FYS-STK4155: Project 3

Simen Nyhus Bastnes

14. December 2020

Abstract

In this project, we want to take a look at audio classification using different machine learning algorithms. The data set we will be looking at is a set of labeled audio files containing either cat or dog sounds. To reduce the complexity of the data set, we transform it into Fourier space, using the binned frequencies as our predictors. Principal component analysis (PCA) is employed to further reduce the predictor set. The classification methods we will be using is a feed-forward neural networks, random forests, and XGBoost. Cross-validation will be used in order to find the optimal hyperparameters. We find that neural nets and random forest classify the samples with an accuracy of around 90% for both the normal data set and the reduced data set, with a slight reduction in the PCA reduced results. XGBoost performs the best with almost 95% accuracy on the full data set, and 93% on the reduced. Further, we find that the neural net is better at classifying dogs than cats, while random forests and XGBoost classifies cats better.

1 Introduction

While image recognition with machine learning seems to have taken the spotlight in recent years when it comes to popularity, using machine learning for audio analysis is a field with a plethora of real world applications, both for supervised and unsupervised learning. Everything from speech recognition to make the lives of deaf individuals easier, to studying seismic data to detect earthquakes, or detect wear in critical machinery at a factory based on the sound or vibrations it makes.

In this project, we will be looking at a fairly simple supervised approach for classifying different audio events, namely looking at the Fourier transform of our audio samples. Under the assumption that different types of sound events has different spectral characteristics, we extract a number of features from each sample by binning the frequencies into 40 Hz bins. The methods used for the classification is a standard feed-forward neural network, random forests, and the XGBoost gradient boosting.

Since it is unlikely that all the different frequency bins are necessary for our analysis, we also test a common dimensionality reduction method, the principal component analysis (PCA), in order to see how much it affects both the results and run time of our code.

The data set we will be using is a set of audio files with cat and dog sounds from [1], consisting of 277 samples. As the data set is quite varied in terms of content and quality, this serves as a decent real world test case. First, in Chapter 2 we will go through the relevant theory, going briefly through feed-forward neural networks, and then decision trees/random forests, gradient boosting and finally PCA. A more in-depth description of the data sets can be found in Chapter

3. Then, in Chapter 4 we go through the results of performing the analysis on both the full data set as well as the dimensionality reduced one, while discussing them. Finally, we conclude our findings in Chapter 5.

2 Theory

In this chapter we will be going through the fundamentals of the classification methods we will employ, as well as the dimensionality reduction and different metrics we use to gauge how well our models work. For the discussion on decision trees and random forests, we will follow chapter 6 and 7 of [2].

2.1 Feed-forward neural network

In this report, we will be using feed-forward neural networks to compare our results with the random forest and gradient boosting methods. A description of how FFNNs function and is trained can be found in [3]. Instead of using the neural net we created in project 2, we will be using Scikit-Learns `MLPClassifier` [4]. In addition to the hyperparameters we tested previously in [3], we will be testing the different solvers SKL has for weight optimization. Besides the stochastic gradient descent, `MLPClassifier` has a solver in the family of quasi-Newton methods, which is claimed to converge faster and perform better for small datasets.

2.2 Decision trees

Decision trees is a powerful machine learning algorithm that can be used for both regression and classification. One of the strengths of decision trees is the inherent simplicity when it comes to when it comes to understanding how it works. Unlike ANNs or other black box models, decision trees are white boxes, making it easy to interpret why the model makes the prediction. Decision trees are also the fundamental building blocks for both of the ensemble methods (random forest and XGBoost) that we will use later.

The base concept of decision trees is that the data set is split into two based on some question/"decision", which can for example be whether or not a person is male or female, or if their height is above or below some 1.7 meters. This process is repeated until all samples are correctly classified or you reach some specified stopping criteria.

Figure 1 shows an example of a decision tree attempting to classify iris flowers based on the sepal and petal length/width. The starting point is the *root node* (depth=0), and the method tries to find the question that separates the data the best. In this case, the question whether or not the petal width is less than or equal to 0.8 cm separates the data best. The left child node (at depth=1) is *pure*, as only samples of a single class is included. Thus, this is a *leaf node*, and there is no reason to split further. The right child node only correctly classifies 50% of the samples, and needs to be branched again. With a split at petal width $<= 1.75$ cm, the tree separates the data almost fully.

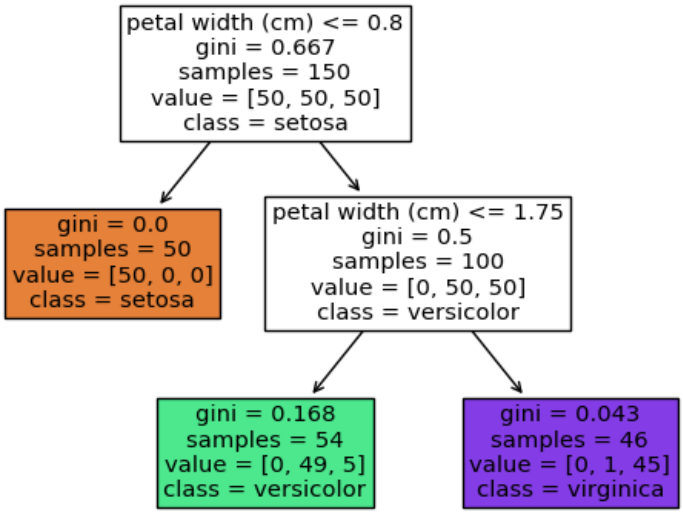


Figure 1: Decision tree example using the Iris data set. Corresponds to figure 6-1 in [2].

In order to quantify how pure a node is, we compute the *impurity* of each of the nodes. One of the common ways to do this for classification trees is the so-called Gini score/impurity

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

where $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node. A fully pure node will have a gini score of 0.

To train/grow the decision tree, we will look at the CART[1] algorithm as it is the one that Scikit-Learn uses. The algorithm splits the data set into two subsets using a single feature k and a threshold t_k, and searches for the pair of (k, t_k) that yields the purest subsets. The cost function that it tries to minimize is the following

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $G_{\text{left/right}}$ is the impurity of the left/right subset, and $m_{\text{left/right}}$ is the amount of instances in the left/right subset, meaning that the impurity is weighted by the size of each subset. It then splits each subset in two following the same logic, continuing until it either cannot find any splits that decrease the impurity, or some stopping criterion is met. One of these criteria is the maximum depth, limiting how deep the tree can go. Some of the hyperparameters that Scikit-Learn has that can be used to tweak how the tree grows is

[1] "*Classification and Regression Tree*"

- `max_depth`: maximum depth of the tree
- `max_features`: maximum amount of features to consider when looking for the best split
- `max_leaf_nodes`: maximum number of leaf nodes
- `min_samples_split`: minimum number of samples required to split
- `min_samples_leaf`: minimum number of samples required to be a leaf node

As decision trees make very few assumptions about the training data, it can easily end up overfitting, and give very poor results for the test data. Thus, using the different hyperparameters discussed above can help regularize the model and reduce the overfitting. Decision trees are also very sensitive to small variations in the training set, making just small variations in the train/test split enough to change your model. This brings us to the next topic, ensemble methods, which attempts to limit this instability aggregating multiple predictions.

2.3 Ensemble methods

Ensemble learning is a machine learning method where you combine the results of multiple methods to obtain better predictive power than each of the methods alone. While you could combine the results from all kinds of different machine learning algorithms, we will focus on methods based on decision trees, namely random forests as well as look at gradient boosting via the XGBoost library.

With ensemble methods even if the classifiers are weak learners (as in a classifier that is only slightly better than random guesses), the ensemble can still be a strong learner, provided there is a large enough and diverse set of classifiers.

