FYS-STK3155/4155 Applied Data Analysis and Machine Learning - Project 3

Lotsberg, Bernhard Nornes
Nguyen, Anh-Nguyet Lise

https://github.com/liseanh/FYS-STK4155-project3

November - December 2019

Abstract

Whole page: 6.24123in Column: 3.01682in

1 Introduction

A wide array of machine learning methods have been developed over the years for various predictive purposes, with the the neural network being the probably most well-known method among the general population. In recent years many other statistical learning methods have proven themselves as well however. In this project we compare the performance of neural networks and gradient boosting in the case of binary classification. In addition to these, we also use the much simpler k-nearest neighbours method as a baseline for classification performance.

2 Data

The data set we will analyse in this project is the MAGIC Gamma Telescope data set retrieved from the UCI Machine Learning Repository, which was generated by a Monte Carlo (MC) program described by D. Heck et. al. [4] to simulate high energy gamma particle registration in a Cherenkov gamma telescope. The set consists of ten explanatory variables and a binary response variable `class` which specifies whether the measured photons resulted from a gamma particle (`class = g`) or a hadron (`class = h`). The entire data set consists of 19020 instances with no missing values, with outcome distribution as shown in Figure 1. The explanatory and response variables are defined as the following by the UCI Machine Learning Repository [2]:

1. `fLength`: continuous # major axis of ellipse [mm]
2. `fWidth`: continuous # minor axis of ellipse [mm]
3. `fSize`: continuous # 10-log of sum of content of all pixels [in #phot]
4. `fConc`: continuous # ratio of sum of two highest pixels over `fSize` [ratio]
5. `fConc1`: continuous # ratio of highest pixel over `fSize` [ratio]
6. `fAsym`: continuous # distance from highest pixel to center, projected onto major axis [mm]
7. `fM3Long`: continuous # 3rd root of third moment along major axis [mm]
8. `fM3Trans`: continuous # 3rd root of third moment along minor axis [mm]
9. `fAlpha`: continuous # angle of major axis

with vector to origin [deg]

10. `fDist`: continuous # distance from origin to center of ellipse [mm]
11. `class`: g, h # gamma (signal), hadron (background)

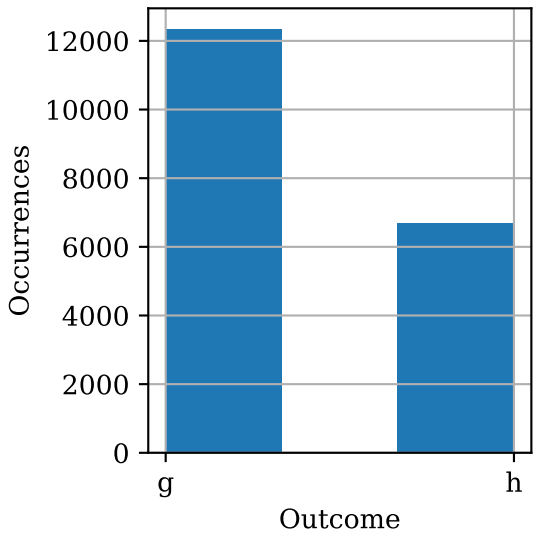

Figure 1: Frequencies of the outcomes g and h in the data set. The numbers of instances for the categories were 12332 and 6688 for g and h respectively.

The correlation between the features can be seen in Figure 2.

Preprocessing

To preprocess the data, we first split the data into a training set and a test set, with 2/3 of the total data set as training data and the remaining 1/3 as the test set, keeping the proportion of the two outcomes approximately equal. All the explanatory features are continuous, so each feature was centered by subtracting the mean of its values and scaled with respect to its standard deviation. The scaling and centering was executed with respect to each feature in the training set.

The response `class` is a categorical variable with classes g (gamma particle) and h (hadron). These were onehot-encoded for our k-nearest neighbour and multilayer perceptron procedures, whereas the extreme gradient boosting procedure managed this on its own.

