Analysis of the Palmer penguins with decision trees and ensemble methods (Project 3 fys-stk4155)

Jørn-Marcus Høylo-Rosenberg, Andreas Dyve

17.December 2021

Abstract

With this project, we have used decision trees and numerous ensemble methods to predict the species of a penguin from exclusively it's physical features. From the help of metrics such as the accuracy score, confusion matrix, precision-recall curves and bias-variance tradeoff, we discovered firstly that a model with a single decision tree easily overfit the data, thus resulting in a high variance. By replacing the single tree with either a random forest or applying boosting methods such as AdaBoost and extreme gradient boost we were able to lower both the bias and variance of the model. In summary, we found that Adaboost provided the best solution to classify the minority class, while xgboost resulted in the overall highest accuracy score.

Introduction

In the previous projects in this course, we have explored linear regression methods such ordinary least squares, ridge and lasso regression. We have developed our own neural network for both regression and classification. We have studied exciting and popular datasets within the machine-learning world, for example the infamous Franke-function, the Wisconsin breast-cancer data, and we have performed linear regression on real-world terrain data over an area in Stavanger, Norway. But one question we never got the answer to, was how can machine learning be applied to predict penguin species? Well, in this project we set out to solve this question. In particular, this project will resolve around decision trees and a host of popular ensemble methods such as bagging, random forest, Adaboost and extreme gradient boosting to predict penguin species with the aid of the Palmerpenguins dataset.

H

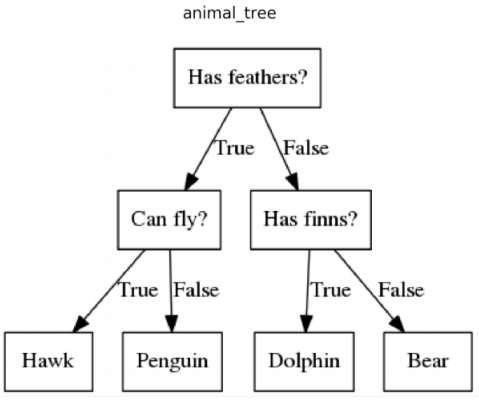


Figure 1: Decision tree example. From `https://towardsai.net/p/programming/decision-trees-explained-with-a-practical-example-fe47872d3b53`

Theory

The theory to be covered in this section is mostly gathered from the lecture notes of fys-stk4155, ch9 [10] and ch 10 [9] in combination with several excellent lectures from statquest on decision trees [5], random forest [14] and Adaboost [1].

Decision trees

A decision tree work very similar to the popular game 21 questions. The method begins by posing a simple questing, based on the outcome of this question, we then ask another question. In a decision tree this process is repeated iteratively in a binary split until a given condition is met. The fundamental operation and implementation of a decision tree is relatively straightforward and easily interpretative, in fact this is one of the biggest strengths of a decision tree. Bellow in figure 1 is the typical layout of this method for the task of classifying an animal based on if it has feathers and/or can fly.

The tree is built up of a root node, and connected through branches to so-called internal nodes and finally the leafs. In figure (1), the root node is the top node asking about feathers, the branches are the true/false responses. And obviously, the internal nodes are the middle ones, and the leafs are the final predictions "Hawk", "Penguin", "Dolphin", "Bear".

The first task of a decision tree is to determine which feature/question should be posed at the root node. To do this we first need to introduce the terms "impurity" and "purity". The impurity relates to the ratio of a binary split. A feature with high impurity will predict large amounts of both predictions. If

we go back to figure 1, for 10 samples, the impurity would be a maximum if the tree predicted 5 hawks, and 5 penguins. On the other hand, a pure leaf would for instance predict 10 hawks and 0 penguins. To quantify this impurity of leafs, a common method is the gini-impurity.

$$G_l = 1 - P_{yes}^2 - P_{no}^2 \tag{1}$$

Where P_{yes} and P_{no} is the probability of True or false. Keep in mind that for continues features, calculating the gini-index become slightly more complicated, but the overall procedure remains the same. We then calculate the gini impurity of both the right and left split, and add a weighted factor to account for the differences in sample sizes on both sides. Then, the feature with the overall lowest total gini-index directly correlates to the leaf with lowest impurities, thus this feature will adopt the root node position in the decision tree. We then repeat this process recursively for the internal nodes until we no longer can produce splits that reduce the impurity. The method described here is called the CART algorithm (short for Classification And Regression Tree), and is one of two primary methods alongside the ID3 for locating the optimal combination of attributes and threshold values (numeric features) to split the data in a decision tree. The final leafs we end up with will output the class with the most votes. Again, going back to our example with the animals, if for 10 samples we get 10 hawks and 0 penguins, the leaf will predict a hawk.

Ensemble methods

Despite the many upsides of a single decision tree, such as easy implementation and interpretation, the ability to solve regression and classification problems, and etc. There are several drawbacks with a decision tree. The most prominent being overfitting due to the tree attempting to fit the training data perfectly to reduce the impurity without any restraints or pruning. One of the most popular method of reducing overfitting in a decision tree is in fact to simply prune the tree. In addition, more advanced methods such as bagging and boosting that aggregate many simply trees, have proven to efficiently overcome the challenges posed from a single tree.