2.3.1 Random forests

The Random Forest is an ensemble method where we train a set of decision trees, and combine the predictions from all the trees to get our final predictions. In order to make our ensemble method as good as possible, we can train each of the decision trees on a different random subset of the training data. Using bootstrap for this is referred to as *bagging* (bootstrap aggregating), while sampling without replacement is referred to as *pasting*.

When making the predictions, one can use either *hard* or *soft* voting. Hard voting is simply majority rule, where the class that has the most predictions/votes is the final prediction. Soft voting on the other hand, takes the average of the prediction probabilities for each of the decision trees, and makes the largest one the final prediction. This can give better results than hard voting as it gives more weight to highly confident predictions.

The random forest has mostly the same hyperparameters as decision trees, but also how many decision trees to grow, as well as other ensemble parameters. Another benefit with random forests is that instead of searching for the best feature to split at for each node, one can consider a random subset of the features, resulting in greater diversity among the trees and lower variance, producing better results.

2.3.2 Gradient boosting / XGBoost

Gradient boosting is another ensemble method where you sequentially add more predictors to the ensemble, with each one correcting the predecessor. The way this is done is that we initialize

an estimate to our targets. Given a cost function, we compute the negative gradient vector with respect to the previous prediction. Then, we fit our model to the negative gradient, and update the estimate by adding the gradient to the previous estimate. This is done for however long we want, and return our final estimate.

In this report, we will be using the XGBoost library [5] to perform gradient boosting. XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It has become an extremely popular library, and has been used to win many machine learning competitions.

2.4 Principal component analysis

At its core, dimensionality reduction involves transforming data to a space with lower dimensionality than the original. Ideally, this new representation of the data retains as much of the information in the data set as possible, however some loss will always occur. Technically, when defining our data set, as described later in Chapter 3, we perform a somewhat arbitrary dimensionality reduction when binning the audio frequencies, hoping it works okay. To further reduce the number of dimensions, we employ a much used dimensionality reduction method, namely the principal component analysis (PCA).

Principal component analysis works by finding the hyperplane that lies closest to the data, and then projects the data onto it. First, it finds the axis that accounts for the largest amount of variance in the data (the first principal component). Then, the second is the axis orthogonal to the first that maximizes the variance, and so on for the amount of dimensions in the data set (or for as many dimensions you want to compute / reduce the data set to).

To compute the principal components, one can for example use singular value decomposition (SVD) to decompose the data matrix into

$$\mathbf{X} = U\Sigma V^{\mathsf{T}}$$

where V contains the principal components. The data set can then be projected down to D dimensions by transforming $\mathbf{X}$ using the first D components of V

$$\mathbf{T}_D = \mathbf{X}V_D$$

When determining the amount of dimensions to reduce down to, we can determine how much of the variance to keep, and then set D as the number of principal components needed to retain for example 95% of the data set variance.

One of the issues with this method of performing PCA is that the entire data set $\mathbf{X}$ needs to fit into memory, which while is not a big problem for our data set, can be difficult with larger data sets. To circumvent this issue, one can for example use an incremental PCA (IPCA) algorithm where the data set is split into mini-batches, or a stochastic approach where an approximation of the principal components is found.

2.5 Performance metrics

To measure the performance of our classifiers, we use a couple of different metrics. First of all, the **accuracy** gives us the number of correctly guessed targets t_i divided by the total number of

targets.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}$$

where I is the indicator function, and y_i is the output from our classifier.

For a 2-class classification problem, the correct guesses (for a positive result) can be expressed as the true positive (TN), while the samples incorrectly guessed as positive is the false positive (FP). Likewise with negative results, we have the true negative (TN) and false negative (FN). With this, the accuracy can be written as

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{n}$$

where n is the total amount of samples. To more closely study how our model performs for each class, we plot the **confusion matrix**

$$\text{confusion matrix} = \begin{pmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{pmatrix}$$

for practicality, we can normalize each row, making the values the ratio of true and false positives/negatives. Finally, we can plot the true positive rate (TPR)

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

against the false positive rate (FPR)

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

at various thresholds to get the so-called **receiver operating characteristic** (**ROC**) curve. The area under the ROC curve (**AUC**) is often used for comparing different methods, but it is worth keeping in mind that it is not a perfect estimator, especially not for small sample sizes [6].

3 Data sets

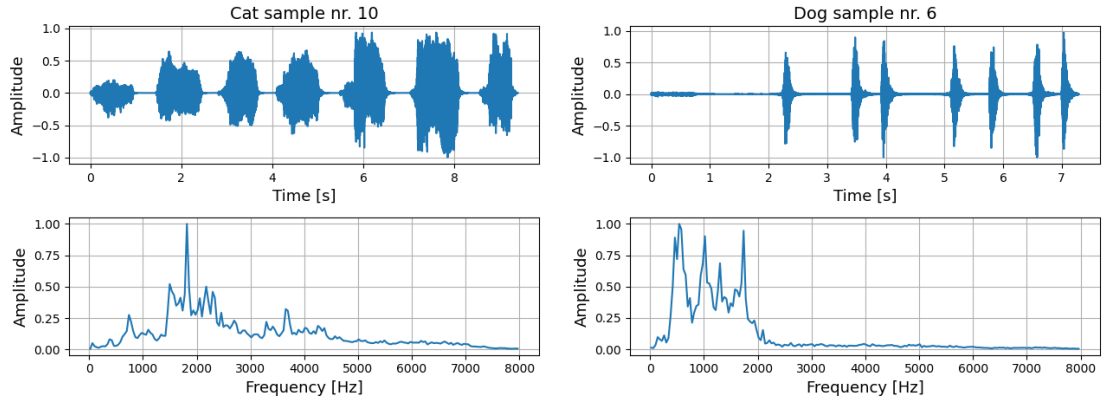
The data set we will be using in this project is a data set consisting of 277 audio files of various cat meows and dog barks, taken from [1]. There are 164 files labeled cat, containing 22 minutes of audio, and 113 files labeled dog, containing 10 minutes of audio. The audio files are originally harvested from `freesound.org`, as described in [7], which leads to quite big differences between the samples, both in length and quality (however all files are 16 KHz). Since the samples were recorded with different types of devices under different conditions, the noise level varies between the samples, and some of them contain other events such as footsteps, or for example many dogs barking at once. Some of the samples are also apparently synthesized. This gives us an opportunity to gain some insight into how machine learning algorithms manages smaller and less robust data sets.

Given the sound files, we need to define how the data will be represented when fed into our classification methods. When it comes to sound files, there are three main approaches one could take, working in either the time domain, the frequency domain, or even wavelet domain. Depending on the specific methods used for the analysis, different approaches may be more optimal.

For example, using recurrent neural networks [8]. In wavelet space one could for example use a convolutional neural network to classify based on the spectrogram images. In [7] convolutional neural networks was used on a larger audio data set (of which our data set is two of the different classes of audio events studied), achieving roughly 93% accuracy.

However, in order to keep this report as a simple demonstration, we will restrict ourselves to the frequency space, making a simple data matrix with the frequencies as our predictors. Since the nearby frequencies are highly correlated, we bin the frequencies, reducing the size of the data set. The Fourier transform of each of the files are computed, and the the frequencies are binned into 200 bins of size 40 Hz[2].

Figure 2 shows the time and frequency signal for one of the cat and one of the dog samples. We see that for at least this dog sample, the dominating frequencies are a lot lower than for the cat, which is the kind of difference we hope our methods will be able to detect. One potential complication here is the frequency differences between dogs of different sizes, as the pitch between them could potentially make it harder to accurately predict dogs, especially when there are less of them sample wise.