3 Methods

3.1 k-Nearest Neighbour (kNN)

The k-nearest neighbour method is a non-parametric method that consists of using the k observations in the training set in feature space closest to a point x to form a model $\hat{y}$. For regression and binary classification, the method considers a point x and the k closest points x_i in the neighbourhood $N_k(x)$ defined by k, such that

$$\hat{y} = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i, \tag{1}$$

i.e. the model output is the average of the k closest points to x.

As we are using this method for binary classification, we assign the classes such that

$$\hat{Y} = \begin{cases} \text{g if } \hat{y} \leq 0.5 \\ \text{h if } \hat{y} > 0.5 \end{cases} \quad [3]. \tag{2}$$

Alternatively, the observation can be assigned according to the majority class of its neighbours, such that the observation is classified as the most common class in its neighbourhood. This is the method used for multiclass classification.

To implement kNN and the hyperparameter optimisation in this project we use the Scikit-Learn library.

3.1.1 Hyperparameter tuning

The kNN method is a simple approach that only requires tuning of a single hyperparam-

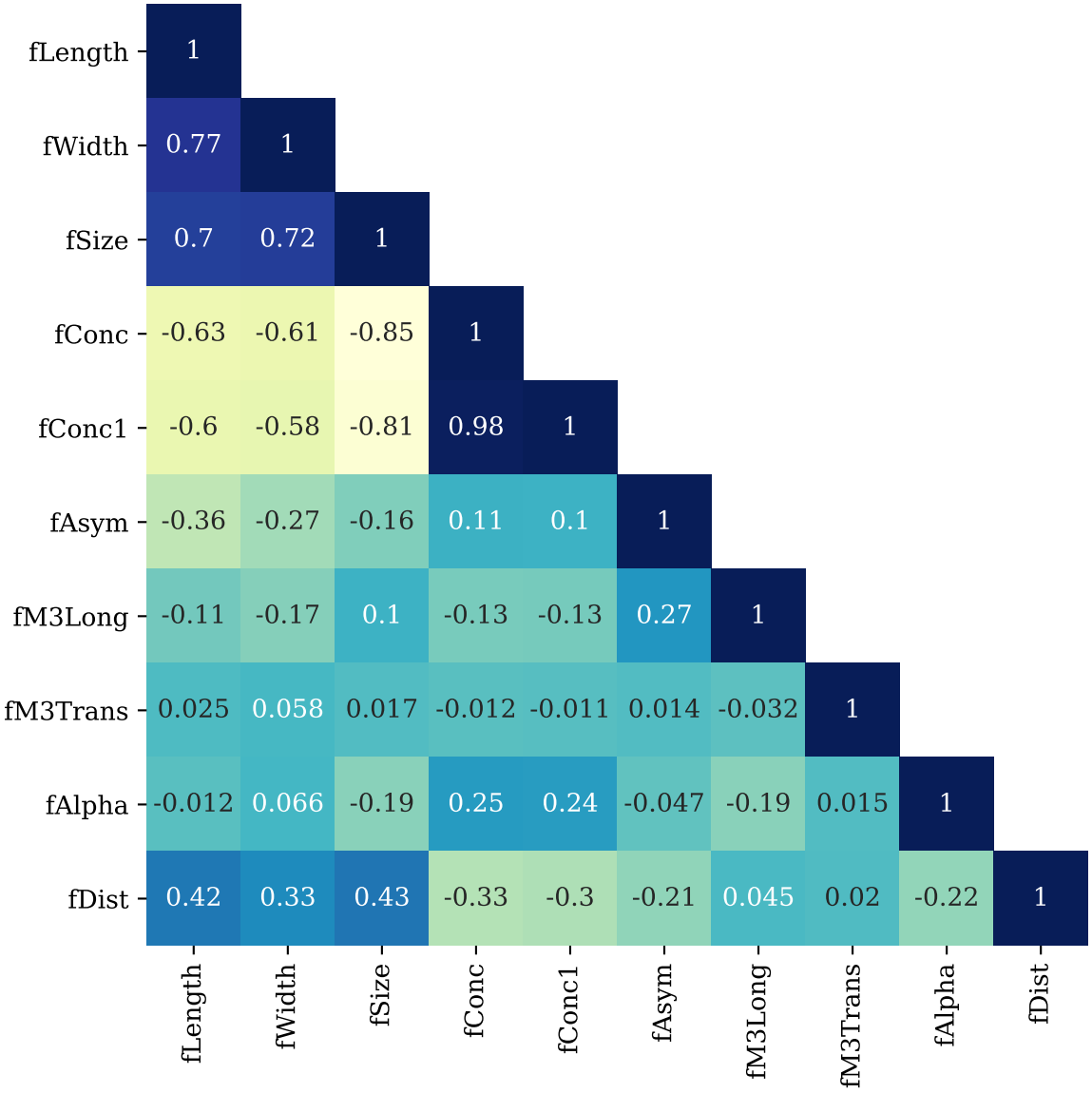

Figure 2: Correlation matrix of the features in the train set. Upper triangle excluded for readability.

eter k. Using $k = 1$ results in a highly complex model with high variance, while using $k = N$, where N is the total number of samples in the training set, results in a simple, highly biased model. To find the optimal value for k in our case, we use 5-fold cross-validation on the training set, keeping 1/3rd of the total data as test set.

3.2 Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) is a feed-forward neural network. It contains an input layer, one or more hidden layers, with tunable number of nodes and layers, and a final output layer. The information flows only in one direction from the input layer to the output layer. The input and output values of the nodes are determined by an activation function, in addition to the weights and biases of the nodes. For a more extensive explanation of how the different components of the MLP works, please refer to our previous work, *Project 2: Regression and Classification* [5].

We have chosen to use the Rectified Linear Unit (ReLU) function as our activitation function $f(z)$ between the layers, which is given by