Random forest

As stated above, a decision tree would make for an excellent model if not for it's lack of flexibility. For this purpose, a random forest is the answer, combining the simplicity of a single decision tree, but with improved flexibility and hence ability to predict unseen data.

A random forest is essentially just a modification to what's known as bagging, short for bootstrapped aggregation. As the name suggest, in bagging the data is first bootstrapped (with replacement). For each bootstrap, we grow a decision tree as before. The process is then repeated for every bootstrapped dataset

up til a given total number of bootstraps. Finally, the end prediction is voted from the total ratio of observations/predictions from every tree (corresponding to the bootstrapped datasets), this is where the term aggregation stems from. In our example, this would correlate to the sum of predicted animal from every bootstrap. If 85 out of 100 bootstraps/trees predict a hawk, the final prediction would be a hawk.

The main drawback of bagging of decision trees is that if the data contain a strong predictor, virtually all trees will employ this feature at the root node and therefore the full-grown forest will consist of somewhat similar (correlated) trees. In a random forest however, a node can only be split according to a random subset of the total feature-space. Typically this value is equal to the square root of the amount of features. That means that in a dataset consisting of 9 features, the root node can only choose between 3 random features out of the total 9. Following, the internal nodes will then be based on 1 of 3 new random features.

The accuracy of a random forest can be evaluated using the out-of-bag error. This error is calculated from counting the correct/wrong prediction on the out-of-bag samples, meaning the samples that were excluded from the forest as a consequence of bootstrapping with replacement.

Boosting

As stated previously, boosting is a popular method to improve on the shortcomings of a single decision tree. The concept behind boosting is to apply a weak learner (or classifier) such as a very shallow tree that may only predict slightly better than a random guess at first, and through an iterative process form a strong learner. The iterative process comes down to growing sequential trees that correct the errors in the previous ones. The model is complete when the complete additive tree predicts perfectly or the maximum limit is reached. There are two primary methods in use today within the family of boosting methods, these are **Adaboost** and **gradient boost**, in this project we will employ an extremely popular and powerful descendent of gradient boost called **XGboost**, or extreme gradient boost.

Adaboost can be compared to the random forest method in that the end model is an aggregate of several trees, however in this case we simply employ decision stumps (trees with one node and two leafs) as the building block of the forest and weak classifiers in the model. The adaboost method works by assigning a weight to each data-point in the training set, initially all weights are equal. A decision stump is then grown for every feature on the weighted sample. As the stumps attempt to classify the samples based on one distinct attribute, the weight of each individual point will be adjusted based on if it was

wrongly (increase) or correctly(lower) predicted.

As stated, all samples are initialized with an equal weight, precisely, the weights are initialized as

$$w = \frac{1}{N} \tag{2}$$

where N is the total amount of training points. As the iteration progress, points with big weights is said to have a "large say" on the training set, and vice versa, points with low weight have a "small say". We quantify this value as the variable α to describe the amount of say a stump have on the final classifier.

$$\alpha_m = \frac{1}{2} ln \frac{1 - err}{err} \tag{3}$$

The "m" index on α correspond to the current classification in the range of a total of M classifications of the model. The term "err" is the total error of a stump, simply calculated by the ratio of miss-classified points in the training data.

$$err = \frac{1}{n} \sum_{i=0}^{n-1} I(y_i \neq G(x_i) \tag{4}$$

where I is equal to one for missclassified events, and 0 when correct. The variable $G(x_i)$ in the equation represent the weak classifier, or decision stump in this case. It's clear that when a stump produce a 100% accurate prediction, α is positive, and negative in the opposite case ($\alpha = 0$ for a random guess). We use this variable to update the weights of the training samples by

$$w = w_{i-1} \cdot \exp(\alpha) \tag{5}$$

when α is positive, the new weight will become smaller. And correspondingly, as seen from equation 4, a negative α increase the sample weight. In this way, subsequent stumps in the adaboost algorithm is trained to emphasize harder to classify samples. This has just been a overview of the key concepts that make up adaboost, please refer to for instance [17] or [9] for a much more thorough mathematical description of the method.

Gradient boosting Extreme gradient boost is an extremely powerful method responsible for some staggering results in the world of machine learning. In this report we will not dwell into the nuts and bolts of xgboost, but rather cover the theory behind plain gradient boosting, as this is the building block of xgboost. Gradient boost have several similarities to Adaboost, most prominent is the boosting process of transforming weak classifiers to a strong classifier. Whereas in adaboost, we assigned weights to the samples to aid the classifiers in what samples were difficult/easy to classify, gradient boost works by updating the classifiers through a gradient of the loss function. To describe this method we need a bit more mathematics, to follow is a quick mathematical description of gradient boosting.

