Project Report

Elise Mol, Wendy Nieuwkamer, and Isobel Smith

21 December 2016

1 Problem

After three months of learning about several machine learning techniques and classifying algorithms it was time to put our newly gained skills into practice. We decided to try our hand at one of the competitions hosted by Kaggle. The challenge we chose is Leaf Classification; the objective of the competition is to use binary leaf images and their extracted features to predict what species each leaf belongs to. The dataset which was provided includes around 1584 images of leaf specimens; these samples include 99 species. From every image three sets of features are extracted: a shape contiguous descriptor, an interior texture histogram and a fine-scale margin histogram. Every feature is represented as a 64-attribute vector.

As the competition started at 30 August 2016, there were already quite some submissions made and kernels active. One of them stood out, namely "10 Classifier Showdown in Scikit-Learn". In this kernel Jeff Delaney compared ten classifiers which are part of the python library Scikit-Learn. The results of his experiment are shown in figure 1. In his experiment Delaney used the default setup for the classifiers , combined with semi-random parameters. He notes that the performance of the classifiers could be improved by tuning the hyper-parameters. Thus, our research question is whether we can improve the accuracy found by Delaney by tuning the hyper-parameters of the classifiers.

As we did not have time to run experiments for all ten classifiers, we decided to look into five of them. We chose the K-neigbors classifier, decision tree classifier and Gaussian Naive Bayes as they had relatively decent accuracy at respectively 88%, 65%, and 57%. Also, we had some background in these algorithms as they had been taught in our Machine Learning course. Additionally, we decided to choose either Ada boost or quadratic discriminant analysis as they could improve the most with found accuracies of 4.5% and 2.5%.

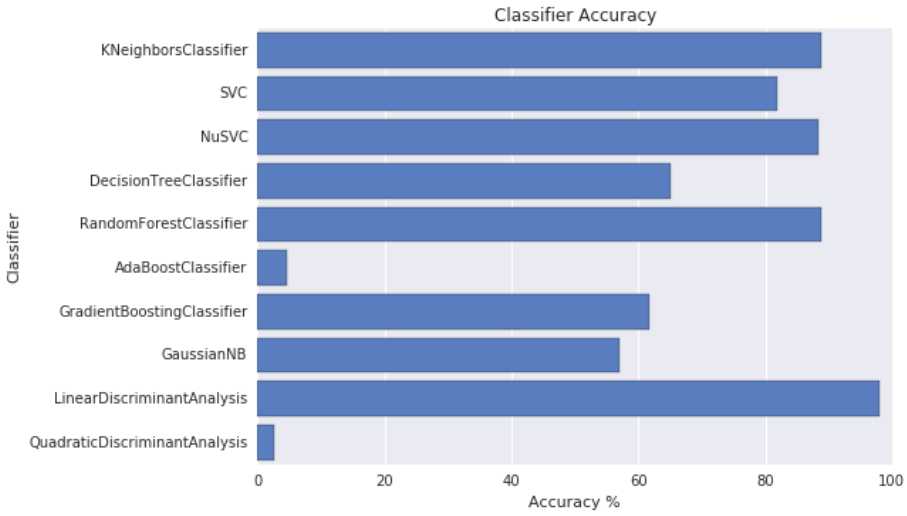


Figure 1: The 10 Classifier Showdown

2 The algorithms

Before we could start optimizing the hyper-parameters we had to research the algorithms we had chosen. In order to discover the most relevant information for our experiment we aimed to answer the following questions:

- How does the classifier work?
- Which hyper-parameters exist for each classifier?
- Which hyper-parameters have the most effect on the classifier?

The answers to these questions are specific to the Scikit-Learn package. In this section we will discuss how the classifiers work. More information on the hyper-parameters we chose to optimize can be found in the next section.

K-neighbors The k-neighbors algorithm classifies an object according to the k nearest objects from the training set. For a higher k noise might be filtered out, but class boundaries might become more vague and less accurate. Thus, it would be interesting to find out which value of k gives the highest accuracy for our dataset. There are two ways of computing the class from the neighbors. The first one is by giving all the neighbors equal weight, no matter how far or close they are. The second is to give each neighbor a weight relative to its distance. The weighted version of the algorithm is especially useful when a dataset has areas with a varying density of training examples.

