Comparative study of ML models on Titanic Survival Dataset

Diwash Pokharel
Department of EECS
University of Arkansas
diwashp@uark.edu

Md Taef Uddin Nadim
Department of EECS
University of Arkansas
nadim@uark.edu

Nidhi Gupta
Department of EECS
University of Arkansas
nidhig@uark.edu

Abstract

This paper presents comparative study of Machine Learning Models taught on Titanic Survival Prediction. The dataset used is from the famous Kaggle dataset *Titanic – Machine Learning From Disaster.* [1] The models used are Logistic Regression, Support Vector Machines, Neural Network and Random Forest. The evaluation of these models is compared based on the accuracy metric.

1 Introduction

We aim to study the behavior of Titanic survival prediction for different machine learning models. The main objective is to implement the learnings gained from the class lectures. Hence, we decided to go with different machine learning models such as Logistic Regression, Support Vector Machines, Neural Networks, Decision Trees and Random Forest. We compare the different models and the model variations using the metric Accuracy to check which models gives the best performance.

The rest of the paper is organized as follows: section 2 gives information about dataset, section 3 contains methodology and experiment details for Logistic Regression, Neural Network and Random Forest and this paper is concluded in section 4.

2 Dataset

On April 15, 1912, during her maiden voyage, the widely considered “unsinkable” RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren’t enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew. While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others [1]. The purpose of this dataset is to answer the question, what sorts of people were more likely to survive? using passenger data (i.e., name, age, gender, socio-economic class, etc.) and Machine Learning models. We have chosen Logistic Regression, SVM, Neural Network and Decision Tree models to compare which one answers the question most accurately.

The Titanic dataset has 891 samples, and it contains 11 features. Table 1 contains the overview of the features:

Table 1: Feature Description of the Titanic Dataset

Variable	**Definition**	**Key**
Survival	Survival	0 = No, 1 = Yes
pclass	Ticket	1 = 1st, 2 = 2nd, 3 = 3rd
Sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton
PassengerID	Unique Passenger ID	

Here 'pclass' is a proxy for socio-economic status (SES). Where, 1st = Upper, 2nd = Middle, 3rd = Lower. Age is fractional if less than 1. If the age is estimated, is it in the form of xx.5. 'sibsp' dataset defines family relations in this way, Sibling = brother, sister, stepbrother, stepsister And, Spouse = husband, wife (mistresses and fiancés were ignored). 'Parch' defines family relations in this way, Parent = mother, father, and Child = daughter, son, stepdaughter, stepson. Some children travelled only with a nanny, therefore parch=0 for them. [1]

2.1 Data Pre-processing

Initially we studied correlation of features to identify the relationship between the features and importance of different features. Before calculating the correlation, we dropped PassengerId, Name & Cabin as these features doesn't create much impact to our problem statement. The correlation study is shown in Figure 1.

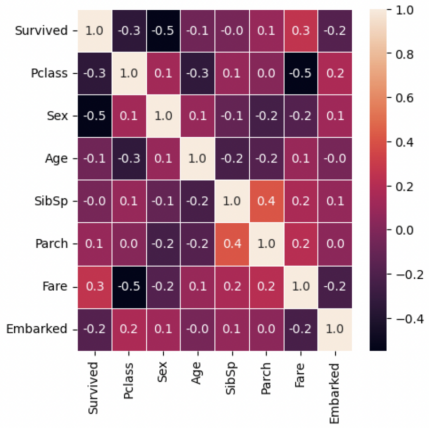

Figure 1: Correlation study of different features

From all the features, we decided to go ahead with features such as Pclass, Sex, Age, SibSp, Parch & Fare for our modelling. As a part of data pre-processing, sex feature was encoded binary categorical values. The Nan values for Age feature were replaced by its mean value. Scaled the continuous features such as Age, Fare, Parch, Pclass, Sibsp and the split between train and test data is 80:20.

3 Methodology and Experiment Details

This section contains different methodology, it's details and their experiment study.