Assume now that we have a loss function we want to minimize, given by

$$C = \sum_{i=0}^{n-1} \mathcal{L}(y_i, f(x_i)) \tag{6}$$

, where $f(x_i)$ is the function to model y_i. We now introduce a term $h_m(u_m, x)$ so that

$$f_M(x) = \sum_{m=1}^{M} h_m(u_m, x) \tag{7}$$

, where f_M is the final classifier, and the negative gradient vector equal to

$$\vec{u_m} = -\frac{dC}{d\vec{f}} \tag{8}$$

, evaluated at $f(x) = f_{m-1}(x)$. We apply these equations to update the next classifier as

$$f_m(x) = f_{m-1}(x) + h_m(u_m, x) \tag{9}$$

until we end up at equation 7.

The main difference between the two ensemble methods bagging and boosting is how the trees are grown. In bagging methods such as a random forest, the trees are grown independently in parallel to make a final aggregated forest-predictor. On the other hand, trees in boosting are grown sequentially based on the mistakes of the previous trees. Therefore boosting method will do a good job of modelling the dependencies between trees, and bagging the independence. We can relate this to the bias and variance of the model, and thus expect bagging methods to impact the variance of the model, while boosting method will do a good job of lowering the bias. [16].

Method

In this part we will briefly go through the selected dataset for this project and how the samples have been treated before implementation with decision trees. Then we will cover some of the methodology behind this project.

Dataset

The dataset that we will apply in this project to trial and error with decision trees and a host of ensemble methods is the Palmerpenguins dataset [8], made available by Dr. Kristen Gorman and the Palmer station. This dataset has become a popular replacement for the classic Iris dataset for predicting flower types. The Palmer penguins dataset consist of 344 samples containing three species of penguin, "Adelie", "Chinstrap", and "Gentoo". Each class is attributed to 8 features ranging from the Island they inhabit, the shape of their bill and body, and lastly gender. A brief header of the complete data can be seen bellow in figure 2.

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	male	2007
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	female	2007
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	female	2007
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN	2007
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	female	2007

Figure 2: Header of the entire Palmerpenguins dataset, note especially the NaN values in row 3. We are to predict the values listed in the "species" column. The dataset were imported, and handled using the pandas software [15].

In this project, we will only employ the numeric features to predict the species, this means columns 3,4,5 and 6 in figure 2. Thus the input data for this project will have shape (344,4). Of the total 344 studied penguins, 152 are "Adelie" penguins, 68 "Chinstrap", and 124 "Gentoo" penguins. Amidst all the samples, there are 2 penguins that have missing data regarding their body features, one of these can be seen in table 1. Given that there is only two samples, we saw it fit to simply remove these datapoints from the set. Additionally, seeing as these values did not correspond to the minority class, this should not have too much of an impact on the final predictions.

After sorting out the NaN elements, we divide the data into two separate lists containing the body features and species respectfully, and split it into training and testing data. A recurrent theme in this project is that the dataset is quite imbalanced between the classes, with almost 3 times as many "Adelie" penguins as "Chinstrap" ones. Hence, the splitting must be conducted thoughtfully, as a

training set unevenly distributed compared to the test set will result in error-prone results. For this we employ the strattifiedSplitter functionality of scikit-learn, this method aims to produce a constant class-ratio between the train and test data respectfully.

Lastly, there is the topic of encoding the categorical data. This is commonly a much-used approach when dealing with categorical inputs. There are several methods to encode the data, the two most common is label-encoding and one-hot encoding. The former is mostly used for ordinal data, ie categorical data that can be sorted in a ranging order, such as t-shirt sizes. The penguin species can obviously not be sorted in this manner, and therefore belong to the class of nominal data. For nominal data, one typically uses one-hot encoding [2]. One of the primary benefits of decision trees is their ability to work with both categorical and numeric data, however several popular packages demand specific data-formats and encoding, and we will be forced to one-hot, or even label-encode the data in these instances. Scikit-learn states that the "DecisionTreeClassifier" functionality demands specific encoding, but we did not note any errors without encoding, thus we kept the class labels when using scikit-learn. In addition, the same is true for the functionality we used to plot ROC and precision-recall curves, in which scikit-learn states that the targets must be binarized, but we did not experience any errors or warnings using the original data. These factors may thus contribute to errors in our results.

Methodology

The main structure of this project consist of 4 parts. Firstly, we prepare the data as discussed above. Secondly, attempt to predict penguin species from a decision tree model, thereafter by ensemble methods such as bagging, random forest, Adaboost and extreme gradient boosting (XGboost). Within all methods, we began by calculating the accuracy score of a completely fresh model without any tuning or optimization. Then we took to a grid search or simply a one-dimensional search for optimal parameters. In the case of a single decision tree, this comes down to the maximum allowed depth of the tree, and possibly the pruning parameter α for minimal cost-complexity pruning of the tree. In regard to ensemble methods, the important parameters is the maximum tree depth as before, and additionally the value n-estimators which is useful for specifying the amount of estimators(trees) in for instance bagging. When we are dealing with boosting methods, one must also specify the learning rate η, which is related to how the successive estimators is updated.

Decision tree, bagging, random forest and adaboost were all implemented using the functionality of scikit-learn [12]. For XGboost we uses the python library xgboost [4]. The code used to obtain the listed results, can be found at our github: `https://github.com/jornmarh/fys-stk4155/tree/main/project3`.