Decision Tree Classifier Decision trees is a very intuitive method of doing classification. When learning a Decision tree a set of rules is constructed, which can be used to make predictions. There are two criteria to construct these rules which are often used. the first is information gain, where the split with the highest information gain is added to the tree. The second is Gini Impurity which looks at possible incorrect labeling. There are several advantages to using decision trees, such as the fact that they are very simple to understand, they require little data preparation and they are able to handle multiple outputs. However, it is important to avoid over-fitting by setting a maximum depth and/or a minimum amount of samples per leaf, especially in datasets with many features and few samples. Also, they work best on a balanced dataset, like the dataset provided by Kaggle for leaf classification. Finally, creating the optimal decision tree is very hard, so it is often best to use it in an ensemble, such as the RandomForestClassifier. Delaney also tried the RandomForestClassifier, but we chose the DecisionTreeClassifier as the initial accuracy was lower.

Gaussian Naive Bayes Gaussian Naive Bayes is a classifier which uses probability to assign classes. In doing this all features equally contribute to the assigned class. The nature of this classifier means that there are no hyper-parameters to optimize, thus we decided not to include it in our experiments.

Ada Boost Classifier AdaBoost stands for adaptive Boosting and is used to improve other algorithms' performances The goal is to create a strong classifier from weak classifiers, traditionally it is used with decision trees. It does this by fitting the classifier and then giving the examples which were wrongly classified a higher weight, which increases the performance on difficult samples. Thus, it improves the classifier by only using those features which actually improve the performance of the classifier, thus reducing dimensionality and possibly computation time. However, this also means that it is sensitive to outliers and noisy data.

Quadratic Discriminant Analysis Quadratic Discriminant Analysis is a classical classifier which uses Bayes' rule and a Gaussian distribution. Like the Gaussian Naive Bayes classifier there are no parameters to optimize, so we decided not to include it in our experiments.

3 Hyper-parameter optimization

Hyper-parameter optimization is one of the most important steps in machine learning [1]. The goal is to find the best hyper-parameters for the given classifier, in order to minimize the loss function and to avoid over-fitting [2]. Some hyper-parameter optimization algorithms do not only find the best hyper-parameters, but identify those that carry the most weight [2]. There are two hyper-parameter optimization algorithms built into Scikit-Learn: grid search and randomized

parameter optimization [3]. Scikit-Learn also provides alternative methods such as model specific cross validation [**?**], but we did not use those.

3.1 Grid Search

Grid Search optimizes hyper-parameters by iterating over all of the variables in a given range and selecting the best combination to use in the chosen classifier. Grid search is guided by a performance metric, such as classification accuracy, which is usually measured by cross-validation [**?**]. A disadvantage of grid search is that it can be time consuming and computationally expensive to run, [**?**].

As grid search was recommended to us by our teacher Rein van den Boomgaard, and is the widely used to optimize hyper-parameters, we decided to use it in our project. The use of Grid Search was easy as Scikit-Learn has a built-in class called GridSearchCV [3], which is defined below:

```
GridSearchCV(estimator, param_grid, scoring=None,
    fit_params=None, n_jobs=1, iid=True, refit=True,
    cv=None, verbose=0, pre_dispatch='2*n_jobs',
    error_score='raise', return_train_score=True)
```

When an instance of GridSearch is initialized, it takes an estimator, a param_grid and any other parameters. The estimator would be our classification functions KNeighborsClassifier, DecisionTreeClassifier, and AdaBoostClassifier. The param_grid is a dictionary with the parameters you want to optimize as the keys, and the range of parameters to try as the values. Finding the optimal range of parameters to pass on to GridSearch is a matter of trial and error. The grid search algorithm will then run the given classifier over the entire parameter grid in order to find the best possible combination. We also chose to use 5-fold cross validation instead of the default 3-fold cross validation by passing cv=5 to GridSearchCV.

3.2 Implementation

Before running the GridSearch we had to prepare our dataset. To do this we used Delaney's code [4], which used a StratifiedShuffleSplit to divide the dataset in such a way that both the test set and the training set contain all classes. The code was updated to fit our needs and to use updated packages. Then, we defined the param_grids for each classifier and ran the GridSearch. We printed the output of GridSearch, which is a dictionary with the names of the hyper-parameter as keys, and the optimal result as values. Finally, the data found during the GridSearch is visualized to show how different hyper-parameters influence the accuracy of the classifier.

KNeighborsClassifier For the K-Neighbors classifier we chose to optimize the hyper-parameters n_neighbors in the range from 1 to 50 for uniform weights and using distance as a weight. Metric is the way distance is calculated, the default is 'minkowski', which is basically the euclidean distance if p is two.

We saw no reason to change this for our data, thus we excluded metric, metric_params and p from the GridSearch. Scikit-Learn also offers the following hyper-parameters: algorithm, leaf_size, and n_jobs. However, we chose not to optimize those as they either optimize themselves (algorithm), affect speed (leaf_size), or refer to the number of CPU cores used (n_jobs).