$$f(z) = \max(0,\, z), \tag{3}$$

and its gradient by

$$f'(z) = \begin{cases} 0, & z < 0 \\ 1, & z > 0 \end{cases}. \tag{4}$$

For the activation function of the final output layer, we use the softmax function to achieve the categorical outcomes,

$$f(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{5}$$

For this project we will be using the TensorFlow library to implement the MLP.

3.2.1 Hyperparameter tuning

In this project we choose to limit ourselves to two hidden layers for computational reasons, and tune the amount of nodes in both of those layers.

To counteract the high variance of the model, we introduce regularisation using dropout. Dropout is tuned for eachlayer, and is the chance of any node in the given layer to be ignored at an iteration in training. This reduces variance by forcing the other nodes to be able to work more independently from each other, lessening the chance for overfitting.

We also tune the batch size. This leaves us with five hyperparameters to tune. A grid search would be too expensive to perform, so instead we perform a random search using the Scikit-Optimize library to optimise the hyperparameters. Ideally we would employ cross validation on the training set to evaluate the candidate hyperparameters, but for computational reasons we have decided to instead set aside 0.10 of the training set for validation. After finding the tuning parameters that maximise F_1 on the validation set, we use these parameters to fit the model on the entire training data including the validation data. This is known as a refit.

3.3 Gradient Boosting and Extreme Gradient Boosting (XGB)

Boosting is a learning technique based on the idea of combining several weak learners b_m into a final strong learner $f_{m_{\text{stop}}}$, where each f_m is improved from the previous iteration f_{m-1} through the step h_m,

$$f_{m_{\text{stop}}} = \sum_{m=0}^{m_{\text{stop}}} h_m. \tag{6}$$

Gradient boosting is one such method. We consider a loss function $L(y, f(x))$, which is

minimised by calculating the negative gradient of the loss function to obtain the minimisation direction. To prevent overfitting, the boosting step size, also referred to as learning rate, $\nu \in (0, 1)$ is introduced as a tuning parameter, with smaller values of ν resulting in larger shrinkage,

$$f_m(x) = f_{m-1}(x) + \nu h_m. \tag{7}$$

The gradient boosting algorithm goes as follows: [1]