To evaluate the different methods, several metrics were in employment. Among them, the accuracy score, confusion matrix, ROC-curve, precision-recall curves and bias-variance tradeoff. In this report we favour the precision-recall curve over ROC, due to precision-recall superior ability to describe unbalanced datasets (ROC curves can be found at our github address). In terms of the precision-recall curve, precision refer to the result relevancy, and is defined as the number of false positives divided by the number of false positives + true positives [12]. Recall on the other hand is the ratio between number of true positives to the sum of true positives and false negatives. In short, a model with both high precision and high recall have both high amounts of true positives and low amounts of false negatives. If the two quantities is opposite, ie high precision and low recall, or low precision and high recall, the model return either accurate but few results or many incorrect predictions.

For a discussion on bias-variance tradeoff, please refer to our previous project at `https://github.com/jornmarh/fys-stk4155/tree/main/project1`. The implementation of bias-variance tradeoff is rather complicated when dealing with classification. Thankfully, the package mlxtend [13] comes to the rescue with the functionality bias-var-decomp(). For a regression case, calculating the bias and variance is a rather straight forward process, in which you simply calculate analytical expressions arsing from for example the mean squared loss function. For classification, it's a bit more complex. To solve this case, the function included in mlxtend uses the 0-1 loss function L. This function will output 0 for correctly classified labels, and 1 for incorrectly classified labels. To calculate the bias and variance for the 0-1 loss function, mlxtend use a main prediction factor $E|y|$, defined as 1 if the model predict a class one more time than 50% of the time, and 0 otherwise. If the main prediction does not equal the true class, the bias is set 1, and 0 otherwise. The variance is set equal to the probability that the predicted class and the main prediction does not match. Refer to mlxtend's official documentation for more details on how the bias-variance decomposition is performed.

Results

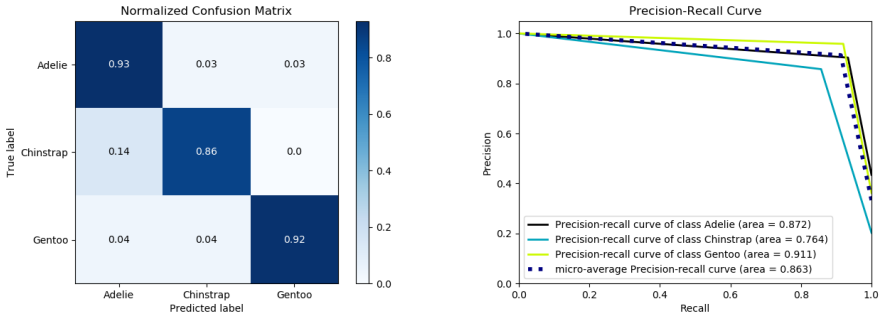
The results section in this project will be divided into four parts. First, we will present our results from a single decision tree, results range from accuracy score, confusion matrix, optimization and precision-recall curves, ROC curves can be found at our github. Then a similar analysis of bagging and boosting methods. Lastly, we do a bias-variance tradeoff comparison between all methods.

Decision trees

First, we display the accuracy score regarding a default tree (not specifying depth) and from one with cross-validated optimal parameters. In table 1 we find no gain from the optimized tree, we also note a sizable discrepancy between training and test accuracy.

	Default	Optimized
Depth	5	6
Pruning parameter	0	0.002
Accuracy	0.913	0.913
Accuracy (CV)	-	0.974
Training	-	1.0

Table 1: Summary of parameter optimization and accuracy score regarding a single decision tree when predicting penguin species from it's body features. The parameters relate to the total depth of the decision tree, and post-complexity pruning.



(a) Confusion matrix

(b) Precision recall curves

Figure 3: Confusion matrix (a) and precision-recall curves (b) from predicting penguin species with a single decision tree

Next, we look to the confusion matrix of the prediction to gain more insight into how the tree performs in classifying each species. The results from figure 3a is quite pleasing, seeing as there is not too large discrepancies between the classes, as might have been expected from the class count provided in the dataset. However, a clear missclassification error do exist for the chinstrap class of penguins as seen in the figure, noting an accuracy of 0.86, compared to 0.93 and 0.92 for Adelie and Gentoo respectfully.

Lastly, we include the precision-recall curves for all 3 classes. From the results illustrated in figure 3b, we observe a clear agreement with table 1, in that the chinstrap class is to a larger extent missclassified, compared to the more abundant classes. On the other hand, we also note that figure 3b points to a higher precision-recall tradeoff for the Gentoo class, in contrast to the accuracy depicted in table 1.

Bagging

In this section, we will look to comparing standard bagging and random forest aggregation of decision trees. Again, we begin by stating the accuracy obtained with the two methods in table 2, from the default implementation and our own attempt at optimizing the models. We observe that random forest and bagging both improve on the results obtained from a single decision tree, with random forest providing the overall best accuracy at 0.942. and 0.98 with cross-validation.

	default	Optimized	Bagging (optimized)
Accuracy score	0.942	0.942	0.928
Accuracy score (CV)	0.979	0.980	0.977
Training	-	0.996	1.0

Table 2: Accuracy score from random forrest and bagging for species-prediction, using a default random forest and after updating the parameters with a grid-search. In both cases of random forest, and bagging, a total of 100 trees was satisfactory.