DecisionTreeClassifier For the Decision Tree classifier we chose to optimize criterion, splitter and min_samples_leaf. Criterion can be gini or entropy, split either chooses the best split, or a random split from the best ones and min_samples_leaf decides how many samples each leaf should minimally have. As our dataset is quite small we chose a range of 1 to 10 for min_samples_leaf. We did not optimize max_depth, max_leaf_nodes or min_samples_split because we already optimize min_samples_leaf. We did not optimize min_weight_fraction_leaf, max_features, random_state, min_impurity_split or class_weight because they are not relevant to our data. Finally, we were interested in presort, but if it is set to True it would take a lot of time to train.

AdaBoostClassifier For the AdaBoostClassifier we chose to optimize n_estimators and learning_rate. n_estimators is the maximum amount of estimators used; more estimators means a better fit, so we tried a range from 25 to 75 estimators. learning_rate is the rate at which contributions of the consecutive classifiers shrinks, this is linked to the amount of estimators used. Finding the proper range for learning_rate was quite hard, but finally we settled on 0.01 to 0.09 in 9 steps. It does not fit within the scope of our research to use a different base_estimator. Changing algorithm would slow the process down, so this is excluded. Finally, we do not change random_state as the random number generator should not have a big effect on the results.

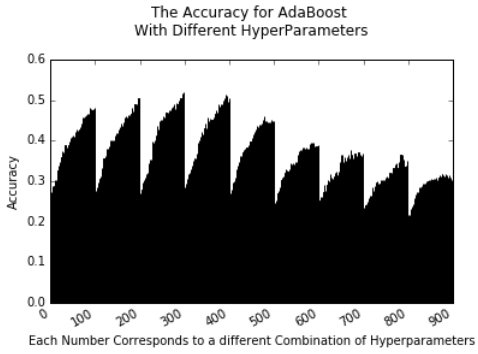


Figure 2: GridSearch AdaBoost

4 Results

AdaBoost Figure 2 shows how grid search across each of the different hyper-parameter combinations improved the accuracy of AdaBoost. The numbers on the x-axis corresponds to a different combination of hyper-parameters. As can be clearly seen from the graph the amount of possible combinations was very high. This was computationally quite expensive, and running grid search on AdaBoost took a long time compared to the other classifiers.

The optimal hyper-parameters for AdaBoost were found to be $n\_estimators = 72$ with $learning\_rate = 0.03$

Decision Trees The accuracy of decision trees were comparatively not as improved as much as the accuracy of the other two were. This was because the default parameters used in the 10 classifier showdown [4] happened to be similar to the optimal parameters as found by grid search.

Interestingly, the difference between grid search using entropy, or using Gini was minimal, as can be seen in figure 3. This suggests that for this problem the splitting criteria hyper-parameter does not hold much weight as other hyper-parameters might.

The optimal hyper-parameters found for decision trees were to be $splitter =' best', criterion =' entropy'$ and $min\_samples\_leaf =' 1'$

K-Neighbors One of the hyper-parameters we needed to tune for k-neighbors were the weights. We needed to choose between using uniform weights or distance as weights. The accuracy graphs in figure 4 clearly show that the accuracy rate for k-neighbors with uniform weights quickly tails of when the number of neighbors increases, whereas the accuracy when using distance as weights remains relatively high as the number of neighbors increases.

The optimal hyper-parameters chosen for K-Neighbors was that $n\_neighbors = 1$, and $weights =' uniform'$