(a) Cat sample nr. 10

(b) Dog sample nr. 6

Figure 2: Time and frequency signal for one of the cat (left panel) samples, and one of the dog (right panel) samples. The amplitudes are normalized.

We also use PCA to create a dimensionally reduced data set to test our classification methods with. As the data set is fairly small, there is no problem using the full PCA, and we perform the reduction by keeping 95% of the variance. This reduces our 200 frequency bins down to only 35 predictors, giving an 83% reduction in the dimensionality of our data set.

4 Results and discussion

In this section we will go through the results from analyzing the data set described in Chapter 3, using different classification methods and dimensionality reduction. The code used to generate the results can be found in the GitHub repository [9], and is implemented using Scikit-Learn [10]

[2]The number of bins and bin size here is chosen somewhat arbitrarily.

in order to be able to parallelize the hyperparameter search[3]. Cross-validation is employed during the hyperparameter search in order to more properly evaluate the results.

4.1 Neural networks

We find our optimal hyperparameters for the neural net by performing grid search using the hyperparameter sets found in appendix A. For simplicity we set all hidden layers to have the same amount of nodes, however a more proper analysis would be useful. When trying the different solvers discussed earlier, it proved difficult to find any parameters where the stochastic gradient descent managed to converge properly. This was however not a problem with the quasi-Newton solver, so all the results presented here uses the `lbfgs` solver.

4.1.1 Full data set

Table 1 shows our best-fit hyperparameters for the neural net. Some of the hyperparameter heatmaps can be found in appendix A, figure 15. We note that the logistic function gives the best results, with tanh being the second best. In the previous project [3], we found that the ReLU activation performed the best for regression, and the worst for classification. Very curious is the fact that the best model is one with only one hidden layer, containing only one neuron. Looking at panel (d) in figure 15 we see how the results change depending on the depth and width of the neural network.

Table 1: Best-fit hyperparameters for feed-forward neural network. Full data set.

Parameter	Value
n hidden neurons	1
n hidden layers	1
activation	logistic
λ/α	0.0001
max iterations	600

Figure 3 shows the confusion matrix for the best-fit neural net, showing that the model performs quite a bit better at classifying the dog samples (96%) compared to the cat samples (88%). This gives us a total accuracy of 0.911, which is still good.

[3]In the previous project [3], one of the grid searches took over 15 hours. Even though this data set is smaller, allowing the code to use more cores/threads (in the authors case, 12 threads) should reduce the run time significantly, allowing us to evaluate a larger part of the hyperparameter space.

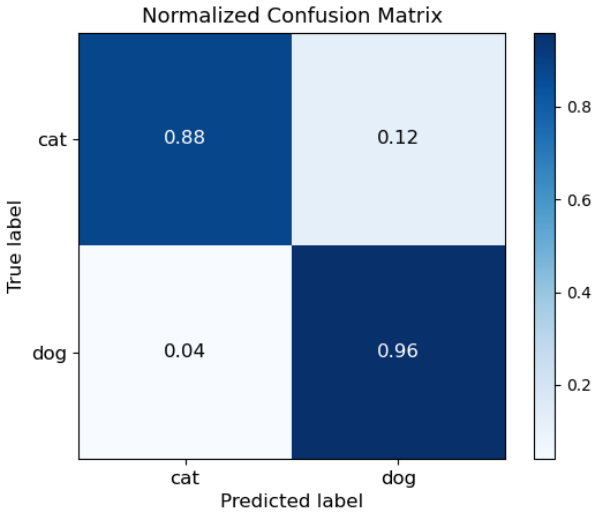


Figure 3: Confusion matrix for the neural net, full data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 4 shows the ROC curve for the best-fit neural net. We see that both of the class curves are good, with the same AUC score.

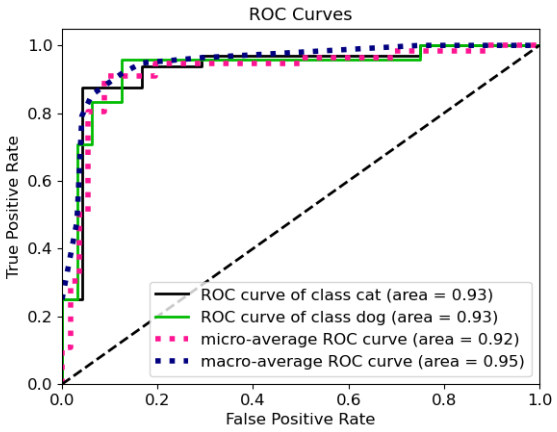


Figure 4: ROC curve for the neural net, full data set. The AUC is given in the legend.

4.1.2 PCA reduced data set

For the PCA reduced data set, we run with the same hyperparameter sets as earlier, and get the following best-fit parameters, as shown in table 2. Some of the hyperparameter heatmaps can be seen in figure 16. This time, a larger regularization strength is chosen, and we see that the best model contains a few more hidden neurons.

Table 2: Best-fit hyperparameters for feed-forward neural network. PCA reduced data set.

Parameter	Value
n hidden neurons	5
n hidden layers	1
activation	logistic
λ/α	0.1
max iterations	400

Training our neural net with the best-fit parameters we get the following confusion matrix for the PCA reduced data set, shown in figure 5. Here, we see that the true positives remain the same (maybe slight improvement as blue color is a bit darker), while the model loses 4% at classifying dogs. This could be due to the larger probable variation in frequencies used by dogs of different sizes, and thus when reducing the dimensions, the information lost affects dogs more. The disparity in sample sizes for both classes could also affect this to some extent.

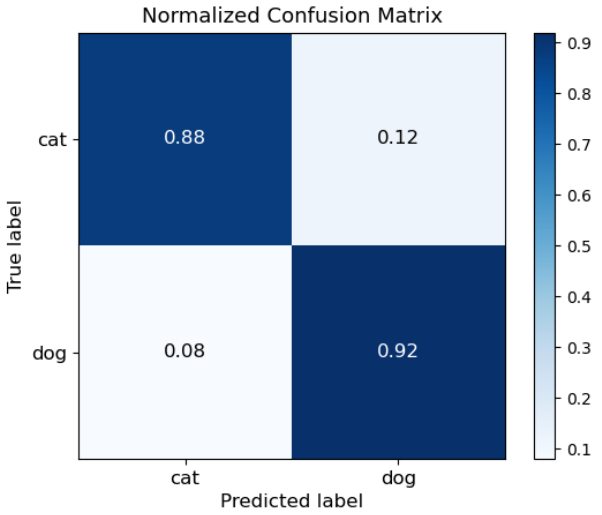


Figure 5: Confusion matrix for the neural net, PCA reduced data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 6 shows the ROC curves for the best-fit neural net. Similar to the full data set, it is hard to tell anything beyond that the models seem to perform well. The AUC is slightly higher than before though, despite the total accuracy for the model decreasing slightly.

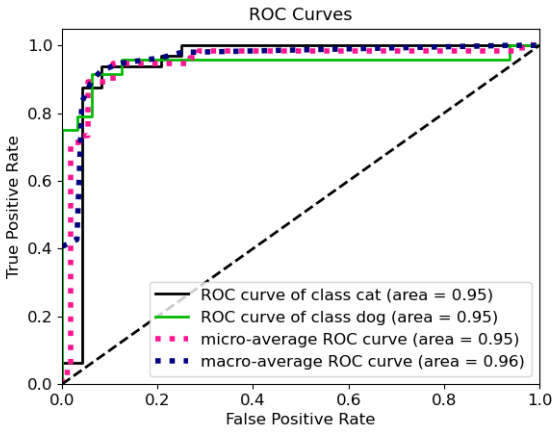


Figure 6: ROC curve for the neural net, PCA reduced data set. The AUC is given in the legend.