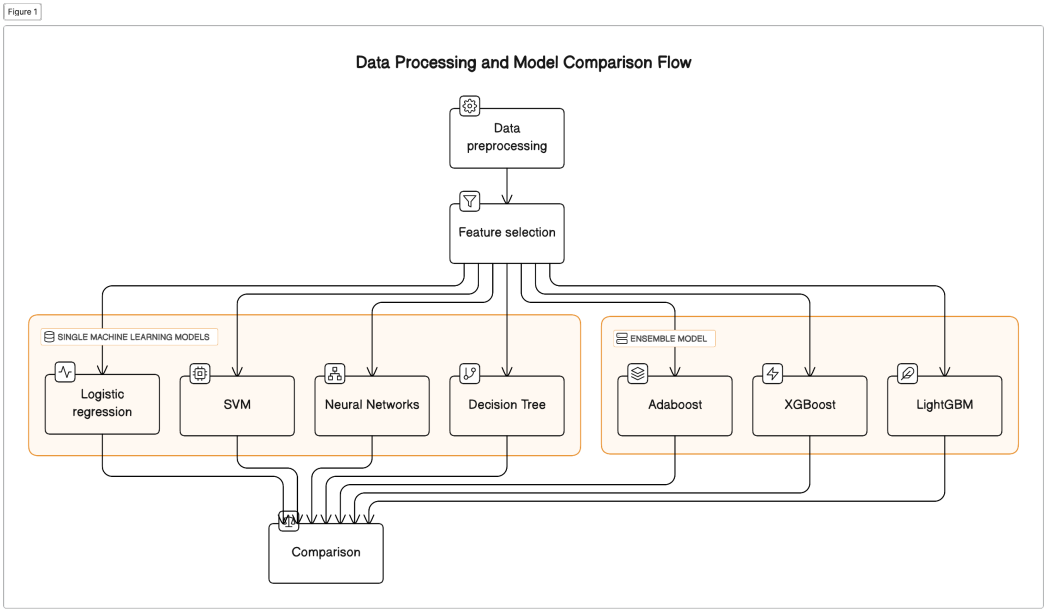

Figure 2: Methodology Structure

3.2 Part 1: Logistic Regression and Support Vector Machines

3.2.1 Logistic Regression: Details

Logistic Regression employs sigmoid function to find out the probability of a label. Sigmoid function is a type of mathematical function that is used to map predicted values to probabilities. The function is able to map any real value to another value within the range of 0 and 1. Using this sigmoid function, Logistic Regression can create a decision boundary that separates the classes among the feature space. The model uses maximum likelihood estimation for its training, which tries to maximize the likelihood of the feature data given the weights or parameters of the model. Figure 3 represents a plot of the sigmoid function.

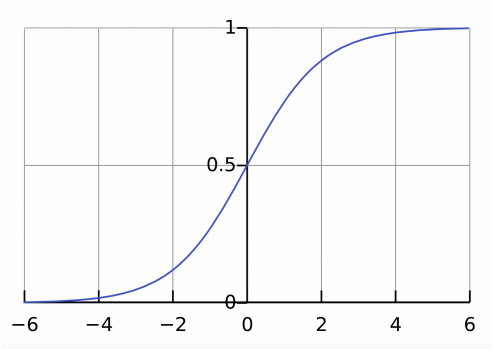

Figure 3: Sigmoid Function

3.2.2 Logistic Regression: Hyperparameters

There are three hyperparameters that we consider tuning in scikit-learn's LogistcRegression module: solver, penalty, and regularization strength. [2]
Solver: Solver is basically the algorithm to use in the optimization problem. 'liblinear' supports both L1 and L2 regularization penalties and well-suited for binary classification problems. 'lbfgs' solver is generally used for small to medium-sized datasets and supports L2 regularization. 'newton-cg' solver is appropriate for handling L2 regularization. 'sag' solver handles L2 regularization and well-suited for problems with many samples. And the 'saga' solveris an extension of the 'sag' solver that supports both L1 and L2 regularization.
Penalty: This hyperparameter determine the type of regularization to apply. 'None' means no penalty is added. 'l2' stands for adding a L2 penalty term and it is the default choice. 'l1' adds a L1 penalty term. 'elasticnet' used when both L1 and L2 penalty terms are added.
C: Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. [2]

3.2.3 Logistic Regression: Experiment Details and Observations

We first called the LogisticRegression() module of scikit-learn and used only the default values which are Solver: lbfgs, Penalty: l2, C=1.0. We observed the accuracy after that, and it was **81%**. Then we manually tuned the hyperparameters over the same train-test split. The penalty kept at 'l2' for all the solvers as only 'liblinear' and 'saga' solver only supports 'l1'. C was kept default. We observed the accuracy after that the accuracy was like this **81%** for lbfgs, liblinear, newton-cg, newton-cholesky and sag and **79.3%** for 'saga'. The accuracies obtained using different solvers shown in Table 2.