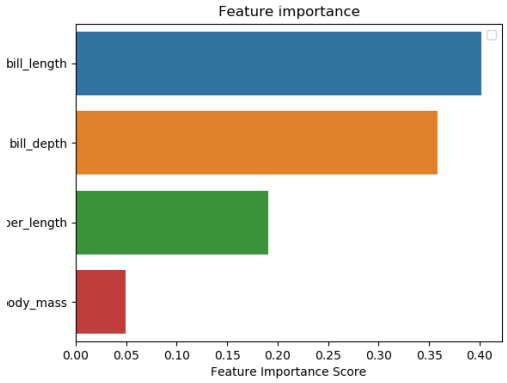
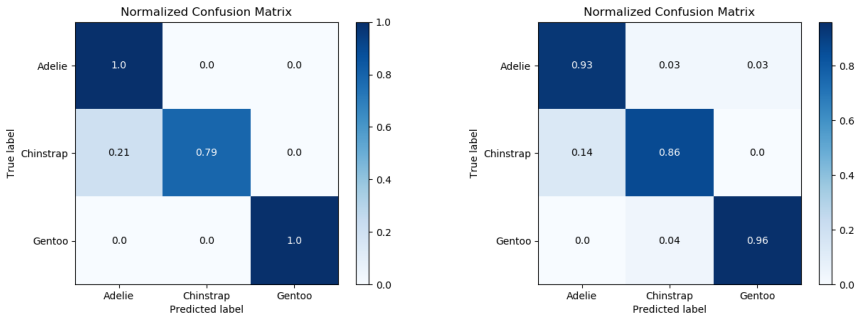


Figure 4: Feature importance when predicting penguin species, the features are "bill length", "bill depth", "flipper length", and "body mass". The data was gathered using a random forest.

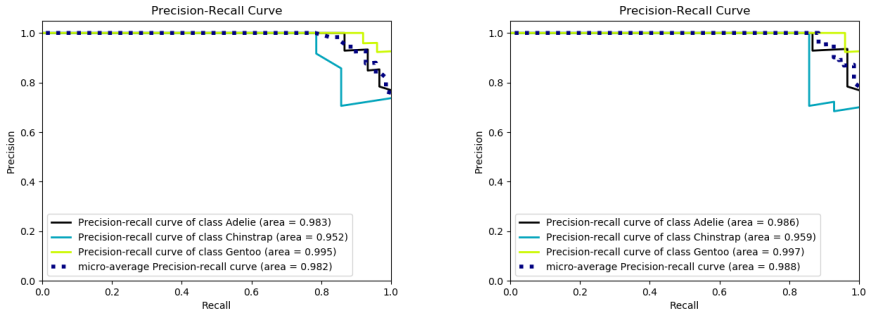
Above, in figure 4 we have included a feature importance plot, from the $feature - importance$ attribute of scikit-learns random forest implementation. As expected, when comparing to images of "Adelie", "Chinstrap" and "Gentoo" penguins (See palmerPenguins documentation), the bill dimension is the deciding factor in separating the penguins.

Moving on the confusion matrix, seen in figure 5 for bagging (a) and random forest (b). Bagging does an excellent classification of "Adelie" and "Gentoo" penguins, but struggle to correctly classify the "Chinstrap" class. Random forest on the other hand manage to a better extent a even prediction score over the three classes.



(a) Bagging (b) Random forest

Figure 5: The confusion matrix for both bagging and random forest when classifying penguin species.



(a) Bagging (b) Random forest

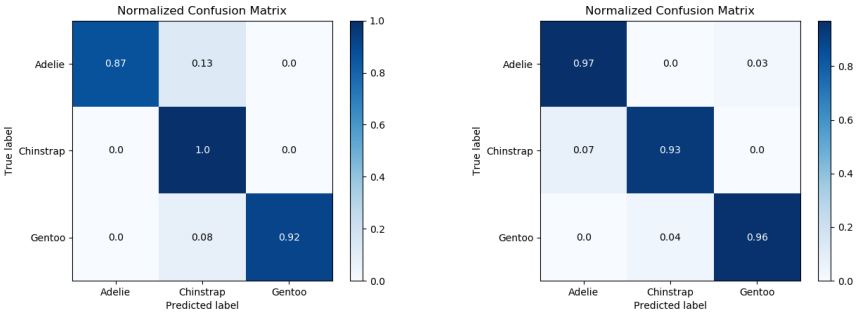
Figure 6: Precision-recall curve for both bagging and random forest when classifying penguin species.

The same result can also be observed in the precision-recall curves plotted in figure 6.

