Summarizing the above, the optimization procedure works as in Algorithm 5

Algorithm 5: Bayesian optimization procedure

```
for t = 1,2,... do
    Find x_t by optimizing the acquisition function;
    Compute the value of the objective function, y_t = f(x_t) + ε_t, where ε_t = gaussian noise;
    Augment the data D_t with (x_t, y_t) and update the posterior;
end
```

We attempt to find optimal parameter settings for the SVM and XGBoost algorithm in Section 4, using a Python library that implements both EI and UCB.

4 Experiments

As mentioned in Section 2.2, we used a 90-10 training-validation split of the data, and tested our models in Kaggle with the test set. We experimented on the above seven methods of binary classification, optimizing for the hyperparameters via Bayesian optimization for SVM and XGBoost. In addition,

- For SVM, we used an RBF kernel. In applying Bayesian optimization, we optimized for the hyperparameters C and let $\phi(\mathbf{x}) = \exp(-0.001\|\mathbf{x} - \mathbf{x}'\|^2)$ i.e., $\gamma = 0.001$
- For random forest, we used the gini impurity to measure quality of splits. We used a minimum leaf size of one sample, $\sqrt{n}$ features for each split, without limiting the maximum height of the tree. However, the number of trees was varied, with results shown in Figure 4.
- The base learners used in AdaBoost were decision stumps, and we used 50 of them. Our classifier implemented the discrete boosting algorithm SAMME (as covered in the quiz), which reduces to the original AdaBoost algorithm we learnt in class (as this is a binary classification problem).
- For Gradient Boosted Trees (GBT) implemented in XGBoost, we used Bayesian Optimization to optimize for the hyperparameters in Table 3.
- Using cross-validation techniques, we used $k = 3$ for our kNN classifier. This model gave us the worst validation and testing accuracy amongst the different models, which was not wholly unexpected, since kNN does not generalize well to categorical variables, and is less robust to noisy data.
- For the multilayer perceptron (MLP), we optimized the hyperparameters for our activation function = '*logistic*', weight optimization = 'lbfgs' (an optimizer in the family of quasi-Newton methods) , L2 regularization $\alpha = 0.001$, initial learning rate = 10^-6, and learning rate for further updates = '*invscaling*'. The hyperparameter tuning was performed through GridSearch. We also implemented early stopping to terminate training if validation score is not improving in successive iterations.

The validation and test accuracy for each of the models are shown in Table 2

Hyperparameter	Description	Range	Value
reg_alpha	L1 regularization term on weights	[0,10]	4.1764
colsample_bytree	Subsample ratio of columns when constructing each tree	[0.1,1]	1
gamma	Minimum loss reduction required to make a further partition on a leaf node of the tree	[0,10]	0
max_depth	Maximum tree depth for base learners	[0,15]	9
min_child_weight	Minimum sum of instance weight (hessian) needed in a child	[1,20]	19.7516
subsample	Subsample ratio of the training instance	[0.5,1]	1

Table 3: Table showing optimized hyperparameters for XGBoost

Hyperparameter	Description	Range	Value
activation	Activation function for the hidden layer	['identity','logistic','tanh', 'relu']	logistic
alpha	L2 penalty (regularization term) parameter	$[10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}]$	0.001
learning_rate	Learning rate schedule for weight updates	['constant', 'invscaling', 'adaptive']	invscaling
learning_rate_init	The initial learning rate used. It controls the step-size in updating the weights	$[10^{-6}, 5*10^{-6}, 10^{-5}, 5*10^{-5}, 10^{-4}, 5*10^{-4}, 10^{-3}, 5*10^{-3}, 10^{-2}, 5*10^{-2}]$	10^{-6}
solver	The solver for weight optimization	['lbfgs', 'sgd', 'adam']	lbfgs

Table 4: Table showing optimized hyperparameters for MLP

Model	Validation accuracy (3 s.f.)	Test accuracy (3 s.f.)	Optimized parameters (if any)	Time taken(s)
Logistic Regression	0.790	0.765	Not applicable	0.084
SVM	0.808	0.770	C = 125.265	0.522
Random Forest	0.817	0.751	90 trees	1.70
Adaboost	0.796	0.779	n_estimators = 50	1.84
XGBoost	0.825	0.789	(Refer to table 3)	4.77
kNN	0.724	0.641	k = 3	0.055
MLP	0.808	0.756	(Refer to table 4)	1456