Table 2: Table for accuracies obtained for using different solvers in Logistic Regression

Solver	Accuracy (%)
lbfgs	81
liblinear	81
newton-cg	81
newton-cholesky	81
sag	81
saga	79.3

After that we used the GridsearchCV where we tuned C in the range of (0.1,1, 10, 100) and here also we kept the penalty at 'l2' only. (As from the observation accuracy doesn't change if we use 'l1' for the supported solvers). The best estimator provided by GridsearchCV was C=1, solver= 'liblinear'. Using these hyperparameters we again got the same accuracy of **81%**.

3.2.4 Support Vector Machine: Details

The primary objective of the Support Vector Machine or the Support Vector Classifier is to find out the optimal decision boundary or the hyperplane that separates the data points best into different classes. When there are given some linearly separable data, the hyperplane then maximizes the margin, which is the distance between the hyperplane and the closest data points from each of the classes, these are called the support vectors. Figure 4 depicts the overview of Support Vector Machine.

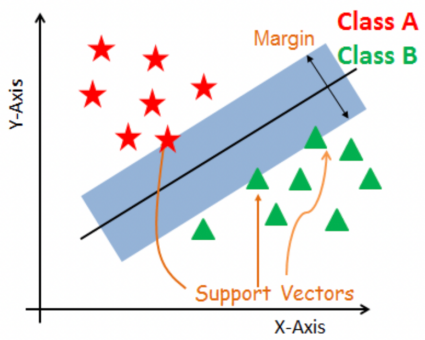

Figure 4: Support Vector Machine

3.2.5 Support Vector Machine: Hyperparameters

There are three hyperparameters that we consider to tune in scikit-learn's SVC() module. [2]
C: It is the Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.
Kernel: The kernel function allows SVMs to operate effectively in high-dimensional spaces without explicitly calculating the transformed feature space. 'Linear' represents a linear decision boundary in the original feature space. 'Polynomial' transforms the features into higher-dimensional space using polynomial functions. Degree parameter determines the degree of the polynomial. Radial Basis Function (rbf) transforms the data into infinite dimensions using a Gaussian-like function centered at each support vector. Sigmoid transforms data based on the hyperbolic tangent function or the sigmoid function.
Gamma: Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma, if 'auto', uses 1 / n_features, if float, must be non-negative. [2]

3.2.6 Support Vector Machine: Experiment Details and Observation

For the support vector classifier, we first used the default hyperparameters from the scikit-learn's SVC() module which are: C=1.0, Kernel= 'rbf', Gamma= 'scale' , Degree=3. The accuracy we observed was 83.2%. Then we used the GridsearchCV for hyperparameter tuning. The best estimator provided by GridSearchCV was {C=1, gamma = 'auto'}. The accuracy we found using these hyperparameters was **83.2%**. (using 'rbf' kernel). After that we tried manual tuning of hyperparameters using default C and 'Auto' Gamma and observed the accuracy. The best accuracy was found for 'rbf' kernel again and it was **83.2%**. The accuracy for all kernel is shown in Table 3. Also, the Accuracy Box Plot for different kernels shown in Figure 5.

Table 3: Table for Accuracies obtained for using different kernels in SVM.

Kernel	Accuracy (%)
RBF	**83.2**
Polynomial	80.4
Sigmoid	76.5
Linear	78.2

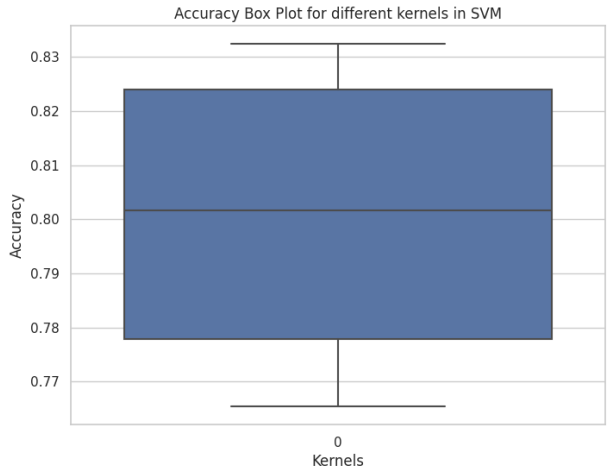

Figure 5: Accuracy Box Plot for different kernels in SVM

3.2 Part 2: Neural Networks

Neural Network mechanism involves majorly two approaches, first being forward propagation and second one is back propagation. With reference to Figure 6, the first layer has input nodes which are the input features and then it has hidden layer(s) nodes and one output node which has the value of the sum of the weighted nodes with corresponding activation function on it in-order to produce non-linearity in the data. Forward Propagation is to process data in the forward direction which makes the predictions, then the predicted values are compared with the true values through which the total loss is calculated. The goal is to minimize the loss between true values & predicted values and hence the Backward Propagation algorithm comes into picture by which the weights are updated through an optimization algorithm. This process is repeated iteratively until the terminal conditions are satisfied.