4.2 Random forests

We find our optimal hyperparameters for random forests by performing grid search using the hyperparameter sets found in appendix B.

4.2.1 Full data set

Table 3 shows our best-fit hyperparameters for random forests. Some of the hyperparameter heatmaps can be found in appendix B, figure 17.

Table 3: Best-fit hyperparameters for random forests. Full data set.

Parameter	Value
n estimators	15
max depth	7
min samples split	2
min samples leaf	5
max features	7

Figure 7 shows the confusion matrix for the best-fit random forest. Here, the results are almost inverse of what was found for the neural net. The random forest performs extremely well at classifying cat samples, at 97% accuracy, while dogs end up with only 83%. This cat accuracy comes at the cost of dogs, leading our model to guess dogs more often, as seen in the false negatives.

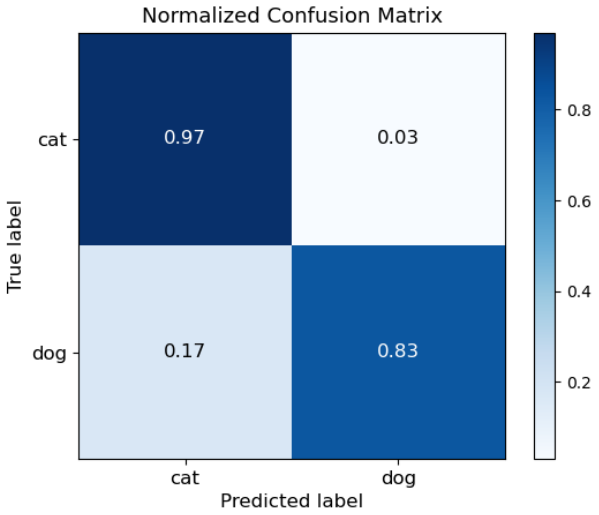


Figure 7: Confusion matrix for random forests, full data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 8 shows the ROC curve for the random forest. The AUC is the same for both classes, so either something is wrong with our implementation, or it really is as mentioned earlier quite problematic for evaluating model fit.

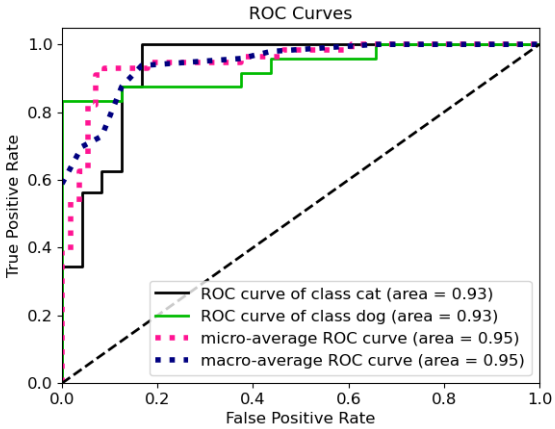


Figure 8: ROC curve for random forests, full data set. The AUC is given in the legend.

4.2.2 PCA reduced data set

Table 4 shows our best-fit hyperparameters for the PCA reduced data set. Some of the hyperparameter heatmaps can be found in appendix B, figure 18. We note that compared to the full data set, there is a higher amount of trees made. The trees are less deep (max depth 3 vs 7), however a higher amount of features is considered for each tree, which along with the fact that there are

less features in total, means that 37% of all dimensions are considered for each tree, compared to 3.5% for the full set.

Table 4: Best-fit hyperparameters for random forests. PCA reduced data set.

Parameter	Value
n estimators	25
max depth	3
min samples split	2
min samples leaf	8
max features	13

Figure 9 shows the confusion matrix for the random forest with the PCA reduced data set. We see that the predictive power drops for the cat class, however the dog accuracy increases. Similar to the full data set, the results are the opposite of the equivalent neural net result, with the random forest being better at classifying the cat samples rather than dogs.

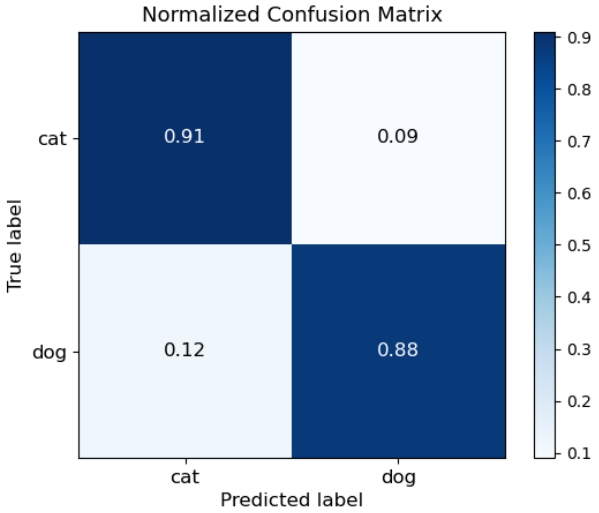


Figure 9: Confusion matrix for random forests, PCA reduced data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 10 shows the ROC curves for the PCA reduced data set. Here, the curves for both classes are very close to each other, which kinda makes sense given the similar accuracy seen in the confusion matrix. The AUC however is higher than it was for the full data set.

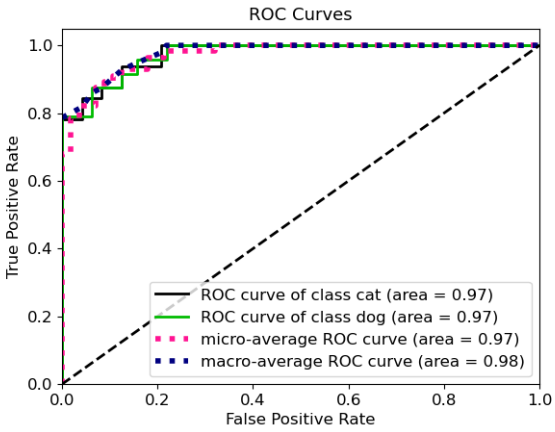


Figure 10: ROC curve for random forests, PCA reduced data set. The AUC is given in the legend.

4.3 XGBoost

Finally, we move on to XGBoost, and find our optimal hyperparameters by performing grid search using the hyperparameter sets found in appendix C.

4.3.1 Full data set

Table 5 shows our best-fit hyperparameters for XGBoost. Some of the hyperparameter heatmaps can be found in appendix C, figure 19. We see that no regularization is preferred. Testing a larger set of learning rates might have been useful.

Table 5: Best-fit hyperparameters for XGBoost. Full data set.

Parameter	Value
learning rate	0.2
n estimators	100
max depth	4
min child weight	7
lambda	0.0

From the best-fit hyperparameters, we get the following confusion matrix for XGBoost, shown in figure 11. We see that XGBoost performs extremely well for both classes, with a 95% accuracy for both classes combined. Like with random forests, XGBoost performs better at classifying cat samples.

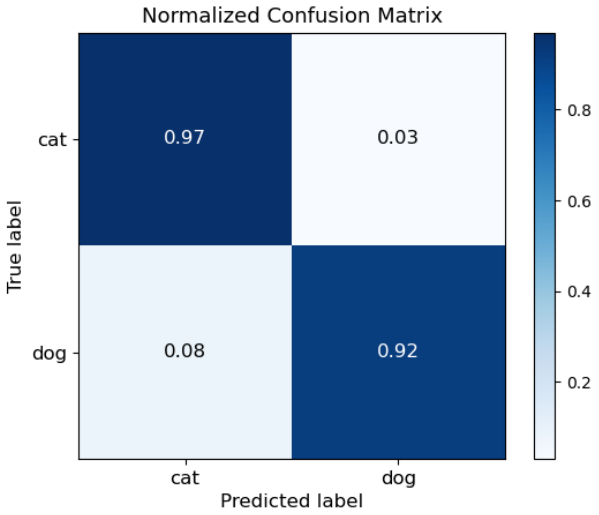


Figure 11: Confusion matrix for XGBoost, full data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 12 shows the ROC curves and AUC score for XGBoost. Like previously, it is hard to gauge anything useful from the plot, beyond it performing well.