1. Initialize the estimate $f_0(x)$
2. for $m = 1, , \ldots, m_{\text{stop}}$:
   (a) Compute negative gradient vector
   $$u_m = -\frac{\partial L(y, f(x))}{\partial f(x)}\bigg|_{f(x)=\hat{f}_{m-1}(x)}$$
   (b) Fit the base learner to the negative gradient vector, $b_m(u_m, x)$
   (c) Update the estimate
   $$f_m(x) = f_{m-1}(x) + \nu b_m(u_m, x)$$
3. Compute final estimate,
   $$\hat{f}_{m_{\text{stop}}}(x) = \sum_{m=0}^{m_{\text{stop}}-1} \nu b_m(u_m, x)$$

For this project, we have chosen to boost decision trees using an implementation of gradient boosting called extreme gradient boosting (XGB) using the XGBoost library.

3.3.1 Hyperparameter tuning

The XGBoost package has many additional hyperparameters in comparison to the ordinary gradient boosting algorithm we described. As with the neural network model, we also do not use cross validation here due to computational cost. We instead use the same train-validation split we had for the neural network and employ a randomised search instead of a grid search using Scikit-Optimize to find the optimal hyperparameters.

The hyperparameters we are tuning are the learning rate, maximum tree depth of the base learners, minimum child weight, and the L1 and L2 shrinkage parameters on the weights. Another useful potential hyperparameter to tune would be the number of estimators to fit, but we choose to let this be managed by XGBoost's early stopping criterion. This criterion stops the training if the validation metric does not improve at least once over a set number of iterations, which we have chosen to be 3, meaning the algorithm will stop if the three last iterations did not improve the model.

The learning rate as described earlier controls the step size of the boosting algorithm with smaller values leading to smaller steps, thus decreasing the amount of correction in each term and preventing overfit. Lower learning rates will lead to an increase in boosting iterations before the stopping criterion is met.

The maximum tree depth is the maximum amount of nodes a base tree is alllowed to grow. Trees with more nodes are able to model more complex relations. However, allowing an excessive amount of nodes will likely lead to overly complex models, so we are tuning this in an attempt to prevent overfit.

The minimum child weight is the minimum weight needed to construct a new tree node, meaning that the tree needs more samples corresponding to the split to create the node. Similarly to the maximum tree depth, it controls model complexity of the tree. Higher values of minimum child weight correspond to less complex models, while lower values correspond to more complex models [6].

The L1 shrinkage parameter shrinks the weights according to the L1 loss term, while the L2 shrinkage parameter shrinks the

weights according to the L2 loss term.

There are numerous other potential hyperparameters to tune, but we have chosen to focus on these five as tuning all of the parameters would be computationally expensive.

3.4 Model evaluation

One method of evaluating classification models is by using the accuracy score,

$$\text{accuracy} = \frac{\sum_i^N I(y_i = \hat{y}_i)}{N}, \tag{8}$$

where I is the indicator function, N is the number of samples in the data set, $\hat{y}_i$ is our model output and y_i is the true observation. The error is then given by

$$\text{error} = 1 - \text{accuracy}. \tag{9}$$

However, as the data set we are using is imbalanced, this is a poor metric of model performance. Instead, we can visualise the performance of a model using the so-called confusion matrix. In our case it is defined as

$$\text{confusion matrix} = \begin{pmatrix} g_g & g_h \\ h_g & h_h \end{pmatrix}, \tag{10}$$

with g_g indicating the amount of correctly classified gamma particles, g_h indicating the amount of incorrectly classified hadrons (false positives), h_g indicating the amount of incorrectly classified gamma particles (false negatives) and h_h indicating the amount of correctly classified hadrons. Mathematically, if $\hat{y}_i$ is our model output, y the true observations, and I the indicator function, this is expressed as

$$\begin{aligned} g_g &= \sum_i^N I(y_i = g_i \text{ and } \hat{y}_i = g_i) \\ g_h &= \sum_i^N I(y_i = g_i \text{ and } \hat{y}_i = h_i) \\ h_g &= \sum_i^N I(y_i = h_i \text{ and } \hat{y}_i = g_i) \\ h_h &= \sum_i^N I(y_i = h_i \text{ and } \hat{y}_i = h_i). \end{aligned}$$

A perfect model will have $g_h = h_g = 0$, i.e. it would not misclassify any of the observations and only have non-zero elements on the diagonal.