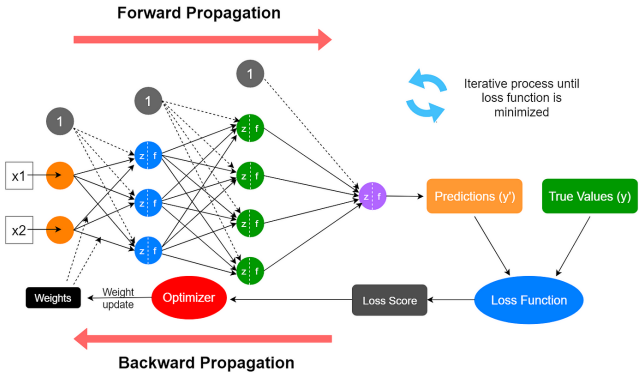

Figure 6: Neural Network Mechanism

3.2.1 Neural Networks: Model Details

For this task, the data pre-processing details is mentioned in section 2.1, the neural network model type is Sequential and the layer type is Dense. It has input layers as features and hidden layers, both input and hidden layers have their corresponding activation functions as ReLU. The output layer uses sigmoid function as the problem statement is binary classification task. For loss calculation, Binary Cross Entropy is used and a corresponding optimizer for loss minimization.

3.2.2 Neural Networks: Experiment Details

Packages used for this task are Scikit-Learn [2], Keras [3], NumPy [4] [5], Pandas [6], Matplotlib [7]. GridSearchCV is used to tune the optimal hyperparameters. GridSearchCV is a technique for finding the optimal parameter values from a given set of parameters in a grid. It's essentially a cross-validation technique. The comparison performance of each parameter is shown in Figure 7.

3.2.3.1 Experiment Details: Batch Size and Epochs

Different combinations of batch sizes and epochs were used to determine the best value. The result is shown in Table 4. The best value is achieved using batch size of 32 with 100 epochs as same can be seen from Accuracy Vs Batch Size & Epochs plot in Figure 7.

Table 4: Batch Size & Epochs Performance Comparison

Batch Size | Epochs	**Accuracy**
16 | 50	80.13%
16 | 100	80.58%
32 | 50	78.90%
32 | 100	**80.69%**
64 | 50	75.86%
64 | 100	80.24%

3.2.3.2 Experiment Details: Optimizers

Different optimizers such as SGD, RMSprop, Adagrad, Adadelta, Adam and Nadam were compared to choose the best suited for this task. The result is shown in Table 5. The best value is achieved using Adam optimizer and same can be inferenced from Accuracy Vs Optimizer plot in Figure 7.

Table 5: Optimizers Performance Comparison

Optimizers	**Accuracy**
SGD	78.79%
RMSprop	80.58%
Adagrad	61.83%
Adadelta	50.51%
Adam	**80.80%**
Nadam	80.13%

3.2.3.3 Experiment Details: Hidden Layer and Nodes

Different combinations of hidden layers and number of nodes were compared to determine the best value. The result is shown in Table 6. The best value is achieved using 2 hidden layers with {12, 6}, also shown in plot Accuracy Vs Layers in Figure 7.

Table 6: Hidden Layer(s) & Nodes Performance Comparison

Hidden Layers and Nodes	**Accuracy**
1 layer - {8}	80.92%
1 layer - {10}	80.47%
2 layers - {10, 5}	80.13%
2 layers - {12, 6}	**81.59%**
3 layers - {12, 8, 4}	80.91%

3.2.3.4 Experiment Details: Dropout

Dropout is a regularization parameter where some layer outputs are ignored or dropped at random during training phase depending on the dropout value. This helps to prevent overfitting of the model. The best accuracy is achieved using the dropout value of 0.01 as shown in Table 7 and Figure 7 Accuracy Vs Dropout plot.

Table 7: Dropout Performance Comparison

Dropout Value	**Accuracy**
0.0	80.92%
0.01	**81.37%**
0.05	80.13%
0.1	80.80%
0.2	80.58%
0.5	79.79%

3.2.4 Neural Networks: Final Model

After performing the hyperparameter tuning with different combinations, the final model parameters are summarized in table 8, with batch size of 32 and 100 epochs. The best suited configuration for hidden layers and nodes is 2 layers with nodes as {12, 6]. The optimizer selected is Adam. Lastly, for prevention of overfitting of the model, the dropout value chosen is 0.01. The final model accuracy is **82.34%** and the mean accuracy for 10 iterations is approximately **84.15%.** Figure 7 shows performance of different combinations of hyperparameters and Figure 8 shows the accuracy vs epochs plots as well as box plot for accuracy vs iteration.