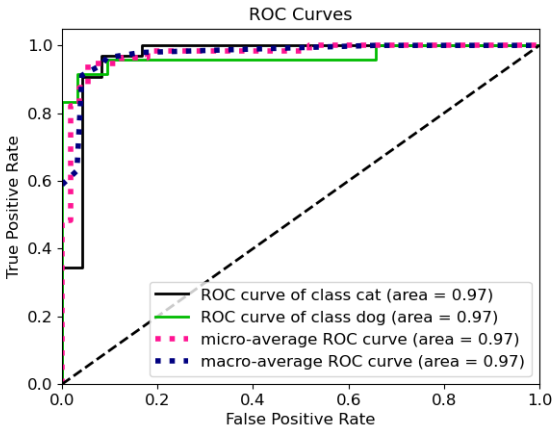


Figure 12: ROC curve for XGBoost, full data set. The AUC is given in the legend.

4.3.2 PCA reduced data set

Table 6 shows our best-fit hyperparameters for XGBoost with the PCA reduced data set. Some of the hyperparameter heatmaps can be found in appendix C, figure 20. We note that a smaller set of shallower trees is preferred, with the amount of estimators almost matching the reduction in dimensionality.

Table 6: Best-fit hyperparameters for XGBoost. PCA reduced data set.

Parameter	Value
learning rate	0.3
n estimators	20
max depth	2
min child weight	1
lambda	0.0

Figure 13 shows the confusion matrix with XGBoost on the PCA reduced data set. The correctly guessed cat samples decreases by 3 percentage points, while the dog accuracy remains unchanged. The total accuracy drops slightly, but it still outperforms the neural net and random forests.

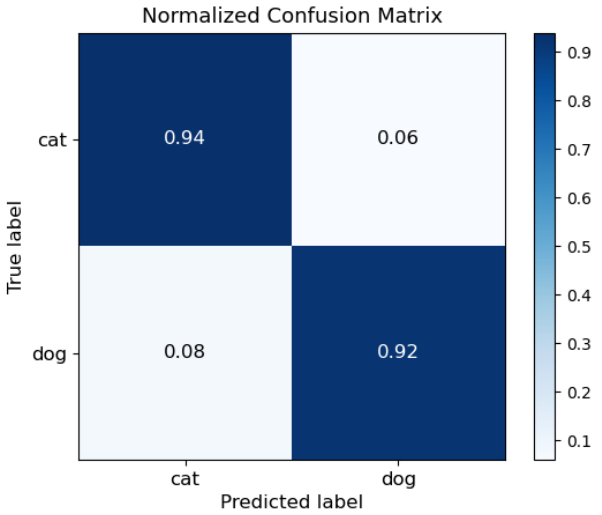


Figure 13: Confusion matrix for XGBoost, PCA reduced data set. The rows are normalized, showing the ratio of correctly/incorrectly labeled samples for each class.

Figure 14 shows the ROC curves and AUC score for the PCA reduced data set. Like with neural net and random forests, the ROC curves and AUC scores seem to be better for the dimensionality reduced data set, despite the model accuracy being lower in all cases.

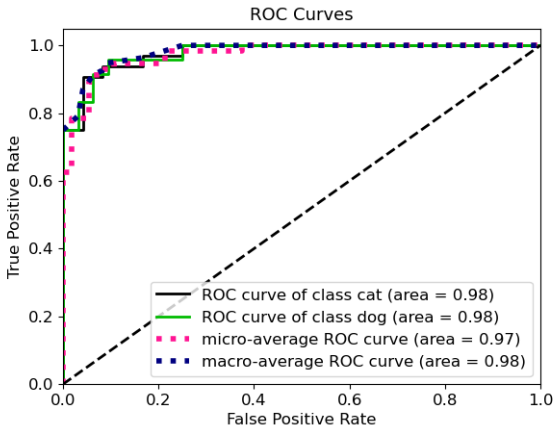


Figure 14: ROC curve for XGBoost, PCA reduced data set. The AUC is given in the legend.

4.4 Time

Table 7 shows the average time the training/fit process took for each of the three classification methods, for both versions of the data set. We see that for the full data set, random forest is the fastest method, being roughly 3 times as fast for training compared to XGBoost. Since we used the parameter setting for number of decision trees for both of them, that is not the reason for the discrepancy either. The neural net on the other hand was extremely slow.

When using the PCA reduced data set both the neural net and XGBoost decreased their run time by a decent amount. This data set has roughly 1/6th as many dimensions as the full one, with neural net speeding up by 2.25 times, and XGBoost at 4.03 times faster, so the run times don't scale linearly with predictor size. Curiously enough, the random forest training time does not change.

Table 7: Average fit/training time for the three different methods examined, given in milliseconds.

Method	Time without PCA $[ms]$	Time with PCA $[ms]$
Neural net	732	325
Random forest	28.4	28.3
XGBoost	83.9	20.8

As the parameter grid search has been run in parallel on all CPU threads, this test does end up somewhat depending on computer load, and while we attempted to limit other things running at the same time, its hard to say exactly how accurate they are, beyond the fact that neural nets were significantly slower.

4.5 Summary of results

The results are summarized in table 8, showing the total accuracy and the ROC AUC for each of the three methods. We see that neural nets and random forests perform the same when looking at the

accuracy, however there is a small improvement with the AUC when using the PCA reduced data set. Given the runtime differences between FFNN and RF mentioned earlier, and the convergence issues, random forest is the best option of the two. XGBoost performs better than both of them however, performing at over 90% for both data set versions.

Table 8: Summary of the accuracy and AUC scores for each of the classification methods. The AUC score corresponds to the micro-average found in the ROC plots. The micro-average is used as there is a class imbalance due to there being more cat samples than dogs.

Method	Accuracy	ROC AUC	Accuracy (PCA)	ROC AUC (PCA)
Neural net	0.911	0.92	0.893	0.95
Random forest	0.911	0.95	0.893	0.97
XGBoost	0.946	0.97	0.929	0.97

5 Conclusion

In this project, we set out to investigate how a frequency-space approach to classifying different animal sounds performs when using feed-forward neural nets, random forests, and the gradient boosting method XGBoost. The data set was a set of audio files containing cat and dog sounds, that were transformed into frequency space using Fourier transform, and then binned into 40 Hz bins. We also tested PCA to see if further reducing the amount of dimensions yielded improvements to either accuracy or run time.

For the full data set we find that both the feed-forward neural net and the random forest performs exactly the same, with an accuracy of 91%. Random forest has a slightly better AUC score, however all ROC/AUC results seem odd. XGBoost performs even better than both of them with an accuracy of almost 95%.

For the PCA reduced data set, very similar differences were found, with everything shifted downwards accuracy wise. Neural nets and random forests got an accuracy score of 89%, while XGBoost also here performed the best with an accuracy of 93%.

For both data sets, FFNN classified dogs better than cats, while RF and XGBoost classified cats more accurately than dogs. The difference between the classes decreased however with the PCA reduced data set for all methods. Time wise, FFNN was by far the slowest, while RF was faster than XGBoost for the full data set. For the reduced data set, XGBoost was the fastest, while RF remained unchanged.