Table 2: Table showing validation and test accuracy of models

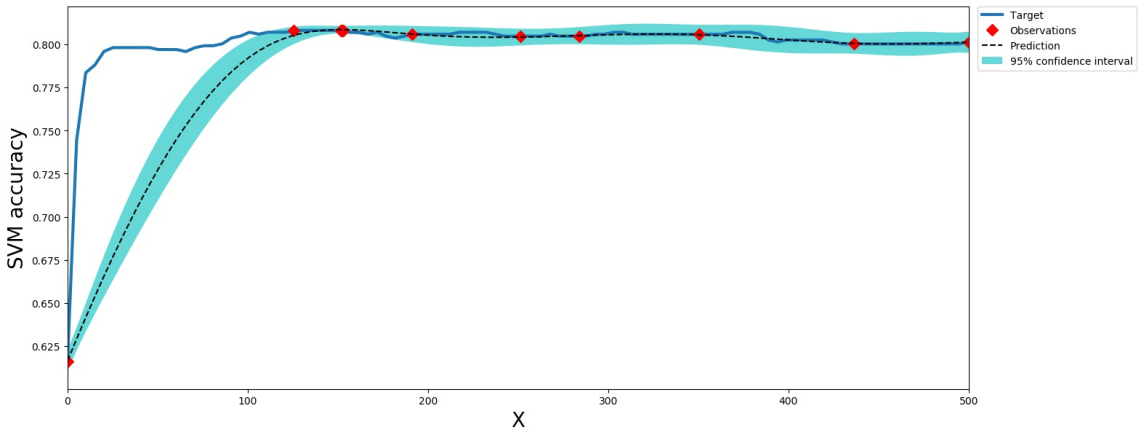

Figure 3: SVM Bayesian Optimization Process

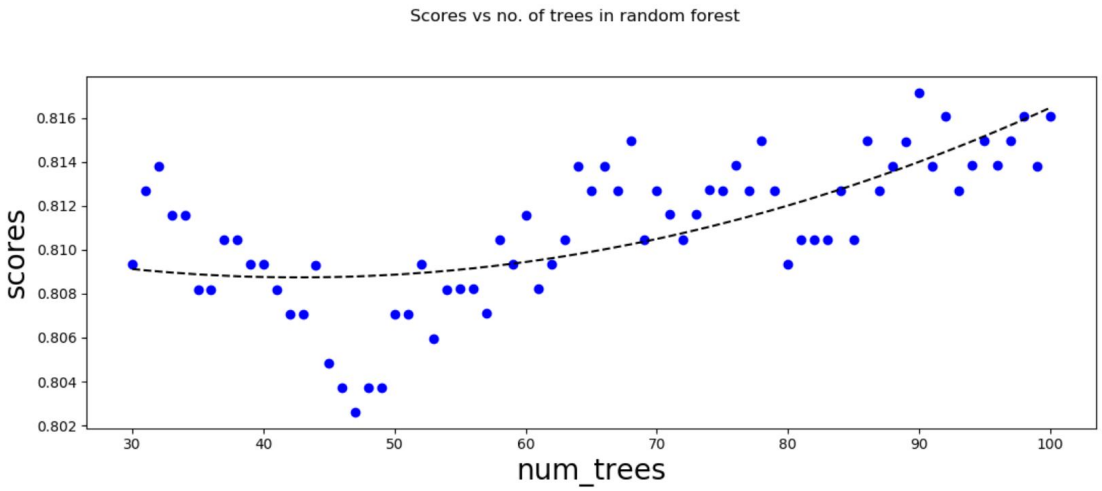

Figure 4: Random forest (no. of trees)

For the SVM Bayesian optimization process, with 15 iterations, the algorithm tends to obtain a relatively good fit to the actual accuracy score for the different settings of C (Figure 3). Also, we noticed from the results that the random forest does not tend to overfit as the number of trees for the model is increased, for the given range of num_trees used in our experiment. This concurs with what was described in Section 3.3.