Table 8: Final Model Layout

Batch Size	32
Epochs	100
Optimizer	Adam
Hidden Layers	2
Hidden Layer Nodes	{12, 6}
Dropout	0.01
Accuracy	**82.34%**

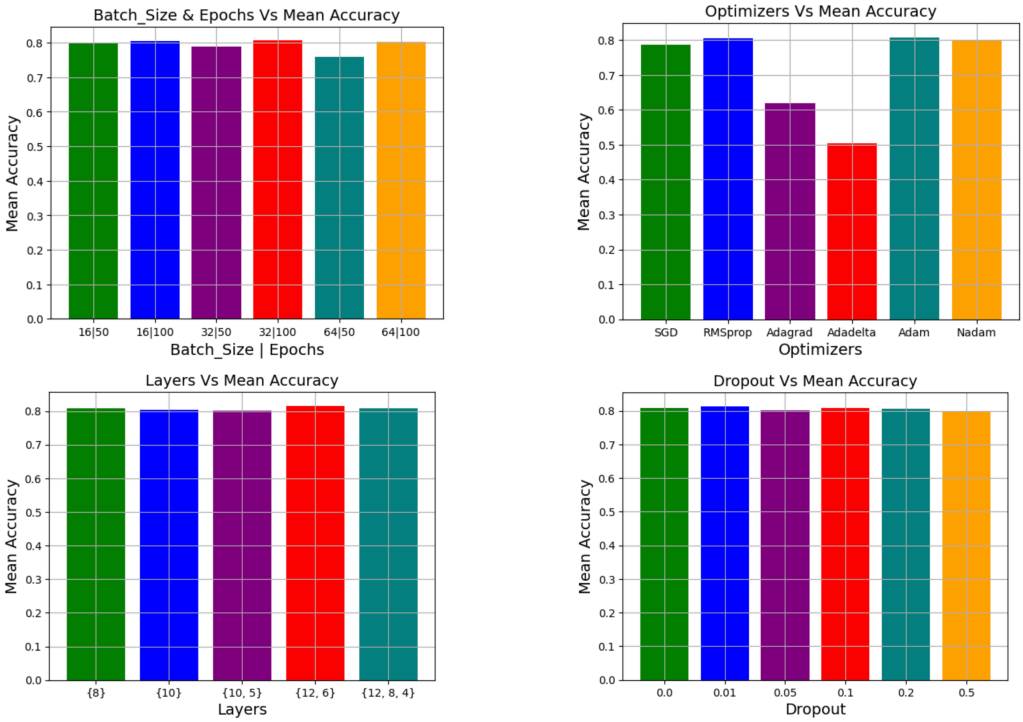

Figure 7: Hyperparameters vs Accuracy

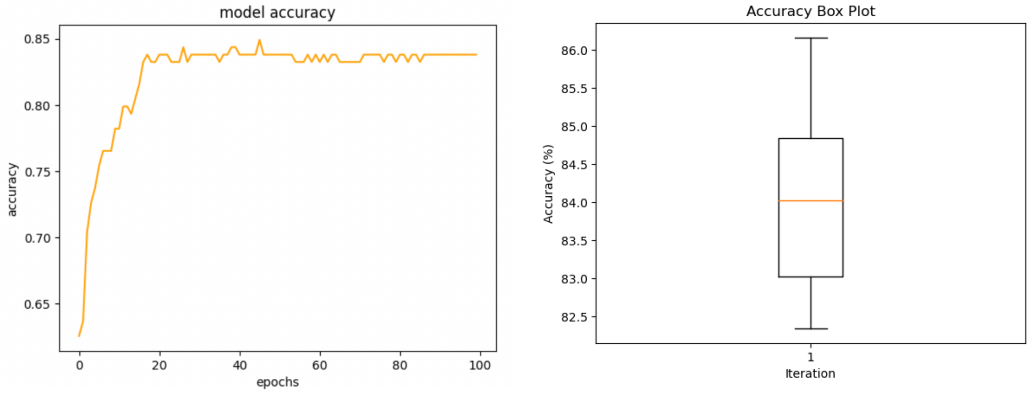

Figure 8: Final Model – Accuracy vs Epochs and Accuracy Box Plot

3.3 Part 3: Decision Trees

In this part of the report, we delve into the intricacies of Decision Tree, Random Forest, AdaBoost, XGBoost, and LightGBM models, assess their performance, and conduct hyperparameter tuning for optimal results. Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

For building the trees scikit-learn uses an optimized version of the CART algorithm. Using the feature and threshold that produce the highest information gain at each node, CART builds binary trees. For this model's data preprocessing we are using the same methods as mentioned above for Neural Networks.