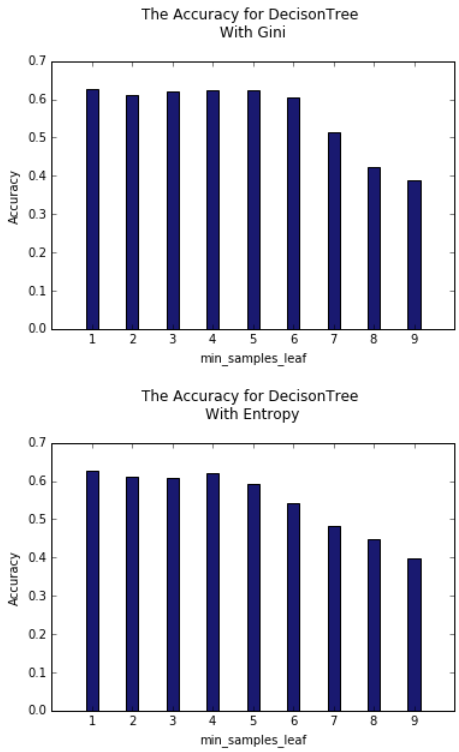


Figure 3: GridSearch Decision Tree

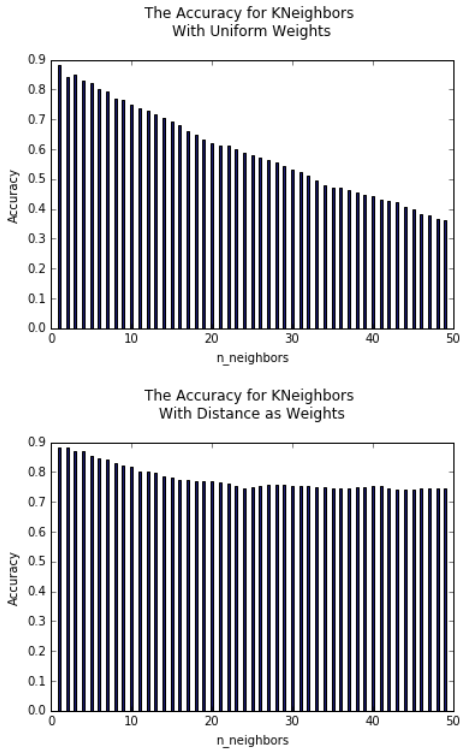
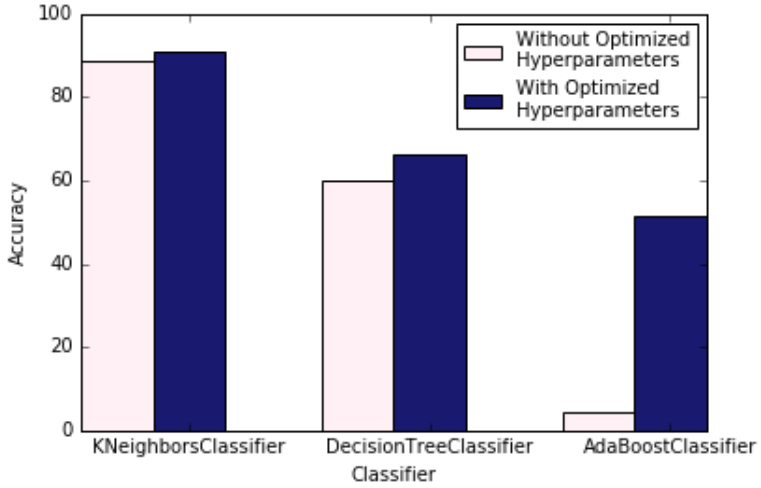


Figure 4: GridSearch KNeigbors

Optimized Accuracies The accuracy of classifiers with optimized hyper-parameters were graphed against those without optimized hyper-parameters:



In each of the cases, grid search clearly improved the accuracy of the classifier, reaffirming the importance of hyper-parameter optimization in machine learning. However, none of the classifiers reached the 98.0% which was reached by LinearDiscriminantAnalysis in Delaney's experiment. AdaBoost was significantly improved by hyper-parameter optimization, from 4.56% to 51.52%. This is due to the fact that the default parameters that were used in the 10 Classifier Showdown [4] were inadequate for the leaf classification problem. K-neighbors was improved from 88.89% to 90.91%, and Decision Trees were improved from 60.10% to 66.16%.

5 Evaluation of Approach and Solution

The goal of our experiments was to improve the accuracy scores of some of the classifiers used in the 10 classifier showdown by optimizing their hyper-parameters [4]. We have reached this goal, although we did not reach the 98.0% which was reached by LinearDiscriminantAnalysis in Delaney's experiment. Still, we encountered a few challenges, one of which was the time it took to run GridSearch on a range of values. A solution for this could be to use another hyper-parameter optimization algorithm, such as randomized parameter optimization [3]. Instead of iterating over each potential combination of parameters, like grid search, it randomly selects these combinations. An extra advatage of this is that adding parameters which do not influence the performance of the classifier does not decrease the algorithms efficiency. The number of random samples that are tried can be specified. As AdaBoost had such a high amount of hyper-parameter combinations, using randomized parameter optimization in this case may reduced the computation time.

Another challenge was the selection of the range of hyper-parameters to test; we had to use trial an error, but a systematic approach might improve the results even more. Also, some of the hyper-parameters that could have been interesting, such as presort of the DecisionTreeClassifier, were not optimized as there was no time. Finally, it would be interesting whether GridSearch gives different results when a different value for cv is given, such as a higher number of folds or a different cross validation set.

References

[1] R. Bardenet, M. Brendel, B. Kegl, and M. Sebag. Collaborative hyperparameter tuning. *30th Annual Conference on Machine Learning.*, 28:199–207, 2013.

[2] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[3] SciKitLearn. Gridsearchcv. [Online; accessed 15-December-2016].

[4] J. Delaney. 10 classifier showdown in scikit-learn. [Online; accessed 18-December-2016].