In our case, the random forest and AdaBoost method did not yield exactly very similar results, contrary to Breiman's conjecture [1]. However, with a difference of about 0.028, the accuracy is not too far apart.

Although XGBoost takes a longer time to train than AdaBoost, it gives better results on both the validation and testing sets (both before and after we performed Bayesian Optimization on XGBoost). AdaBoost uses exponential loss, and the only parameter we can modify is the *maximum number of base learners / estimators*. In XGBoost, the loss function is generalized (using gradient statistics find splits), and there are multiple hyperparameters that can be tuned, making it more customizable and adaptive (especially for our dataset). Thus, XGBoost does better than AdaBoost in this case.

Our neural network (MLP) classifier gave disappointing results for the testing set. This was most likely due to overfitting - GridSearch returned 'lbfgs' as the best solver [2], so early stopping didn't take place. Neural networks find applications in complex problems such as image classification, natural language processing, and speech recognition (of which our dataset is neither). MLP also requires extensive tuning of a number of hyperparameters (e.g. number of hidden neurons, layers, and iterations), and is also sensitive to feature normalization (which did not make sense for our dataset). Moreover, MLP took a long time to train, and used significant memory during the training process.

Currently, XGBoost is the fastest implementation of Gradient Boosted Trees (the scikit-learn implementation is significantly slower than XGBoost).

If we compare XGBoost to Random Forests, we notice that both of them give similar validation accuracy, but XGBoost wins in test accuracy. We try to further understand this using the bias-variance tradeoff, which is made when we are generalizing beyond the training set.

Boosting is based on weak learners (high bias, low variance). The low variance is in part due to the averaging of output from the base learners. Boosting reduces error mainly by **reducing bias**. On the other hand, random forest uses fully grown decision trees (low bias, high variance). The variance also tends to increase with the depth of the trees, as it becomes more sensitive to fluctuations in the data. The random forest method tackles the error reduction task in an opposite way: by **reducing variance** [10].

In this case, based on our empirical results, reducing bias seems to give better results than reducing variance, since our Gradient Boosted Tree model (XGBoost) performs better than random forests.

[2]Scikit-learn only implements early stopping for stochastic gradient descent, so if the solver is something other than a stochastic gradient based solver (e.g. lbfgs) early stopping doesn't happen and the neural network will tend to overfit

5 Conclusions

Our benchmark classifier (Logistic Regression) gave surprisingly good results on Kaggle; in fact, most of our simpler methods (with the notable exception of kNN) gave reasonably good accuracy results on the dataset. We decided to apply neural networks (multi-layer perceptron) to this problem too to compare against these *simple* classifiers; however, the results from MLP were worse than expected. From the above, it seems that simpler classifiers tend to perform well on this dataset (Occam's razor).

Our best performing classifiers (according to testing set results on *Kaggle*) were our ensemble methods using decision trees - Random Forests, Adaboost and XGBoost. Our best classifier out of all was the XGBoost model. A close second was AdaBoost. Our boosted models performed better than random forests, implying that this dataset is likely to be more sensitive to changes in bias rather than variance.

It can be seen that our prediction accuracies for all the models hover around 80%. We believe the accuracy can be improved further through the following techniques:

- Additional tuning of hyperparameters (using Bayesian optimization/GridSearch)
- Construct ensembles of these models by stacking [4]
- Feature Engineering (create more features from existing ones)

References

[1] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[2] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[3] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[4] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine Learning*, 54(3):255–273, Mar 2004.

[5] Michael Nielsen. How the backpropagation algorithm works, Dec 2017.

[6] Robert E Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *Annals of statistics*, pages 1651–1686, 1998.

[7] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[8] Vincent Y.F. Tan. Lecture notes in ma4270: Data modelling and computation, 2018.

[9] Bunheang Tay, Jung Keun Hyun, and Sejong Oh. A machine learning approach for specification of spinal cord injuries using fractional anisotropy values obtained from diffusion tensor images. *Computational and mathematical methods in medicine*, 2014, 2014.

[10] Yisong Yue. Lecture notes in machine learning & data mining: Boosting & ensemble selection, 2016.