3.3.1 Base Decision Tree Model

A Decision Tree classifier is initialized using scikit-learn's Decision Tree Classifier module which uses entropy as splitting criteria for the feature space which is just minimizing the log loss or binary cross entropy. After training on the 80:20 split of the dataset the model's accuracy was 75.42%.

So, to understand the effect of the choice of tree depth on accuracy we plotted the accuracy against it and found that 10 is a sufficient tree depth to fit the dataset.

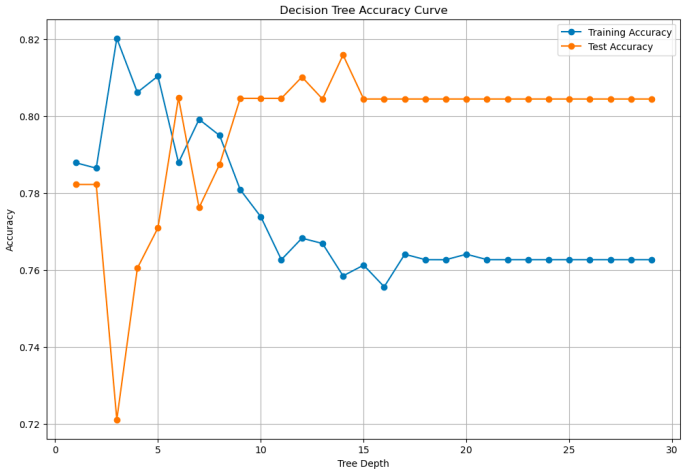

Figure 9: Depth Analysis- Accuracy vs Tree Depth

3.3.2 Random Forest Model

During training, Random Forests, an ensemble learning technique was used, build many Decision Trees. The average of all the Decision Trees' predictions yields the final predictions.

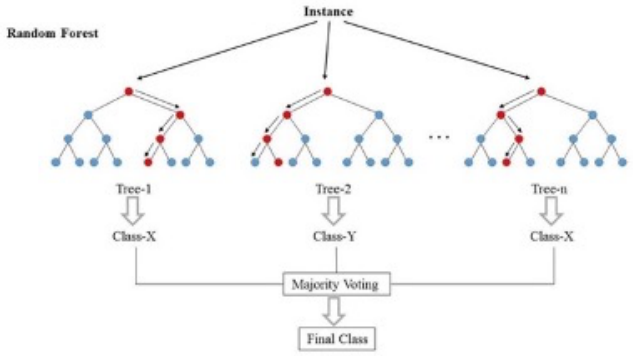

Figure 10: Random Forest Classifier working visualization.

So, for random forest classifiers the total number of estimators is a key hyperparameter which can dictate the overall accuracy of the ensemble. A technique called grid search is an effective way to determine this; we plotted the accuracy vs a range of number of estimators to determine the optimal n. The best accuracy was observed for n=200 which was 82.12%.

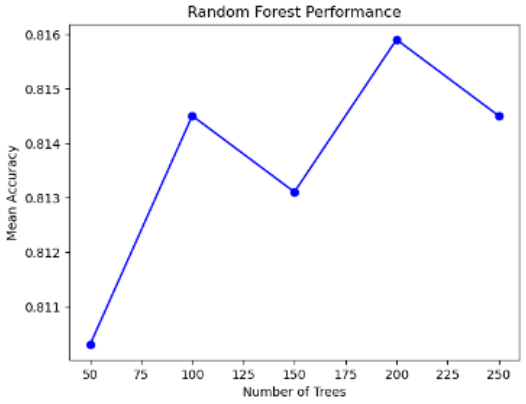

Figure 11: Grid search- best at n=200

To make sure we are heading in the right direction we evaluated this new improved model by using ROC curves and confusion matrix on the test data. From the ROC curve we can see a clear Improvement of model's accuracy of the random forest model over the base decision tree model.