Boosting

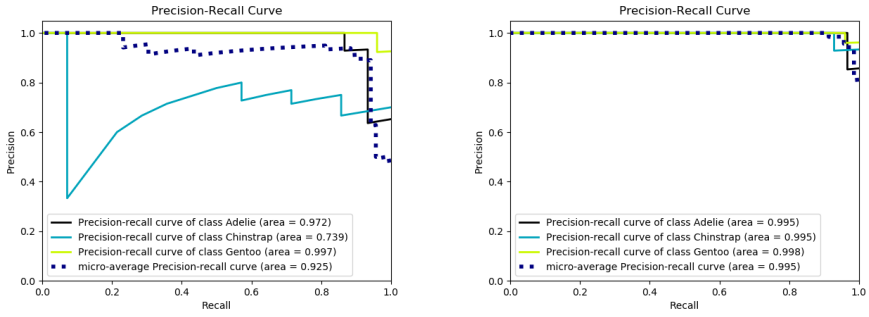
	Adaboost (default)	Adaboost (optimized)	XGboost (default)	XGboost (optimized)
Accuracy score	0.899	0.913	0.956	0.956
Accuracy score (CV)	0.804	0.956	0.974	0.980
Training	-	0.993	-	1.0

Table 3: summary of boosting ensemble methods for predication of penguin species. The optimized values refer to the number of estimators, and the learning rate of the estimators. For Adaboost this was 50 and 0.5 respectfully, and 100 and 0.05 for XGboost.



(a) Adaboost (b) XGboost

Figure 7: The confusion matrix for both Adaboost (a) and xgboost (b) for predicting penguin species.



(a) Adaboost (b) XGboost

Figure 8: Precision-recall curve for adaboost (a) and xgboost (b) from predicting penguin species.

Above, we have listed the key findings from predicting penguins species with Adaboost and xgboost. We see from table 3 that optimization of the adaboost model is essential, as we went from 0.80 to 0.96 percent accuracy with cross-validation, seen in table 3. When optimized, the adaboost method does a superior prediction of Chinstrap penguins, as seen in figure 7a. However this is at the cost at overall lower accuracy for the other classes, when compared to for instance xgboost. In fact, xgboost manage an impressive accuracy over all three classes, seen for instance in the confusion matrix figure 7b, or the precision-recall curves in figure 8b, resulting in a total 0.98 accuracy on the total set, matching that of the optimized random forest model with cross-validation, and producing the highest observed accuracy of the test set without cross-validation, at 0.96.

Bias-variance tradeoff

Finally, we will compare the bias-variance tradeoff of the numerous methods used in this project. Recall that all values and plots in this section were calculated using the $bias - var - decomposition$ functionality in the mlxtend library, refer to the methodology section for more details on this method. From the results plotted bellow (figures 9-11), we observe that generally the bias and variance does not depend greatly on the depth of the tree, provided that it's greater than 2. Furthermore we find from table 4 that all ensemble methods improve on the bias and variance of a single decision tree, with adaboost delivering the overall lowest bias, and surprisingly xgboost the overall lowest variance.

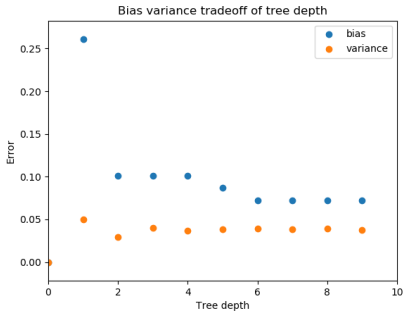
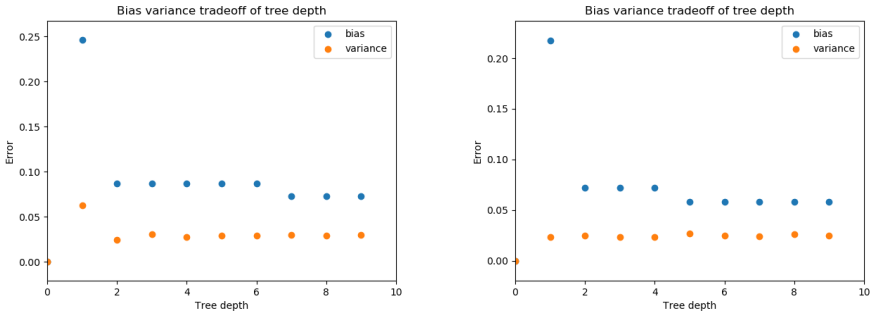
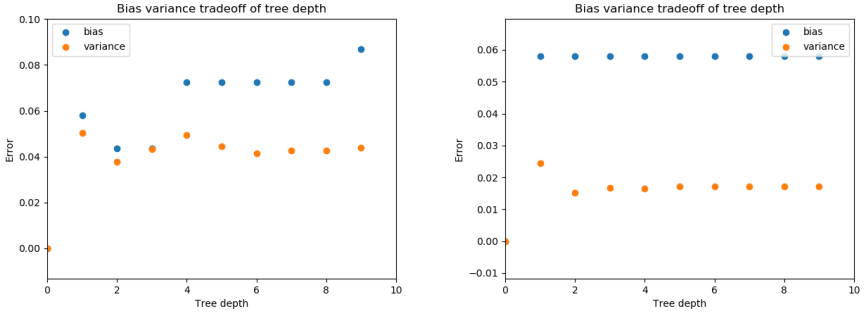