A common way to evaluate classification models is the F_1-score. This is a metric quantifying the amount of diagonal and off-diagonal elements of the confusion matrix and is defined as

$$F_1 = \frac{2g_g}{2g_g + g_h + h_g} \tag{11}$$

The F_1 score values can lie between 0 and 1, with 1 representing a perfect fit with no off-diagonal elements. We use the F_1 score to evaluate the MLP and XGB models and select the hyperparameters that yield the models with the highest F_1 scores.

When considering the confusion matrix, the best performing model for this data set is the model which maximises F_1 and minimises g_h on the test set, as false negatives are worse than false positives in this case as they lead to loss of important data.

4 Results

Figure 3 shows the result of the cross-validated grid search for the optimal k in the kNN model and the corresponding confusion matrix. The value of k that resulted in the highest F_1 score was $k = 5$. The F_1 scores for

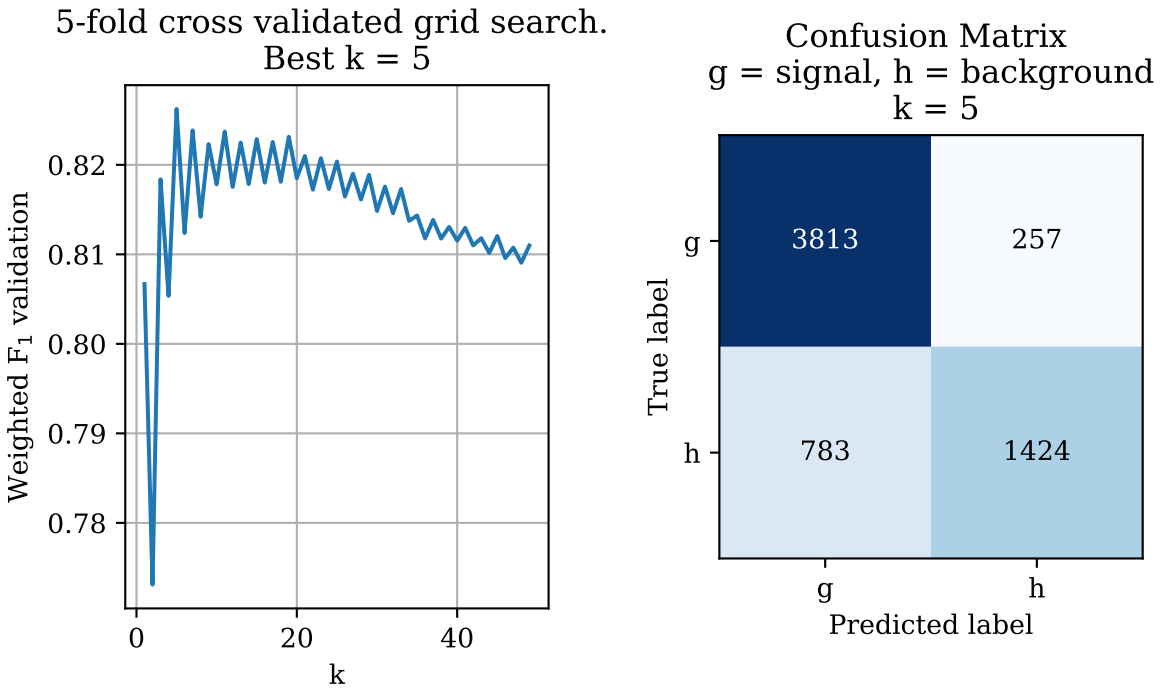

Figure 3: Results from tuned kNN using cross validation. The confusion matrix was found using the test set.