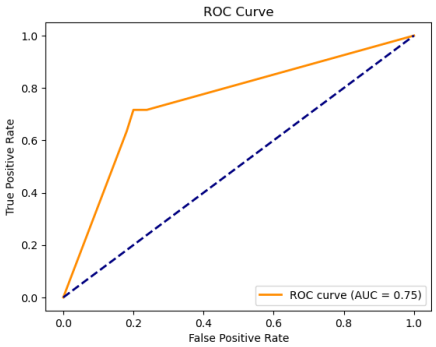

Fig 12(a)

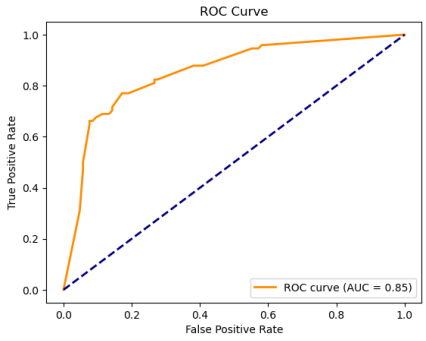

Fig 12(b)

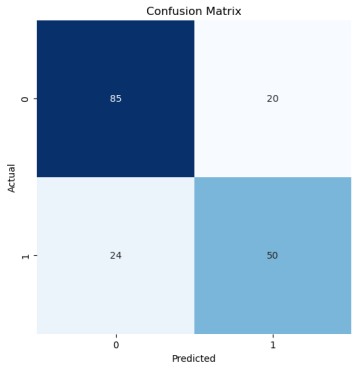

Fig 12(c)

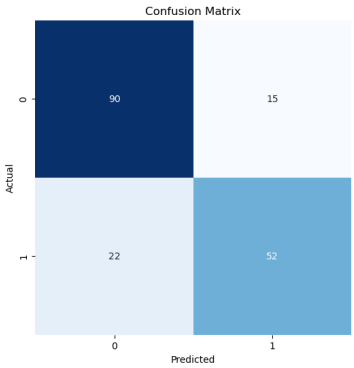

Fig 12(d)

Figure 12: Comparison between ROC of Random Forest and base model (Fig 12(a) & 12(b)); Between Confusion matrix of Random Forest and base model (Fig 12(c) & 12(d))

3.3.3 Boosting Methods

Boosting or hypothesis boosting is a meta-training technique which involves using weak hypothesis to build a strong classifier. The basic idea of boosting is to learn from the mistakes made by previous classifiers by assigning weights to misclassified data as well as the final ensemble of weak learners. Boosting builds a better classifier iteratively and quickly.

3.3.3.1 Hyperparameter Tuning

Here in our project, we apply AdaBoost [8], XGBoost [9] and LightGBM [9] algorithms to improve the classifier accuracy. For these methods also we need to tune the number of estimators as a hyperparameter. Using grid search the best accuracy are found at n=150 for AdaBoost and 50 for XGBoost and LightGBM.

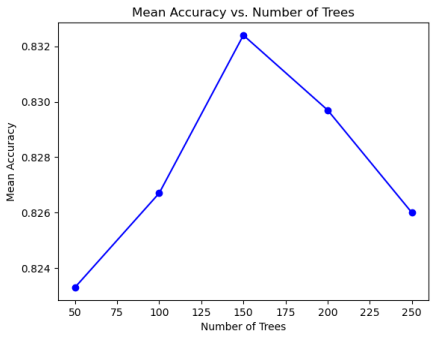

Fig 13(a)

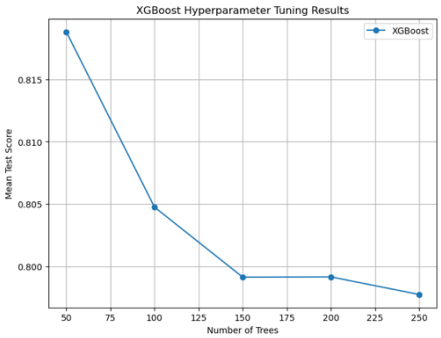

Fig 13(b)

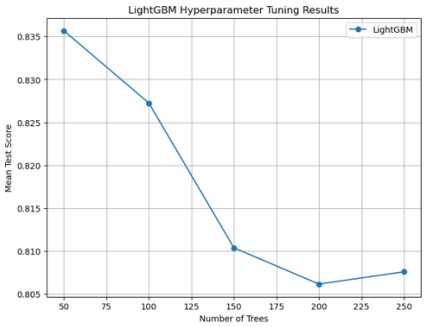

Fig 13(c)

Figure 13: Determining the best number of estimators: Fig 13(a) AdaBoost, Fig 13(b) XGBoost, Fig 13(c) LightGBM

3.3.3.1 Evaluation: Training and Testing Accuracies

We conducted 10-fold cross validation on the training data to obtain the accuracy of using different boosting methods. From the box plot we can see that on the training set the best performing model is LightGBM with a mean accuracy of 83% and even reaching above 90% at times. [10]

But on the test data the best performing model was LightGBM at 85.47% and the worst was base random forest classifier at just 81.13%. The comparison can be seen in the bar graph.