Figure 9: Bias-variance tradeoff for a single decision tree, as a function of tree depth



(a) Bagging (b) Random forest

Figure 10: Bias-variance tradeoff for bagging (a) and random forest (b), as a function of tree depth of the single trees



(a) Adaboost (b) XGboost

Figure 11: Bias-variance tradeoff for a Adaboost (a) and extreme gradient boosting (b), as a function of tree depth of the single trees

	Decision tree	Bagging	Random forest	Adaboost	XGboost
Bias	0.073	0.087	0.058	0.043	0.058
Variance	0.039	0.029	0.025	0.038	0.015

Table 4: The bias and variance of the various models tested throughout this project, all values were calculated using the "mlxtend" python package with 100 rounds, and random seed = 2021. The values seen in this table corresponds to the optimal value of maximum depth. max-depth = 6,7,2 for decision tree, (bagging + random forest), (Adaboost + xgboost)

Discussion

In this project, we first studied the success of a single decision tree to correctly classify penguin species from it's physical features. The first factor we would like to point out is the use of cross-validation. As seen in table 1, there is a large gap between values with and without cross-validation. This largely comes down to the fact that decision trees are very sensitive to the input data. Additionally, with cross-validation every single datapoint is used in both training and testing of the model, as opposed to standard calculations that make use of a predetermined train and test split. With the implementation of cross-validation, we are also unsure if the training and test data are stratified, as discussed in the dataset section.

In this regard, we also note that the optimized parameters we found with cross-validation does not improve the accuracy obtained with scikit-learns default tree. We relate this result to the fact that scikit's implementation is initially very good, without specifying any parameters the tree is allowed to grow until the impurity condition is met. The only possible downside of this is that a tree that is allowed to perfectly fit a training sample, is prone to overfit the data. From table 1, we observe that the training data is predicted 100% accurate in comparison to 91% on the test data, clearly indicating that the tree has overfit the training data. In spite of our efforts, we were not able to reduce overfitting by specifying the tree depth (between 1 and 10), or pruning the tree.

From the bias-variance tradeoff depicted in figure 9, the variance is at a minimum at tree depth equal to 2, but the bias is much lower than it's optimal value at tree depth equal to 6. When comparing with the results in table 3, we see that as expected, bagging and particularly random forest improve on the accuracy score from a single tree. From table 4 we can relate this improvement to the almost halving in variance when moving from a single tree to a random forest with 100 trees. Furthermore, with boosting method we were able to lower the bias further with Adaboost, and the variance with xgboost.

A high bias and low variance would mean that the model predicts well, but varies largely depending on the inputs. Similarly, a high bias and low variance would result in consistent bad results regardless of input [11].

If we think of a model with high bias and low variance to predicts well, varying depending on the inputs. Similarly, a high bias and low variance result in consistent bad results regardless of input [11]. From the theory section on decision trees, bagging and boosting, recall that a decision tree is a model that predicts well, but varies largely from input to input. And that a bagging method such as random forest is simply put an aggregation of several decision trees. While boosting methods employ so-called weak learners, shallow decision trees in our case, that predicts consistently poorly. Relating this back to the definition of bias and variance, one would therefore expect a random forest to initially have

low bias, and attempt to lower the variance through aggregation. A boosting method on the other hand would initially start with a high bias, but a low variance, and attempt to lower the model bias.

As a final note on the bias-variance tradeoff, we would like make a point out of several factors that may have affected the results listed in figures 9-11 and table 4. Firstly, we calculated the bias-variance tradeoff as a function of tree depth. From for instance figure 9, it's clear that maximum depth in the range of 1-10 is not sufficient, or simply ineffective in our case and implementation, as we never encounter clear overfitting by an increase in the variance. We could have attempted a larger range of values, but due to large computational times in some of the more advanced ensemble methods we opted for a consistent smaller baseline for the sake of comparison. Secondly, modeling the bias and variance after tree depth is non-sensible for boosting methods, per definition, these methods employ weak classifiers. Again as a result of large computation time, we were also forced to limit the bias-variance function to only 100 rounds. In addition, the implementation from "mlxtend" demand the target data to be label-encoded, for nominal data such as the penguins this is not a proper way of encoding the data, therefore these result are not necessary 100% valid. Which may serve as an explanation for some of the abnormalities such as the unusual high bias in low variance in xgb, or high bias in bagging.

Another point to overfitting, is the degree of fitting one the different classes. In the dataset section, we discussed the imbalance and class count in the Palmer-penguins set. In which we discovered that there is a significant lack of representation of Chinstrap penguins compared to Adelie and Gentoo penguins. From this starting point, it's reasonable that the models will do a better job at predicting these majority classes based on the shear number of training instances. As seen in figures 3(decision tree), 5 (bagging and random forest), and 7(Adaboost and xgboost) this is clearly shown, especially for bagging in figure 5a, with a 100& accuracy on the two majority classes and only 79% on the Chinstrap class.