For future work it would have been interesting to create an ensemble method consisting of neural nets and either of the other methods, as the former was better at classifying dogs, while random forests and XGBoost performed better when classifying cats, so this might lead us to an even better performing classifier. A closer look at the frequency spectra for the misclassified samples might have yielded some insight into the classification differences. It might also be beneficial to use something else than grid search in order to find better hyperparameters. Finally, it would be interesting to see how well our method performs on the full data set that [7] used, as it contained more classes than just the two used here.

References

[1] Marc Moreaux. Audio Cats and Dogs. `https://www.kaggle.com/mmoreaux/audio-cats-and-dogs`. (accessed 20. November 2020).

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly, first edition edition, 2017.

[3] GitHub repository, Project 2. `https://github.com/simennb/fysstk4155-project2`.

[4] Scikit-Learn: MLPClassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`. (accessed 10. December 2020).

[5] Tianqi Chen and Carlos Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016.

[6] Blaise Hanczar, Jianping Hua, Chao Sima, John Weinstein, Michael Bittner, and Edward R. Dougherty. Small-sample precision of ROC-related estimates. *Bioinformatics*, 26(6):822–830, 02 2010.

[7] Naoya Takahashi, Michael Gygli, Beat Pfister, and Luc Van Gool. Deep convolutional neural networks and data augmentation for acoustic event recognition. In *Interspeech 2016*, pages 2982–2986, 2016.

[8] Huy Phan, Philipp Koch, Fabrice Katzberg, Marco Maass, Radoslaw Mazur, and Alfred Mertins. Audio scene classification with deep recurrent neural networks, 2017.

[9] GitHub repository, Project 3. `https://github.com/simennb/fysstk4155-project3`.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

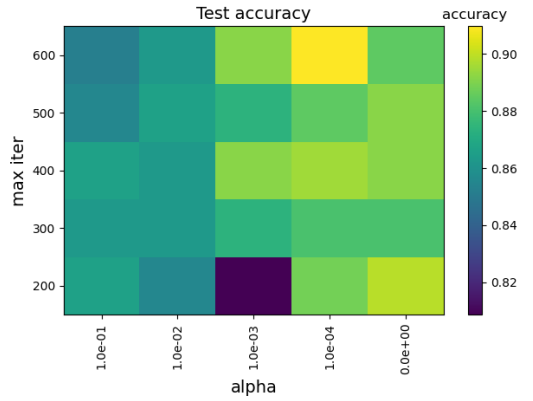
Appendix

A Neural Net Heatmaps

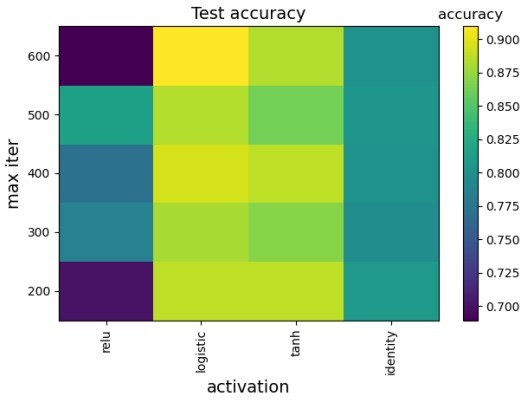
The following hyperparameter sets were used for finding best fit parameters as well as heatmaps below.

$$
\begin{aligned}
\text{n hidden neurons} &\in \{1, 5, 10, 25, 50, 75, 100\} \\
\text{n hidden layers} &\in \{1, 2, 3, 4, 5\} \\
\text{max iterations} &\in \{200, 300, 400, 500, 600\} \\
\text{activation} &\in \{\text{relu}, \text{logistic}, \tanh, \text{identity}\} \\
\text{lambda} &\in \{0.1, 0.01, 0.001, 0.0001, 0.0\}
\end{aligned}
$$

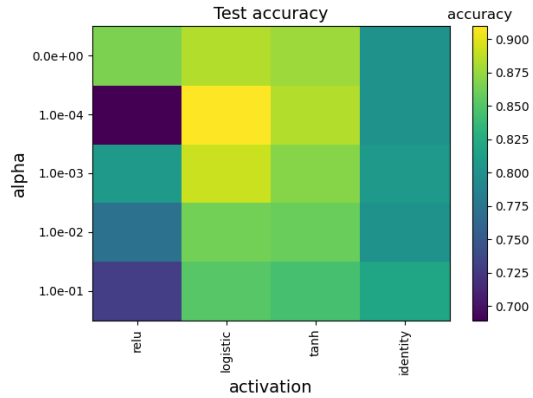
A.1 Without PCA



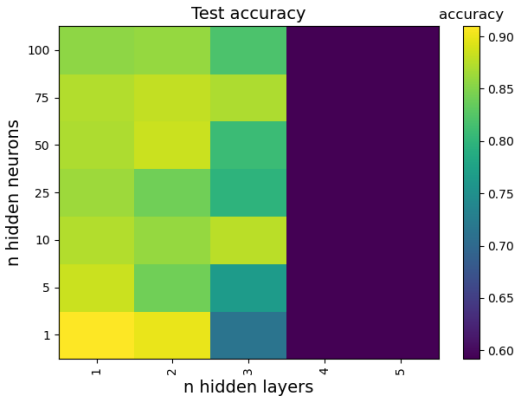
(a) Regularization vs maximum iterations



(b) Activation function vs maximum iterations



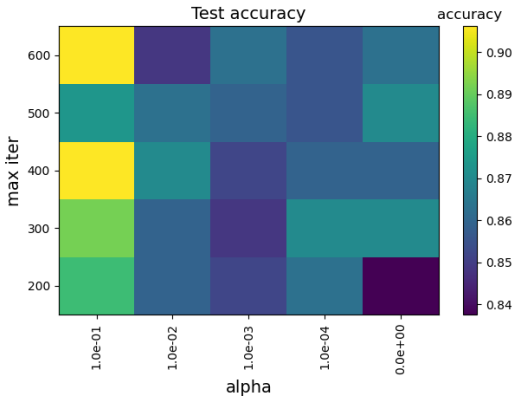
(c) Activation function vs regularization



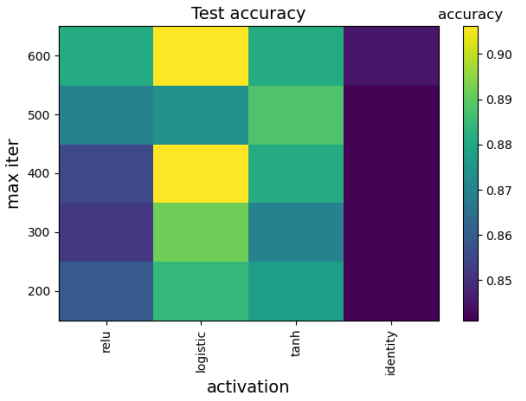
(d) Number of layers vs number of hidden neurons

Figure 15: Accuracy heatmaps for different hyperparameters. Neural net, full data set. All other hyperparameters set to the best-fit from table 1.