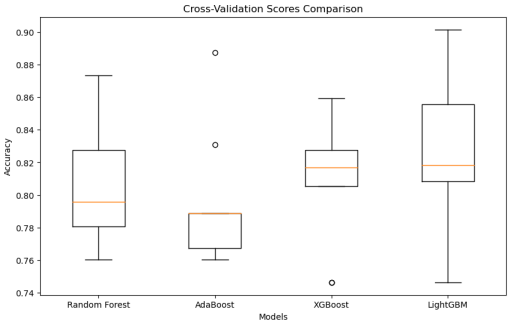

Figure 14: Training Accuracies

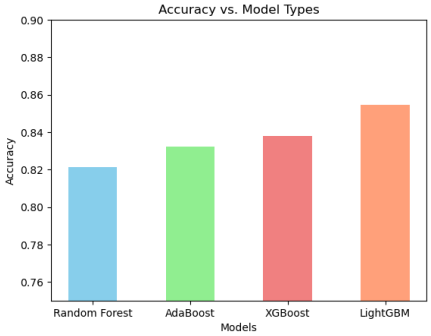

Figure 14: Testing Accuracies

4 Conclusion

The *Titanic: Machine Learning from disasters* is an excellent competition for illustrating the workings of machine learning model that predicts which passengers survived the Titanic shipwreck. We were thoroughly able to analyze the meaning behind each feature and able to work out the best features for the model namely passenger class, sex, age, siblings, parent child and the fare. After following the due diligence of data preparation and representation, we were successful in crafting numerous models of wide variety to model this problem.

After studying the different models, we found that out of Logistic Regression, Support Vector Machines, Neural Networks, Decision Trees & Random Forest, LightGBM gives the highest performance with 85.47% accuracy. The different model and its variations performance is summarized in Table 9.

Table 9: Different model comparison

Models	**Accuracy**
Logistic- Default	81.00%
Logistic - Tuned	81.00%
SVM- Default	83.24%
SVM- Tuned	**83.24%**
Neural Network – 32 batch & 100 epochs	80.69%
Neural Network – Adam	80.80%
Neural Network – 0.01 dropout	81.37%
Neural Network – 2 hidden layers	81.59%
Neural Network – Final Model	82.34%
Neural Network – 10 iterations mean acc.	**84.15%**
Decision Tree (Base)	79.33%
Random Forest	81.13%
AdaBoost	83.24%
XGBoost	83.79%
LightGBM	**85.47%**

Acknowledgments

The authors would like to thank Dr. Lu Zhang (Dept. of EECS, University of Arkansas) for giving this opportunity and his teachings in class lectures. Also, a heartful thanks to the team members and colleagues for their hard work and intellectual contributions in this project.

References

[1] W. Cukierski, "Titanic - Machine Learning from Disaster," 2012. [Online]. Available: https://kaggle.com/competitions/titanic.

[2] F. Pedregosa, "Scikit-learn: Machine Learning in Python," 2011.

[3] F. e. a. Chollet, "Keras," 2015.

[4] T. Oliphant, "NumPy documentation," [Online]. Available: https://numpy.org/doc/stable/user/index.html. [Accessed 7 12 2023].

[5] C. R. e. a. Harris, Array programming with NumPy, Nature 585, 357–362, 2020.

[6] T. p. d. team, pandas-dev/pandas: Pandas, Zenodo, 2020.

[7] J. D. Hunter, Matplotlib: A 2D graphics environment, IEEE COMPUTER SOC, 2007.

[8] R. E. Schapire, "Explaining adaboost," in *Empirical Inference: Festschrift in Honor of Vladimir N. Vapnik*, Springer, 2013, pp. 37--52.

[9] S. Kothari, "What Is Boosting in Machine Learning: A Comprehensive Guide," 24 6 2023. [Online]. Available: https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-boosting. [Accessed 12 2023].

[10] C. M. Ellis, "Titanic feature selection: Permutation importance," 2020. [Online]. Available: https://www.kaggle.com/code/carlmcbrideellis/titanic-feature-selection-permutation-importance. [Accessed 7 12 2023].