Table 1: Estimated optimal hyperparameters found using randomised search for the multilayer perceptron model.

Hyperparameter	**Optimal value**
Batch size	109
Epochs	55
Dropout, first layer	0.13
Dropout, second layer	0.095
Nodes, first layer	17
Nodes, second layer	4

Table 2: Estimated optimal hyperparameters found using randomised search for the extreme gradient boosted tree model.

Hyperparameter	**Optimal value**
Number of estimators	125
Learning rate	0.048
Maximum tree depth	9
Minimum child weight	3
L1 shrinkage parameter	0.0074
L2 shrinkage parameter	0.0035

Table 3: F_1 score for the training and test set for our k-nearest neighbour (kNN), multilayer perceptron (MLP) and extreme gradient boost (XGB) models .

F_1 score	**kNN**	**MLP**	**XGB**
Training set	0.88	0.87	0.95
Test set	0.83	0.87	0.88

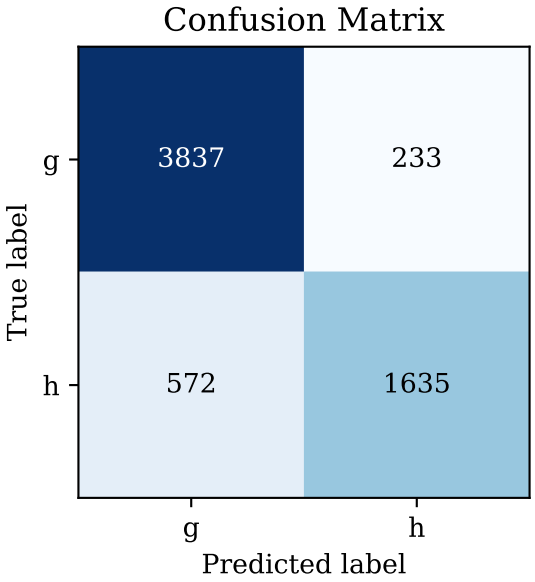

Figure 4: Confusion matrix of the neural network model applied to the test set.

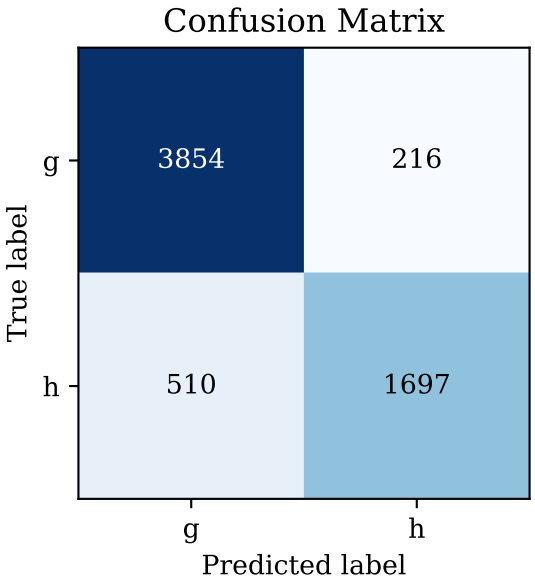

Figure 5: Confusion matrix of the gradient boosted model applied to the test set.

Table 4: Classification rates for the k-nearest neighbour (kNN), multilayer perceptron (MLP) and extreme gradient boosting (XGB) models.

	kNN	**MLP**	**XGB**
True positives	0.607	0.611	0.613
False negatives	0.041	0.037	0.034
False positives	0.125	0.091	0.081
True negatives	0.227	0.260	0.270
Error	0.166	0.128	0.116

the training and test set using this model can be found in Table 3. We see that the F_1 test score is lower than that of the training score, with values 0.83 and 0.88 respectively.