From comparing the confusion matrices of different methods, we find that adaboost is the far superior method for classification of the minority class. Going back to the working principle of Adaboost covered in the theory section, we recall that this method grow trees based on samples that are the most difficult to classify. This seems to be good agreement with what we see in figure 7, where the minority class is predicted with 100% accuracy (heavily prioritized), and the two majority classes are assigned small weights (less prioritized) and result in lower accuracy. In regard to the adaboost method, the precision-recall display a very strange behaviour compared to that of other models, this give rise to some suspicion. But, given limited experience with both precision-recall analysis and boosting methods in general, not to mentioned all the possible errors that may be present and affect the displayed results, it's difficult make a firm statement on the topic. On the other hand, we generally find good agreement between confusion matrices and precision-recall curves for the other methods. Such as in

figure 7b and 8b, both displaying overall good classification over all classes for xgboost. And in figure 5a and 6a, both in agreement that bagging to a lesser degree is able to classify Chinstrap penguins. But, also in this case there seem to be some dissimilarities, as the curves for Adelie and Gentoo penguins does not correspond to the 100% accuracy depicted in the confusion matrix.

Between the results of Adaboost and xgboost we see from table 3 that xgboost provide an overall better accuracy. With somewhat consistent predictions over all three classes. However, due to Adaboost's exceptional ability to classify the minority class, it may be argued that Adaboost should be the favoured method for this dataset. Especially considering the model optimization, we see from table 3 that adaboost is very sensitive to hyper-parameters. Maybe given more time we could do a broader search over more variables to bring the results of adaboost closer to that of xgboost. To conclude, in this project, as is the case in several projects in the literature of machine learning, for instance [7], [6] and [3], also our project end up with the conclusion that boosting provide the overall best classification. This is in no-way a general statement regarding decision trees, bagging and boosting ensemble methods, but serve as a compliment to boosting methods alone.

Conclusion

Throughout this project we have realized our dream of predicting penguins with machine learning. We have been through decision trees, bagging, random forest and boosting. From the basic accuracy score of the different methods we find that random forest and extreme gradient boost performed the best. Studying the bias- variance tradeoff we discover that this in large part relate to the reduction in variance. And when evaluating the confusion matrix of both, we find that xgboost does a better job at handling the imbalanced class count of our dataset. Comparing the different methods, such as standard bagging to random forest, it's clear that a small modification/improvement result in a sizable accuracy gain, and similarly for Adaboost and xgboost. From this project we have gained a greater understanding of decision trees and ensemble methods. Furthermore we have gained valuable experience with machine learning in the real world, from implementing popular python libraries and dealing with real-world datasets.

References

[1] *AdaBoost, Clearly Explained*. URL: `https://www.youtube.com/watch?v=LsK-xG1cLYA`.

[2] Alakh Sethi. *Categorical Encoding — One Hot Encoding vs Label Encoding*. en. Mar. 2020. URL: `https://www.analyticsvidhya.com/blog/`

2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/ (visited on 12/17/2021).

[3] Nurheri Cahyana, Siti Khomsah, and Agus Sasmito Aribowo. "Improving Imbalanced Dataset Classification Using Oversampling and Gradient Boosting". In: *2019 5th International Conference on Science in Information Technology (ICSITech)*. Oct. 2019, pp. 217–222. DOI: 10.1109/ICSITech46713.2019.8987499.

[4] 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*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[5] *Decision and Classification Trees, Clearly Explained!!!* URL: https://www.youtube.com/watch?v=_L39rN6gz7Y.

[6] Thomas G Dietterich. "An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization". en. In: (), p. 19.

[7] Harris Drucker and Corinna Cortes. "Boosting Decision Trees." In: vol. 8. Jan. 1995, pp. 479–485.

[8] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. R package version 0.1.0. 2020. DOI: 10.5281/zenodo.3960218. URL: https://allisonhorst.github.io/palmerpenguins/.

[9] Morten Hjorth-Jensen. *10. Ensemble Methods: From a Single Tree to Many Trees and Extreme Boosting, Meet the Jungle of Methods — Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter7.html (visited on 12/17/2021).

[10] Morten Hjorth-Jensen. *9. Decision trees, overarching aims — Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter6.html (visited on 12/17/2021).

[11] Patrick Chao. *SAAS - Education Committee*. URL: https://saas.berkeley.edu/education/bias-variance-decision-trees-ensemble-learning (visited on 12/17/2021).

[12] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[13] Sebastian Raschka. "MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack". In: *The Journal of Open Source Software* 3.24 (Apr. 2018). DOI: 10.21105/joss.00638. URL: http://joss.theoj.org/papers/10.21105/joss.00638.

[14] *StatQuest: Random Forests Part 1 - Building, Using and Evaluating*. URL: https://www.youtube.com/watch?v=J4Wdy0Wc_xQ.

[15] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[16] Vihar Kurama. *A Guide To Understanding AdaBoost*. en. Feb. 2020. URL: https://blog.paperspace.com/adaboost-optimizer/ (visited on 12/17/2021).

[17] Wikipedia contributors. *AdaBoost*. en. Page Version ID: 1056566624. Nov. 2021. URL: https://en.wikipedia.org/w/index.php?title=AdaBoost&oldid=1056566624 (visited on 12/17/2021).