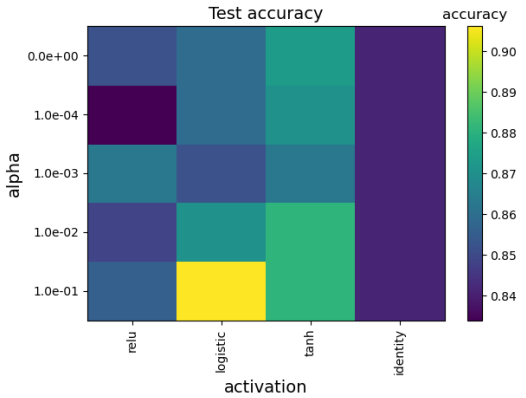
A.2 With PCA



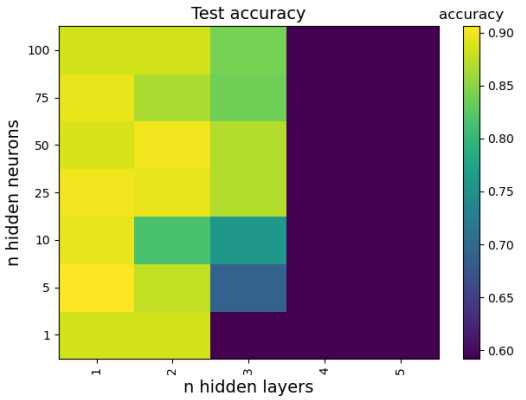
(a) Regularization vs maximum iterations



(b) Activation function vs maximum iterations



(c) Activation function vs regularization



(d) Number of layers vs number of hidden neurons

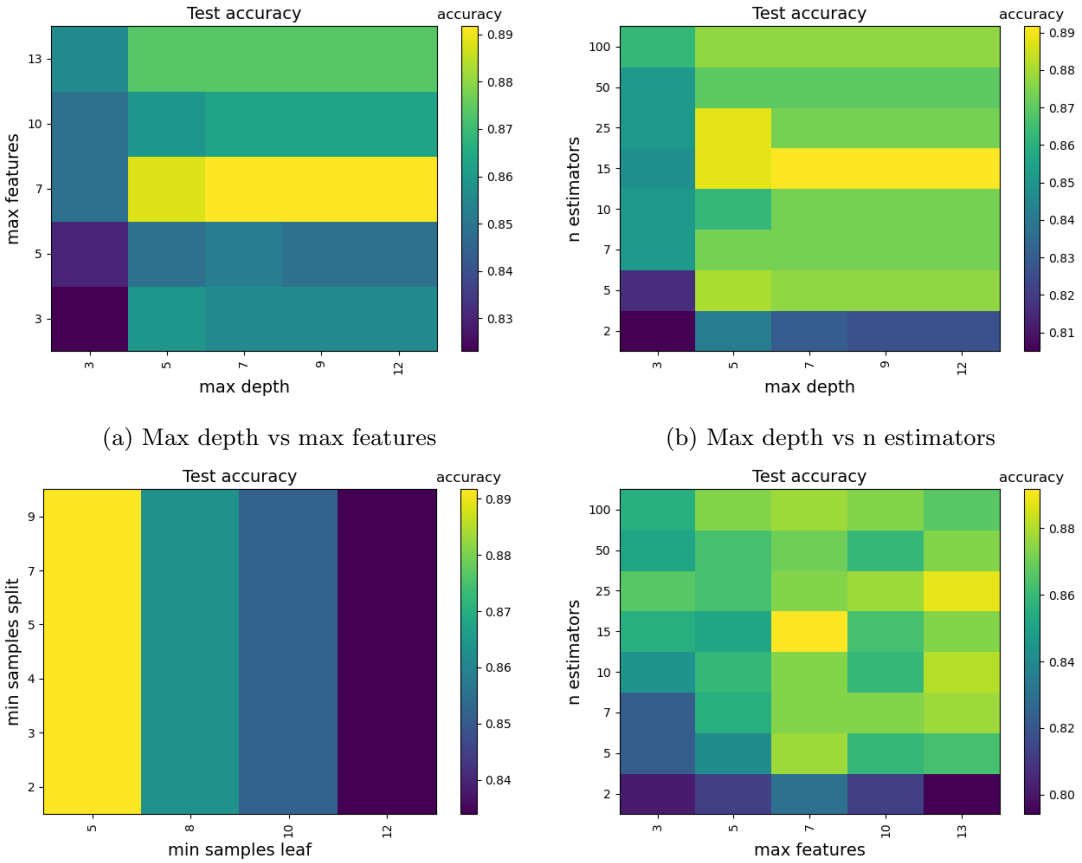
Figure 16: Accuracy heatmaps for different hyperparameters. Neural net, reduced data set. All other hyperparameters set to the best-fit from table 2.

B Random Forests Heatmaps

The following hyperparameter sets were used for finding best fit parameters as well as heatmaps below.

$$\begin{aligned}
\text{max depth} &\in \{3, 5, 7, 9, 12\} \\
\text{max features} &\in \{3, 5, 7, 10, 13\} \\
\text{min samples leaf} &\in \{5, 8, 10, 12\} \\
\text{min samples split} &\in \{2, 3, 4, 5, 7, 9\} \\
\text{n estimators} &\in \{2, 5, 7, 10, 15, 25, 50, 100\}
\end{aligned}$$

B.1 Without PCA



(a) Max depth vs max features

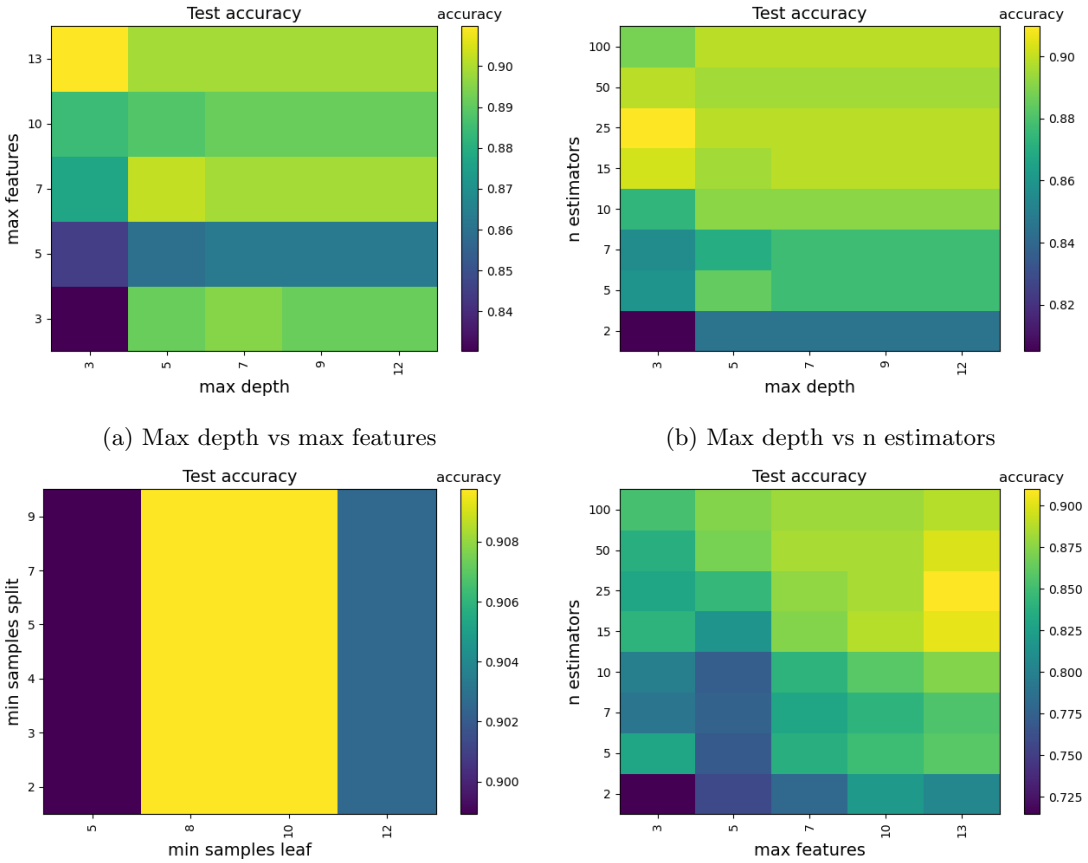
(b) Max depth vs n estimators

(c) Min samples leaf vs min samples split

(d) Max features vs n estimators

Figure 17: Accuracy heatmaps for different hyperparameters. Random forest, full data set. All other hyperparameters set to the best-fit from table 3.