The optimal hyperparameters found using randomised search for the MLP model are shown in Table 1. The corresponding confusion matrix and the F_1 scores are shown in Figure 4 and Table 3 respectively. The F_1 score for the training and test set are both 0.87.

The results of the hyperparameter randomised search for the XGB model are shown in Table 2. Figure 5 shows the confusion matrix of the XGB model predictions and the F_1 scores for the training and test sets are shown in Table 3. The F_1 training score is 0.95, which is noticably higher than the test score at 0.88.

The true positive, true negative, false positive and false negative rates are shown in Table 4 for all three models along with the error rates.

When considering the F_1 scores in Table 3 for the three models , the highest test score value was obtained by the XGB model at 0.88, compared to 0.87 for the MLP and the 0.83. This corresponds to the values seen in the confusion matrices of the models in Figure 3, 4 and 5. Additionally, the rates of false positives and false negatives found in Table 4 show the lowest values for the XGB model, followed closely by the MLP, with the kNN model having the highest misclassification rates.

5 Discussion

Although the cross-validation procedure on the kNN model gave the value $k = 5$ as the optimal amount of neighbours to consider in the algorithm, the F_1 score in Table 3 still suggests that there might have been a model overfit. The F_1 score is lower for the test set, indicating that the predictive capability of the model is notably reduced on the test set compared to the training set. However, looking at the graph in Figure 3 of the validation F_1 score over k, it is not clear if using another k would have resulted in a better model. Generally, the kNN method performs well, but can be highly unstable.

The F_1 test and training scores in Table 3 of the MLP model are both 0.87, suggesting that the chosen hyperparameters in Table 1 did not lead to an overfitted model. Neural networks in general are capable of producing highly complex models and are consequently prone to overfitting. As this does not seem to have occurred here, it is possible we could have been able to find a better-performing model by tweaking the hyperparameters further to find more complex models with higher predictive power.

The XGB model F_1 test and training scores are 0.88 and 0.95 respectively. The large discrepancy in scores is indicative of an overfitted model, and is surprisingly the largest difference observed from our three models. Looking at the XGB model's hyperparameters in Table 2, a maximum tree depth of 9 might create overly complex models. Additionally, both the optimal values for the L1 and L2 shrinkage parameters are small at 0.0074 and 0.0035 respectively, which might suggest that setting both to zero and choosing to instead tune other hyperparameters would yield better results. Some of the candidate hyperparameters to tune in an attempt to reduce overfit could be to implement the subsample ratios of the features or samples used

to grow each tree. As the tree would be built on a different data set each iteration, the probability of overfitting to certain samples or features would be reduced.

Despite the discrepancy in F_1 training and test scores of the XGB model, the model seems to be the best-performing model in our case. Looking at the confusion matrices in Figure 3, 4, 5test and Table 4, the XGB method gave the lowest rates of false positives and false negatives, while also being the method with the highest rates of true positives and true negatives.

6 Conclusion

Acknowledgements

We want to thank the University of California, Irving for providing us with the data used in this project.

References

[1] Riccardo De Bin. STK-IN4300 Statistical Learning Methods in Data Science: Lecture 10. `https://www.uio.no/studier/emner/matnat/math/STK-IN4300/h19/slides/lecture_10_ho.pdf`, 2019. Retrieved: 09-12-2019.

[2] Dheeru Dua and Casey Graff. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`, 2017.

[3] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*, volume 27. Springer Series in Statistics, 2009.

[4] Dieter Heck, G Schatz, J Knapp, T Thouw, and JN Capdevielle. CORSIKA: a Monte Carlo code to simulate extensive air showers. Technical report, 1998.

[5] Bernhard Nornes Lotsberg and Anh-Nguyet Lise Nguyen. *Project 2: Classification and Regression*. FYS-STK3155/4155 Applied Data Analysis and Machine Learning. 2019.

[6] Cambridge Spark. Hyperparameter tuning in XGBoost. https://blog.cambridgespark.com/hyperparameter-tuning-in-xgboost-4ff9100a3b2f, 2017. Retrieved 11-12-2019.