B.2 With PCA



(a) Max depth vs max features

(b) Max depth vs n estimators

(c) Min samples leaf vs min samples split

(d) Max features vs n estimators

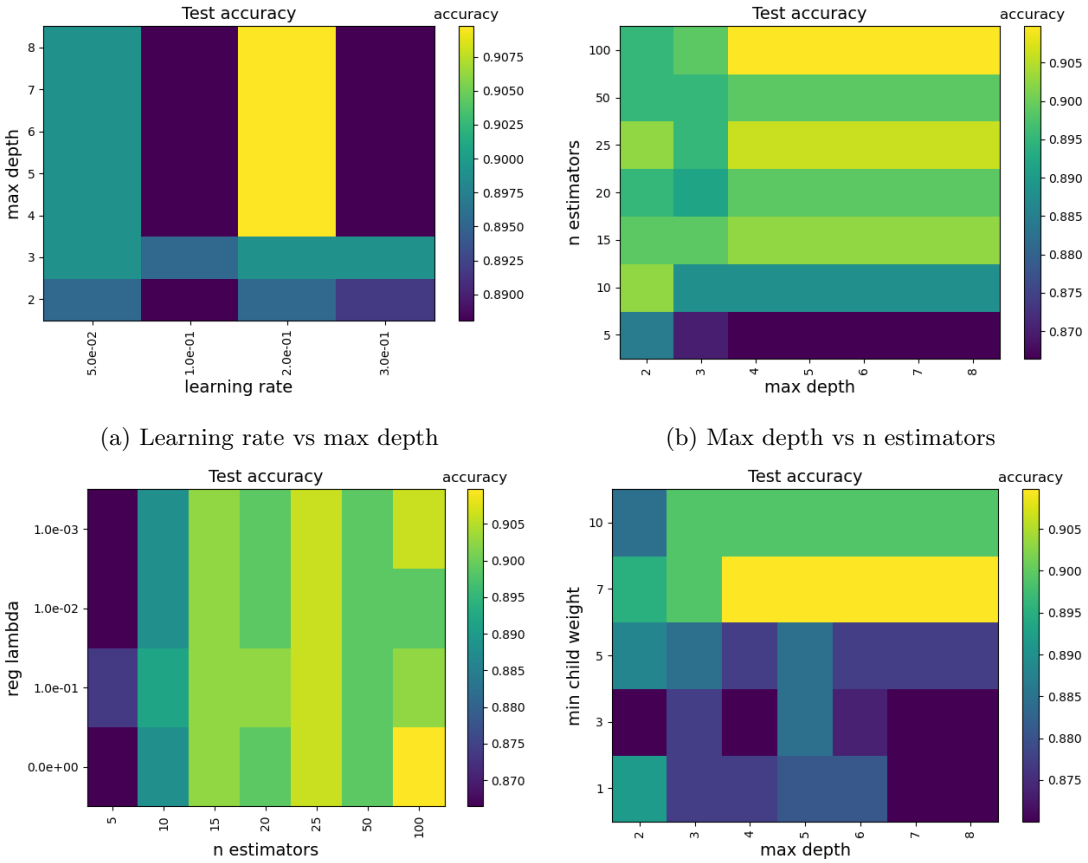
Figure 18: Accuracy heatmaps for different hyperparameters. Random forest, reduced data set. All other hyperparameters set to the best-fit from table 4.

C XGBoost Heatmaps

The following hyperparameter sets were used for finding best fit parameters as well as heatmaps below.

$$\begin{aligned} \text{learning rate} &\in \{0.05, 0.1, 0.2, 0.3\} \\ \text{n estimators} &\in \{5, 10, 15, 20, 25, 50, 100\} \\ \text{max depth} &\in \{2, 3, 4, 5, 6, 7, 8\} \\ \text{min child weight} &\in \{1, 3, 5, 7, 10\} \\ \text{lambda} &\in \{0.0, 0.1, 0.01, 0.001\} \end{aligned}$$

C.1 Without PCA



(a) Learning rate vs max depth

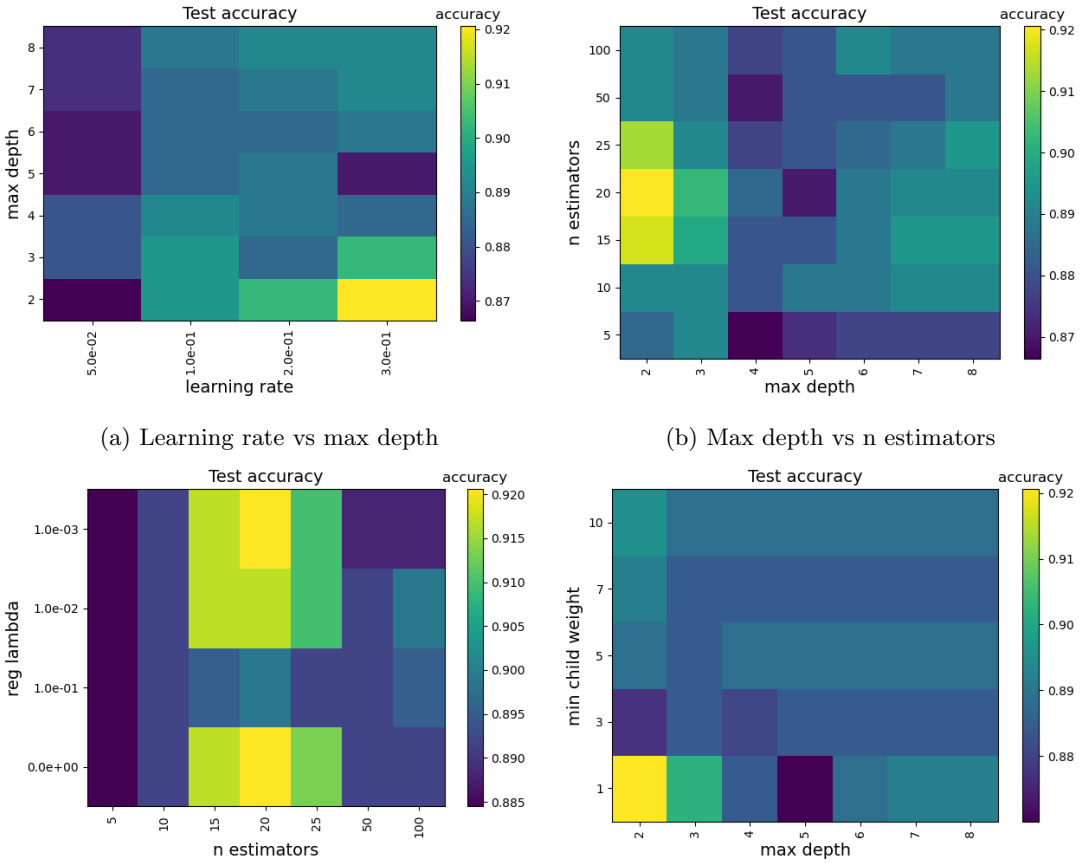
(b) Max depth vs n estimators

(c) Regularization vs n estimators

(d) Max depth vs min child weight

Figure 19: Accuracy heatmaps for different hyperparameters. XGBoost, full data set. All other hyperparameters set to the best-fit from table 5.

C.2 With PCA



(a) Learning rate vs max depth

(b) Max depth vs n estimators

(c) Regularization vs n estimators

(d) Max depth vs min child weight

Figure 20: Accuracy heatmaps for different hyperparameters. XGBoost, reduced data set. All other hyperparameters set to the best-fit from table 6.